

The Paved Road

The manifests behind The Paved Road and the reasoning that decides each one.

CONTENTS

- 01 The platform is a product, and adoption is the only metric
- 02 Adoption is earned, not mandated
- 03 Standardization is only standardization if it's the default
- 04 Security has to live on the paved road, or it gets bypassed
- 05 Instrument the platform like a product, not like infrastructure
- 06 Guardrails, not gates

HOW TO READ THIS

→ *One idea runs under every move here: a platform is a product, its only customers are your own engineers, and it succeeds on adoption rather than architecture. The job is to make the right way the easy way. This guide carries that case and then gets concrete, ending each chapter with what the platform actually ships. Additionally, a Leadership Edition is written for the VP of Engineering and CTO who fund the work, and section numbers match across both so you can hand a chapter sideways without translating it. You don't need it to use this one.*

The example we'll keep coming back to

Abstractions are easy to nod along to and hard to act on, so this guide is anchored to one situation we'll return to in every chapter. It's composed from the kind of environment platform teams inherit constantly:

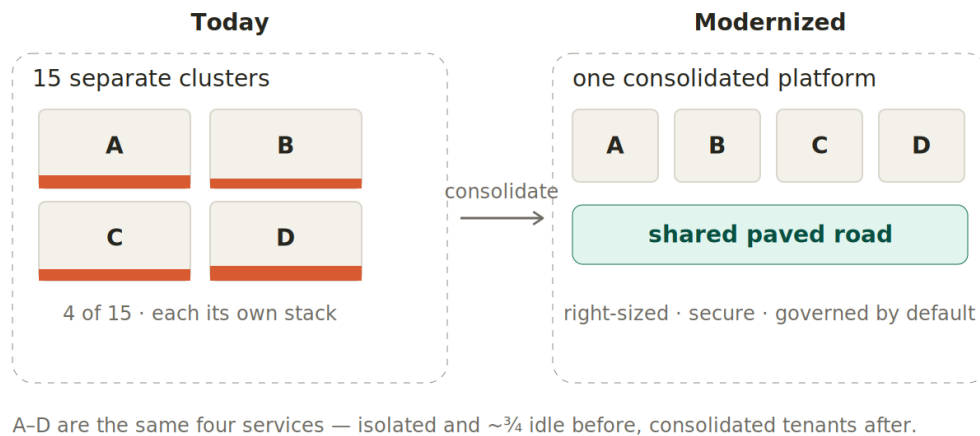
STARTING POINT — 15 EKS CLUSTERS

Fifteen separate teams, each running their own cluster, each with their own way of working — their own Terraform (or clickops), their own CI, their own RBAC, their own monitoring, their own quiet bypasses for the controls that got in their way. Across all 15, roughly **three-quarters of the provisioned compute is paid for and idle**, because every team independently sized for peak-plus-fear and nobody trusted autoscaling they hadn't configured themselves.

THE GOAL

Consolidate this into one modernized platform — a shared paved road that's standardized, observable, and governed — that's genuinely more manageable than the sprawl it replaces.

If that 75%-idle figure sounds exaggerated, it's roughly in line with what large-scale measurement reports — and from more than one source. The most-cited vendor benchmark, CAST AI's across tens of thousands of production clusters, puts average CPU utilization at 8–13% and memory around 20%. Independently, Datadog's 2025 *State of Containers and Serverless* — drawn from thousands of companies — found most workloads use under 25% of the CPU they request and less than half the memory they request: a different metric (against requests, not provisioned capacity) pointing the same way. Two caveats worth knowing before you quote this number: the CAST AI figure comes from a vendor that sells the optimization product the number justifies, so treat the exact percentage as directional rather than gospel; and the only utilization figure anyone will accept in review is the one you measured against your own clusters. Run it on your fleet before you put a percentage on a slide. But the direction isn't in dispute across independent sources: most clusters waste most of their compute, and the 15-cluster example below is, if anything, generous to the status quo.



A-D are the same four services — isolated and $\sim\frac{3}{4}$ idle before, consolidated tenants after.

Fifteen fragmented, mostly idle clusters become tenants on one shared road. Services A-D track across the consolidation.

Sections 3, 4, and 6 carry the manifests. Sections 1, 2, and 5 carry the operating decisions that determine whether anyone uses them. Jump to whatever you're building.

CHAPTER 01

The platform is a product, and adoption is the only metric

In one line a platform you built but nobody chose is shelfware — the only metric that survives contact with reality is whether engineers reach for it or around it.

In one line security that lives off the paved road gets routed around — bake the baseline into the default path so the secure way is also the easy way.

The thesis of this entire guide is simple and ruthless: a platform is a product, its only customers are your engineers, and it succeeds or fails on adoption. Not architecture. Not uptime. Not the number of services onboarded. Adoption. The 15-cluster situation this companion uses throughout is the proof: somewhere behind each of those clusters is a moment when a team decided DIY was faster than the shared path. Your platform has to reverse those decisions, and the only way it does that is by being faster — every time, including your engineers' worst days.

Why developer experience is the entire game

"Developer experience" tends to get treated as a quality-of-life concern — happier engineers, better retention. That framing undersells it. DevEx is your adoption engine. It is the only one you have. Nobody migrates off a working cluster because leadership asked; they migrate because the new platform beats the thing they're already running, every time,

with no asterisks. That bar is high, and it's worth being honest about how high. The paved road has to beat the workaround on your engineers' worst day — tired, under deadline, the old way is muscle memory. If your golden path only wins when everything goes right, you lose every time something goes wrong, which is exactly when habits get formed.

Three things determine whether it does:

Time to first success.

From "I have a service to run" to "it's serving traffic in a real environment" — measure it in minutes on the happy path. For migrating teams, the equivalent is time-to-parity: how long until the new platform can do everything their current cluster does? If that's measured in weeks of platform-team tickets, team 7 keeps their cluster. You've earned it.

Cognitive load removed, not relocated.

A platform earns its keep by reducing what an engineer has to hold in their head to ship safely. If you've moved the complexity — they no longer hand-roll the node group but now must learn your custom abstraction over node groups — you've added a layer, not removed one. Sometimes that trade is worth it.

Often it isn't. Be honest about which.

Escape hatches that don't punish you for using them.

The fastest way to kill adoption is to make the paved road a prison. One of those 15 teams has a genuinely weird workload — a GPU job, a licensing constraint, a latency requirement — and if your platform's only answers are "wait for us to support it" or "leave entirely," they'll leave. A good platform lets you drop down a level and climb back up without penalty.

Self-service is the test, not a feature

"Self-service capabilities" shows up on every platform checklist as a box to tick. Reframe it: self-service is the test of whether you've built a platform or just a team that provides infrastructure on request for other people on request.

The diagnostic: pick a routine task — provision a namespace with the right quotas, stand up a new environment, rotate a credential. Can an engineer who's never done it complete it correctly and safely without opening a ticket or pinging you on Slack? If yes, that's a platform. If the answer is "well, they could, but they usually ask us," you've rebuilt one of the 15 clusters' support models with extra centralization, and it won't scale beyond your team's attention span.

Self-service only works if the abstraction is right-sized: hiding enough that the common case is trivial, exposing enough that the engineer still understands what they're creating. Hide too little, and you haven't helped. Hide too much, and the first time something breaks, nobody — including the platform team — can reason about what's running. The goal isn't to make infrastructure invisible. It's to make the safe version of it the obvious one.

What this costs to get wrong

A consolidated platform the 15 teams don't actually adopt is worse than the sprawl you started with, and the first cost lands on you. Every control in Chapters 4 and 6 assumes engineers are on the paved road. A half-migrated fleet — some workloads on the platform,

some still on old clusters, some in between— is the worst possible thing to secure, debug, and audit: two sets of RBAC, two network models, two on-call surfaces, and no baseline of normal in either. You also end up operating three environments instead of one, because the old clusters never get decommissioned and teams stand up shadow infrastructure to bridge the gap. And you've taught 15 teams to distrust the platform, which makes the second attempt harder than the first.

The one consequence for your build: instrument adoption, not just health. That's Chapter 5.

FAILURE MODE TO WATCH FOR

Measuring the platform by what it contains — clusters consolidated, services onboarded, dashboards built — instead of by what engineers actually do. The first set of numbers always goes up and to the right. The second set is the only one that tells you the truth. "We migrated 12 of 15 clusters" feels like progress; "3 teams are still quietly running production on the old clusters because ours is slower" is the fact that matters.

CHAPTER 02

Adoption is earned, not mandated

In one line you can't mandate adoption into being — you sequence migrations so the platform earns it: easy and visible workloads first, the hard holdouts last.

A mandate can choose the destination. Only the platform makes arrival painless — and a mandate without a road is one teams quietly ignore. The 15-cluster situation is the proof. Behind each of those clusters is a mandate that didn't hold: a "we're all standardizing on X" that one team opted out of because X was slower for them. A migration deadline that slipped because the deadline was the only thing making it happen. If mandates alone worked, you'd have one cluster, not 15.

There's a real exception: some consolidation is mandated and does stick — security and compliance baselines hold because the alternative is an audit finding. But notice what the mandate actually achieves even there: it sets what must be true, not whether engineers route around the platform to make it true. Mandate and pull aren't opposites; they're two halves of the same move. The mandate names the destination. The road is what gets people there willingly, which is the only way they stay.

Migration happens by pull, not push, so sequence it for proof

You can't push 15 teams onto the platform. You have to pull them. The order in which you migrate is the single biggest lever for whether that pull ever starts. Get the first few migrations right, and the rest start recruiting themselves; get them wrong, and you've publicly proven the skeptics' point.

Two approaches, de-risking different things:

Easiest-first: de-risk the platform

Start with the simplest workloads: a grouping of similar, low-stakes services — stateless HTTP apps before anything with persistent data, stateful dependencies, or unusual constraints. You prove the paved road works on cases that can't hurt you before you bet a hard workload on it; you shake out the platform's bugs cheaply; and when you batch by stack similarity, you build the migration path once and reuse it across every service that looks like the first one. Your real unit of sequencing isn't the team — it's the workload archetype. Solve "stateless HTTP service on the platform" completely, migrate all of them, then move to the next pattern. Fast wins, low blast radius, reusable paths.

Hardest / most-pain-first: de-risk the belief

Start with the team whose cluster is the worst to operate, whose on-call is the most miserable, who'd most love for someone to take this off their hands. You don't do this because it's safe — it isn't. You do it when your binding constraint is skepticism rather than capability. A dramatic win on a genuinely painful workload is the most credible testimonial available, because the advocate is the last person anyone expected to convert. The cost is real: pain teams usually have the hardest workloads, so you're betting the platform's reputation on its hardest case, first, in public. Miss and you've poisoned the well for everyone behind them.

Which to reach for

Default to easiest-first. It's the lower-risk, higher-momentum path, and you do not want to discover that the platform can't carry the real load for your highest-stakes migration. Reach for most-pain-first only when the thing blocking adoption is belief, not capability. Within early waves, weight toward teams whose success will travel — the visible ones, the respected ones, the ones sitting next to the loudest skeptics. Save the genuinely hard, weird-workload teams for last — by then the platform has earned the right to ask, and the path has been worn smooth by everyone ahead of them.

Watch for the railroad

The anti-pattern has a name: a golden path that gets mandated, forgets the "optional" part, and leaves engineers either finding workarounds or ignoring it entirely. Teams that invest in a genuinely good path report voluntary adoption north of 80%; teams that force a half-built one stall around 20%. That contrast is practitioner field observation, not a controlled study — treat it as direction, not a figure to quote — but the direction is consistent everywhere it's seen: the variable that moves adoption is whether engineers chose it.

What makes this sequencing cheaper than it used to be

A practical note on what makes this sequencing cheaper than it used to be. The slowest, most expert-dependent part of any migration is archaeology: reverse-engineering a cluster nobody fully documented to work out what it actually does before you can move it. That's where time-to-parity goes to die, and it's why the hard holdouts get put off forever. This is the part of the operating work that has genuinely changed. Code analysis that reads a cluster's real manifests, IAM bindings, and dependency graph — and drafts the migration steps from what's actually there, not what the wiki claims — turns weeks of senior-engineer spelunking into a review-and-correct loop. The same goes for social archaeology: surfacing the real blockers buried across fifteen teams' channels and tickets, so that "resistance is information" becomes something you act on at the fifteen-team scale rather than a principle you nod at. None of this *decides* anything — the sequencing, the archetypes, what gets a gate, which team goes first, all of that is still judgment. What changes is that the friction *between* the decisions gets cheaper, which is exactly what determines whether a paved road is buildable by a team you can actually staff.

Peers, not management

Whichever order you choose, the mechanism that makes pull work is the same: peers, not management. The most credible advertisement for your platform is one engineer telling another, "I moved, and my life got better," in a channel where the platform team isn't speaking. You're not running 15 migrations on a Gantt chart — you're running the first few extremely well and letting them recruit the rest.

FAILURE MODE TO WATCH FOR

Treating resistance as a communication problem. When a team won't move, the instinct is to message harder — more demos, more docs, more exec pressure. Resistance is almost always information: the platform is missing something that team actually needs, or it's genuinely slower for their case. The team that "won't get on board" is usually your most valuable bug report. Listen before you escalate.

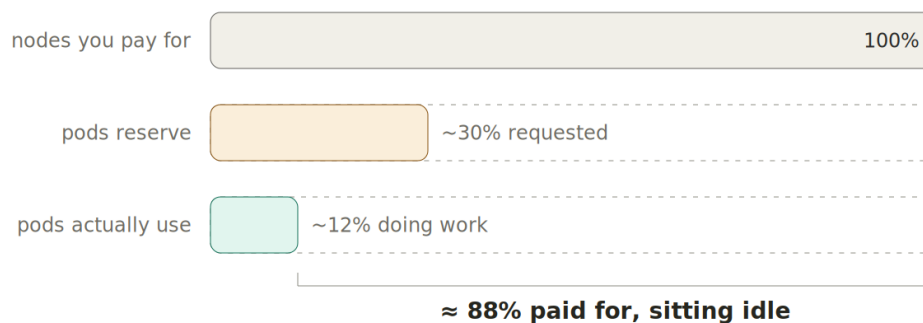
CHAPTER 03

Standardization is only standardization if it's the default

In one line a standard only counts if it's the default — make the right-sized, secure config the easiest path and most of the 75% waste fixes itself.

→ Standardization only counts when it's the default, and the 75% idle compute is what funds the work. It's the number that buys you the headcount and the migration window, not a technical curiosity. Here is what the platform actually ships.

The 15-cluster fleet is what a world of suggestions looks like. There was probably a “recommended” instance type, a “preferred” way to set resource requests, a “standard” monitoring setup. All optional, all therefore ignored under deadline, which is how you end up burning three-quarters of your compute: every team picked instance types out of fear and set requests by copying whatever the last service used, padded “to be safe.” Nobody was wrong individually. Collectively, it’s a fortune in idle compute. The mechanism is consistent wherever it’s measured — teams pad requests to avoid throttling and out-of-memory kills, that padding is invisible to whoever owns the bill, nothing ever revisits the numbers after deployment, and the autoscaler dutifully provisions nodes to match the inflated requests as if they were real demand. The gap becomes structural.



Across tens of thousands of clusters, CPU runs ~8-13% utilized before optimization (CAST AI).

The chart's 88% and this guide's three-quarters are the same gap measured two ways. 88% is provisioned capacity minus what pods are actually doing; roughly 70% is provisioned capacity minus what pods even reserve. Three-quarters is the conservative version, and it's the one to use in a room.

The fix is not a better recommendation. It’s making the right-sized configuration the one teams get without thinking about it.

Attack the over-provisioning at the default layer

Start with the headline number, because it funds the whole project. The 75% waste comes from two compounding habits: oversized nodes and oversized pod requests. Standardize both as defaults and the number moves on its own — no per-team negotiation required.

Nodes: let the platform right-size, and make consolidation non-optional. The teams over-provisioned nodes because static node groups punish you for guessing low and never punish you for guessing high. Karpenter inverts that — it provisions to fit actual pending pods and actively removes nodes it can consolidate. The key is the disruption policy: consolidation must be on by default, not something teams opt into.

```
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: default
spec:
  template:
    spec:
      requirements:
        - key: kubernetes.io/arch
          operator: In
          values: ["amd64", "arm64"] # allow Graviton — cheaper per unit of work
        - key: karpenter.sh/capacity-type
          operator: In
          values: ["spot", "on-demand"] # spot-first for fault-tolerant workloads
        - key: karpenter.k8s.aws/instance-category
          operator: In
          values: ["c", "m", "r"]
        - key: karpenter.k8s.aws/instance-generation
          operator: Gt
          values: ["5"]
      nodeClassRef:
        group: karpenter.k8s.aws
        kind: EC2NodeClass
        name: default
      disruption:
        consolidationPolicy: WhenEmptyOrUnderutilized # this line is the over-provisioning fix
        consolidateAfter: 15m # gentle by default — tighten per workload class, not fleet-wide
      limits:
        cpu: "1000" # ceiling so a runaway workload can't eat the account
```

The teams don't choose instance types anymore. They describe what their pods need; the platform selects the cheapest node that meets those requirements and removes it when the node is idle. That single `consolidationPolicy` line does more for the utilization number than any amount of “please right-size your clusters” ever did. The two requirements worth defending in a design review are the ones carrying real money: letting Karpenter pick **Spot** capacity for fault-tolerant workloads cuts compute cost on the order of 60% for partial Spot use and as much as ~77% for Spot-heavy fleets, and allowing **arm64 / Graviton** taps a price-performance pool that's now roughly 9% of cloud CPUs and growing several times faster than x86.

Both numbers come with a condition that belongs in the open, not the footnotes: Spot only pays off for workloads that tolerate interruption, and Graviton for workloads you can run on ARM. So those are savings on the *qualifying fraction* of the fleet, not the whole bill — and sizing that fraction honestly against your actual workloads is part of the cost case, not a

detail beneath it. One more default worth setting with care: `consolidateAfter` controls how aggressively Karpenter reclaims nodes. Fast consolidation maximizes savings but churns nodes, and that churn punishes stateful and latency-sensitive workloads — so it's a per-workload-class dial, not a fleet-wide one. Shipping aggressive consolidation as the universal default is exactly the kind of thing that sends a team back to its own cluster, which is the failure this whole guide is about.

Two honest caveats about consolidation itself, because the whole guide pushes toward it. First, the strategic one: not every one of the 15 clusters is a waste. Some of that fragmentation is deliberate and correct — blast-radius isolation, a hard compliance or regulatory boundary, a deliberately autonomous team — and consolidation trades those properties away for manageability. The trade isn't free: one platform is also one failure domain and one noisy-neighbor surface where fifteen clusters were fifteen smaller ones. “More manageable than the sprawl” is usually true and occasionally how you turn fifteen contained outages into one company-wide one. So consolidate what *should* be shared and keep a documented reason for every boundary you preserve. Second, the concrete one, because it's the same instinct expressed in config: the `NodePool` above sets no disruption budget, which means Karpenter falls back to its default of disrupting up to 10% of the pool's nodes at once. On a shared platform carrying many teams, that default is worth making explicit — a `disruption.budgets.blockCaps` how much of the pool can churn at once (it rate-limits voluntary disruption only, so expiring nodes aren't covered), which is exactly the failure-domain discipline the first caveat is asking for. The platform lets you collapse fifteen blast radii into one; the disruption budget is part of how you keep that one from becoming a single point of failure.

Pods: set sane request/limit defaults so teams stop guessing. Half the over-provisioning lives in pod requests that were copied and padded. A `LimitRange` gives every workload a sensible default request without the team specifying one, and a `ResourceQuota` caps what the namespace can consume so one team can't reproduce the old fleet inside the new platform.

```
apiVersion: v1
kind: LimitRange
metadata:
  name: defaults
  namespace: payments
spec:
  limits:
    - type: Container
      defaultRequest:      # what a container reserves if it doesn't say
        cpu: 100m
        memory: 128Mi
      default:            # its ceiling if it doesn't say
        cpu: 500m
        memory: 512Mi
  ---
```

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: tenant-ceiling
  namespace: payments
spec:
  hard:
    requests.cpu: "32"
    requests.memory: 64Gi
    limits.cpu: "64"
```

The two lines doing the work are `defaultRequest` and the `ResourceQuota` hard caps. The first means a team that specifies nothing still gets a sane, modest reservation instead of copy-pasting the last service's padded number — the single most common source of idle compute in the first place. The second means that no team can quietly rebuild the 15-cluster fleet within a single namespace. Neither requires a conversation, a review, or a doc anyone has to remember: right-sized is simply what you get, and oversized is now the thing you'd have to go out of your way to ask for. That is standardization doing its real job — not a recommendation that evaporates under deadline, but the path of least resistance.

This is also where you'll hit the objection that funds all over-provisioning: *we pad for safety; tighter requests mean more outages*. It's wrong, and it's worth being able to say why in the room. In practice, the padding doesn't buy reliability — clusters drowning in headroom still rack up out-of-memory kills because static padding targets the wrong workloads. Automated right-sizing tends to *reduce* OOM kills while cutting waste, not cause them, because the same mechanism that strips slack from over-provisioned workloads adds room to the genuinely starved ones humans miss. Efficiency and reliability aren't a trade here; the padding was hurting both. (The strongest version of this claim comes from optimization vendors with a product to sell, so test it on your own workloads before you bank on it — but the direction holds up: uniform human padding is a worse allocator than per-workload right-sizing.)

The unit of self-service is the tenant, not the cluster

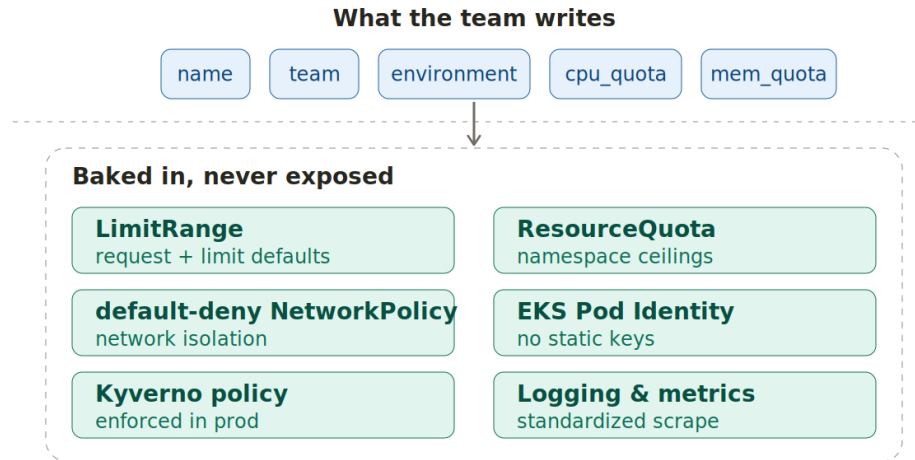
Here's the structural shift that makes consolidation real. In the old world, the unit a team owned was a *cluster* — which is why there were 15. In the new world, the unit a team owns is a namespace/tenant on the shared platform, and provisioning it must be genuinely self-service: a team gets a fully configured, guard-railed tenant without a ticket.

The way to make that safe is with a module with a deliberately small surface area. The team fills in what's theirs; everything that makes the tenant standard, secure, and right-sized is baked in and not exposed:

```
module "tenant" {
  source = "git::https://github.com/acme/platform-modules.git//tenant?ref=v2.3.0"
```

```
# — what the team chooses (the entire surface they see) —
name   = "payments"
team   = "payments-platform"
environment = "prod"
cpu_quota = "32"
mem_quota = "64Gi"

# — baked in by the module, not exposed as knobs —
# • LimitRange with sane request/limit defaults
# • ResourceQuota ceilings
# • default-deny NetworkPolicy + platform-required allows
# • EKS Pod Identity association wired to the team's IAM role
# • Kyverno policy binding (audit in non-prod, enforce in prod)
# • Pod Security Standards 'restricted' enforced via namespace label
# • standardized logging/metrics scrape config
}
```



Five fields above the line; everything that makes the tenant safe and right-sized baked in below it.

That tiny interface is the standardization. A team can't *accidentally* deploy without resource limits, without network isolation, or with static AWS keys, because the only path to a tenant is the path that includes all of it. The standard isn't documented; it's the default, and the default is the easiest thing to do. That's the whole chapter in one module.

FAILURE MODE TO WATCH FOR

Standards by exception. The moment you let teams "temporarily" bypass the module — hand-rolled namespaces, one-off Terraform, "just this once" quota overrides — you're back to suggestions, and you'll rebuild the 15-cluster sprawl one exception at a time inside a single cluster. Exceptions are sometimes necessary; they should be rare, visible, expiring, and owned, never the quiet default that the real standard erodes into.

CHAPTER 04

Security has to live on the paved road, or it gets bypassed

In one line security that lives off the paved road gets routed around — bake the baseline into the default path so the secure way is also the easy way.

→ *The risk you inherited is concrete: fifteen drifted IAM postures, static keys that never rotate, default-open networks. Consolidation is the one moment you get to fix all of it at once. Here is the baseline the platform bakes in.*

Embed the baseline; don't publish it

The consolidation is a rare opportunity to make the secure configuration the default — for all 15 teams at once — instead of asking each team to implement it. Four things belong in the tenant module from Chapter 3, baked in, not optional:

Workload identity with no static keys. The single biggest security win of consolidation is killing long-lived AWS credentials — the static access key sitting in a Kubernetes Secret that never rotates is the exact artifact that turns one leaked repo into a breach. EKS Pod Identity (or IRSA) removes the credential entirely: a workload assumes a scoped IAM role through its service account, short-lived and rotated automatically. The whole mechanism is one association, and the single line that carries the security is the role binding:

```
resource "aws_eks_pod_identity_association" "payments_api" {
  cluster_name = "platform-prod"
  namespace   = "payments"
  service_account = "payments-api"
  role_arn     = aws_iam_role.payments_api.arn # ← scoped to exactly what this app needs, nothing more
}
```

That `role_arn` is the paved-road argument in miniature: because the binding is least-privilege *by construction*, a team on the platform gets correct, scoped, auto-rotating credentials without thinking about it, while the engineer who wants a static key now has to go out of their way to do the less safe thing. That is the inversion the whole chapter turns on — the secure path becomes the path of least resistance, so it's the one that survives a deadline. Wire it into the tenant module, multiply it across fifteen migrated teams, and the consolidation has done what no security memo ever did: made *no static keys* true by default, rather than aspirational.

Default-deny networking. In the old clusters, everything could probably talk to everything, because someone would have had to write policies to stop it and nobody had time. Flip the default: a new tenant starts closed — in *both* directions — and the module ships the handful of allows that keep a closed tenant usable.

```

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
  namespace: payments
spec:
  podSelector: {}          # every pod in the namespace
  policyTypes: ["Ingress", "Egress"] # closed in BOTH directions by default
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-dns          # the one allow every team forgets — shipped WITH the deny
  namespace: payments
spec:
  podSelector: {}
  policyTypes: ["Egress"]
  egress:
    - to:
      - namespaceSelector: { matchLabels: { kubernetes.io/metadata.name: kube-system }}
      podSelector: { matchLabels: { k8s-app: kube-dns }}
    ports:
      - { protocol: UDP, port: 53 }
      - { protocol: TCP, port: 53 } # ← without this, every pod silently fails to resolve names

```

The load-bearing pair is default-deny in *both* directions, with the `allow-dns` rule shipped alongside it. Default-deny ingress is the easy half; closing egress is where teams hurt themselves, because a namespace that can't reach `kube-dns` can't resolve a single name — pods start, then mysteriously fail to talk to anything, and someone burns an afternoon discovering it was DNS all along. That footgun is exactly what the paved road is for: the platform absorbs it once, in the module, so no team meets it fifteen separate times. On the road, the secure topology — closed both ways, DNS already allowed — is the one you inherit on day one; an engineer who wants the flat, open network of the old world has to deliberately punch holes to get it. Same move as the credentials above: the safe default is the effortless one, which is the only reason it survives contact with a deadline.

A hardened pod baseline from a single label. This is the purest expression of the whole thesis, so it's worth stating plainly: most of “no root, no privilege escalation, no host namespaces” isn't something you need a policy engine to author at all. Kubernetes ships it built-in as the Pod Security Standards `restricted` profile, turned on per namespace with one label — `pod-security.kubernetes.io/enforce: restricted`. It matters specifically on EKS because EKS ships every namespace with the wide-open privileged profile by default, and managed clusters can't change that fleet-wide; the namespace label is the opt-in to hardening. Bake the label into the tenant module, and a fully hardened security posture arrives for free with every tenant—zero policy written, the secure default inherited rather than authored. No cleaner example of “make the right way the easy way” in this entire guide than security-by-default as a single line of namespace metadata.

Admission control for what the baseline can't express. The `restricted` profile is broad but fixed; it can't encode *your* rules. That's the job of a policy engine (Kyverno or OPA Gatekeeper): "you can only pull from our registry," "no `:latest` tags," "every workload declares resource requests," "these labels are mandatory." The division of labor is the point — Pod Security Standards handle the standard hardening for free, the policy engine handles the bespoke, and you reach for the heavier tool only where the free one can't reach. Crucially — and this is the bridge to Chapter 6 — those custom policies start in *audit* mode so you can see what would break before anything does.

Security monitoring is for the road you actually built

Security monitoring is usually presented as a tour of IDPS, SIEM, and vulnerability scanners: Snort, Splunk, Nessus, the usual cast. The tools are fine. The framing is backward. Detection tooling is only as good as your knowledge of what *should* be running, and the 15-cluster world had no such baseline — every cluster was a little different, so "anomalous" was undefined. Consolidation is what makes monitoring meaningful: once there's one paved road, a workload doing something off-pattern is *visibly* off-pattern. The platform doesn't just standardize deployment; it standardizes normal, which is the precondition for detecting abnormal.

So: pick your detection tools to match the consolidated reality, not the sprawl. You need far less sprawling SIEM correlation when the fleet is uniform, and far more value from admission-time prevention (stopping the bad config before it deploys) than from runtime detection (catching it after). Prevention on the paved road is cheaper than detection across chaos.

FAILURE MODE TO WATCH FOR

Security that's only on the new platform while the old clusters still run production. During a multi-team migration, your strongest controls cover the workloads that already moved — i.e., the ones least likely to hurt you — while the legacy clusters with the drifted RBAC and the static keys keep serving real traffic. Sequence the security baseline against the *migration*, not the calendar, and treat any cluster still outside the paved road as your highest-risk surface until it's decommissioned, not after.

CHAPTER 05

Instrument the platform like a product, not like infrastructure

In one line instrument adoption, not just infrastructure — a green dashboard can't tell an adopted platform from an ignored one.

The question isn't "how do we collect metrics" — every one of the 15 clusters already collected metrics, in 15 incompatible ways. The question is which metrics tell you whether the platform is winning. The answer follows directly from Chapter 1: if a platform is a product, you instrument it like one.

That means two distinct measurement layers. Platform teams almost always build only the first.

Layer one: is the platform healthy? (necessary, insufficient)

This is the infrastructure telemetry you already know how to build — CPU, memory, latency, error rates, the utilization curve. One slice of it matters more than the rest for this project: the utilization number is your proof of value. Three-quarters idle compute was the business case; fleet utilization climbing out of single digits as teams migrate is evidence the case was real. Put that single number where leadership can see it — it's the one that funds everything else.

Alongside it: the four DORA metrics — deployment frequency, change lead time, change failure rate, and time to restore service. They're the right scoreboard for "is the platform making delivery better," and the State-of-DevOps research consistently ties strong platform engineering to better scores on them.

Notice the limit both share: neither DORA metrics nor raw infrastructure telemetry can distinguish a healthy, adopted platform from a healthy, ignored one. Both can have green dashboards. The console had a green dashboard too.

Layer two: is the platform being adopted? (the layer nobody builds)

This is product analytics for your internal product, and it's where the real decisions come from:

Adoption and migration progress — measured by usage, not cutover.

Not "12 of 15 clusters migrated" but what fraction of production traffic, deploys, and new services actually originate on the platform. A team can be marked "migrated" and still run their real work on the old cluster. Traffic doesn't lie; project trackers do.

Time to first success, tracked over time.

Instrument the golden path itself: how long from a new tenant request to serving traffic? If that number creeps up release over release, your platform is getting harder to adopt, and you'll feel it as stalled migrations months later. This is a leading indicator; migration stalls are the lagging one.

Friction and escape-hatch usage.

Where do engineers abandon the paved road? Every support ticket, every "how do I..." in Slack, every time someone drops to raw `kubectl` or hand-rolled Terraform is a friction signal. A spike in escape-hatch usage for one workload type isn't a support problem — it's a product gap telling you exactly what to build next.

The teams who haven't moved, and why.

Track the holdouts as a cohort. They're your roadmap. The team that won't migrate is your most valuable bug report — but only if you're actually measuring why.

Self-service ratio.

The share of requests — new tenant, new environment, credential rotation — completed without a human on your team in the loop. It's the cleanest single proxy for whether you've built a platform or a managed service (Chapter 1), and it indicates whether the platform will scale beyond your team's headcount.

The tools don't change much between the layers; Prometheus and Grafana serve both. What changes is what you choose to put on the dashboard the platform team looks at every morning. If that dashboard is all CPU and no adoption, you've instrumented infrastructure and called it a platform.

FAILURE MODE TO WATCH FOR

Vanity metrics that only go up. "Services onboarded," "clusters consolidated," "policies enforced" — these never go down, so they always look like progress, which makes them comfortable and nearly useless. The honest metrics are the ones that can deliver bad news: flat adoption, rising time-to-success, an escape hatch that's becoming more popular than the path it bypasses. If none of your dashboards can tell you the platform is losing, you're not measuring the platform.

CHAPTER 06

Guardrails, not gates

In one line guardrails let everyone move and stop only the harmful action; gates stop everyone and get routed around — so automate, and reserve human gates for genuine blast radius.

→ A gate stops everyone to catch the few who'd do harm. A guardrail lets everyone move and stops only the harmful action. Gates are human-in-the-loop, so they scale with your team's attention, which is to say they don't scale, which is to say they get routed around. The test for a real gate is blast radius, not habit: deleting a production tenant, widening an IAM role, changing a network boundary, touching shared platform infrastructure. Routine deploys aren't on that list. If you're gating something that happens fifty times a day, you've built a bottleneck and labeled it governance. This gets more urgent as AI-generated code and config raise the volume of change hitting your controls, because human gates are the first thing to buckle under that load. The implementation is one pattern: audit, then enforce.

Most governance should be automated, audited first, then enforced

Almost everything you want to govern — resource requests, image provenance, network posture, tagging for cost allocation — is a property you can check automatically at admission time. That's a guardrail, and the safe way to introduce one is the same way you'd ship any product change: observe before you enforce.

```

apiVersion: kyverno.io/v1
kind: ClusterPolicy      # readable form shown; Kyverno now recommends CEL-based ValidatingPolicy
metadata:
  name: require-resource-requests
spec:
  background: true
  rules:
    - name: check-requests
      match:
        any:
          - resources:
              kinds: ["Pod"]
      validate:
        failureAction: Audit    # ← start HERE. watch what would break. Then flip to Enforce.
        message: "Every container must declare cpu and memory requests."
        pattern:
          spec:
            containers:
              - resources:
                  requests:
                    cpu: "?*"
                    memory: "?*"

```

Audit mode is the whole discipline in one field. You turn the policy on; it blocks nothing and shows you every workload across the newly consolidated fleet that would violate it. You fix the violations (or discover the policy was wrong), then flip to Enforce. This is how you roll governance across 15 teams' worth of inherited workloads without breaking production on day one — and it's why governance and migration have to be sequenced together, not bolted on at the end.

FAILURE MODE TO WATCH FOR

Governance that optimizes for control instead of for the paved road staying the easy path. The tell is simple — when a guardrail starts pushing engineers back toward the workarounds you spent five chapters eliminating, the guardrail is the problem, not the engineers. Every gate you add is a small tax on adoption; charge it only where the blast radius justifies it, and audit your own governance the same way you audit everything else: is the right way still the easy way?

CLOSING

The hard part isn't the architecture

Everything in this guide is, in a sense, simple. Karpenter consolidates nodes. Pod Identity kills static keys. Kyverno enforces policy. A Terraform module makes a tenant. None of it is exotic. If platform engineering were an architecture problem, the 15-cluster fleet would have been consolidated years ago. The hard part is the operating discipline — keeping the paved road faster than the workarounds, every day, while the pressure to add one more gate and support one more exception never stops.

→ For the buyer's questions — how long this takes to learn, what a failed first attempt costs, and the cost math — see the leadership edition's close.

APPENDIX

Grounding & sources

The load-bearing claims in this guide are drawn from primary or well-established industry sources.

- **Platform-team adoption trend.** “By 2026, 80% of large software engineering organizations will establish platform engineering teams ... up from 45% in 2022.” Gartner, *Top Strategic Technology Trends for 2024* (Willemsen, Olliffe, Chandrasekaran), 16 October 2023.
- **Kubernetes utilization / over-provisioning.** Two sources, deliberately. (1) *Vendor benchmark* — CAST AI *Kubernetes Cost Benchmark* (2024, 2025) and *State of Kubernetes Optimization Report* (2026): average pre-optimization CPU utilization ~8–13%, memory ~20% across tens of thousands of clusters. CAST AI sells the optimization product these figures justify, so treat the exact numbers as directional. (2) *Independent corroboration* — Datadog *State of Containers and Serverless* (6 November 2025), drawn from thousands of companies: most workloads use under 25% of requested CPU and under half of requested memory. Note this is measured against *requests*, not *provisioned capacity* — a different, more conservative denominator pointing the same direction. The *direction* (most clusters waste most of their compute) is consistent across both; the precise percentage is not load-bearing, and the only figure that matters for a given reader is the one measured against their own fleet.
- **Spot / Graviton economics.** ~60% compute savings for partial Spot use and ~77% for Spot-heavy fleets; ARM / Graviton at ~9% of cloud CPUs and growing several times faster than x86: CAST AI reports, as above.
- **Paved road / golden path.** Netflix “paved road” and Spotify “golden path” (term via Frank Herbert), per-discipline paths starting with backend microservices, and the security-baked-in platform (Netflix Wall-E): Spotify Engineering and Netflix engineering writing; Red Hat and Team Topologies summaries.
- **Platform as a product / cognitive load / thinnest viable platform.** Team Topologies (Skelton & Pais); Thoughtworks Technology Radar (product management for internal platforms, 2017).
- **“Secure option is the fastest.”** Gartner guidance that architectural and security controls belong in the platform rather than left to voluntary adoption.
- **Delivery metrics.** DORA four keys (deployment frequency, lead time, change failure rate, time to restore) and the State of DevOps correlation between platform engineering and organizational performance.

- **Adoption-rate contrast (~80% vs ~20%).** Practitioner-reported, not a controlled study — frame as field observation, not a benchmark, if cited externally.
- **AI amplifies the platform (strategic claim).** *2025 DORA Report: State of AI-assisted Software Development* (Google/DORA, ~5,000 respondents) — AI magnifies existing strengths and weaknesses; internal-platform quality is the strongest determinant of whether AI delivers organizational value or accelerates dysfunction. Independent, survey-based; the load-bearing source for the AI thread.
- **AI volume buries human gates (mechanism).** *Faros AI AI Engineering Report 2026: The Acceleration Whiplash* — telemetry across ~22,000 developers; under high AI adoption, incidents per PR rose ~242% and ~31% more PRs merged with no review. **Vendor source**, explicitly correlation across independent cross-sections, not causal — cite as directional and flagged. The report also finds that maturity offered no exemption — its high-performing, high-DORA organizations showed the same downstream deterioration as everyone else — which the body now reflects rather than softens.