PLATFORM FIX

Co-authored by Neil Cresswell, Portainer
and Steven Wade, Platform Fix

EBOOK

WHAT KUBERNETES REALLY TAKES: TIME, TEAMS, AND EIGHT SEPARATE PROJECTS

EVERY KUBERNETES PROJECT STARTS WITH A LIE

"We'll containerize our applications and modernize delivery."

Six months later, you're drowning in eight parallel workstreams, burning cash, and your best engineers are updating their LinkedIn profiles, whilst simultaneously complaining on Reddit about how the project they are working on is going sideways.

The pitch sounded simple. Strategic, even. A technical evolution from traditional infrastructure toward containerization and cloud-native operations. One project. One transformation. One path to operational excellence.

But here's what nobody tells you: Kubernetes isn't a project. It's eight projects wearing a trench coat, pretending to be one initiative.

THE REAL COST OF ILLUSION

When enterprises discover this truth, it's usually too late.

The damage shows up in:

- **Budget overruns of 300-400%** (that $500K project becomes $2M)
- **Timeline explosions** (6-month initiatives stretch to 18 months)
- **Talent exodus** (your best engineers leave for companies that "get it")
- **Technical debt** (half-finished implementations that work... sort of)

This isn't a technology problem. It's a planning problem. And it affects everyone - whether you're building custom software, deploying COTS applications, or working with outsourced development teams.

The Eight Hidden Projects

What organizations actually sign up for when they adopt Kubernetes:

1. **Application Refactoring**
   Making legacy apps container-ready
2. **Container Supply Chain**
   Standardizing builds, scanning, and registries
3. **Platform Operations**
   Running Kubernetes itself
4. **CI/CD Transformation**
   Rebuilding entire delivery pipelines
5. **Zero Trust Networking**
   Securing container-to-container communication
6. **Cloud-Native Observability**
   Monitoring ephemeral infrastructure
7. **Everything-as-Code**
   Adopting GitOps and declarative infrastructure
8. **Platform Engineering**
   Making it actually usable for humans

Each of these is substantial enough to be a significant initiative. Together, they represent a fundamental operating model change that touches every corner of IT.

THE ILLUSION OF "ONE PROJECT"

In practice, Kubernetes adoption unfolds as a bundle of interdependent workstreams; each large, each complex, and each demanding time, tooling, budget, and expertise. The reason many Kubernetes rollouts stall, overrun, or deliver underwhelming outcomes is not because the technology is flawed. It is because organizations treat a multi-track transformation as a single-track implementation.

And this challenge is not limited to companies that write their own software. The same complexity appears when deploying commercial off-the-shelf (COTS) software or using outsourced development teams. Kubernetes changes how software is delivered, how it is run, how it is secured, and how it is observed, regardless of who wrote the code.

What is actually being introduced is not just a container orchestrator. It is an operating model change. One that redefines infrastructure provisioning, application packaging, deployment pipelines, runtime security, networking, observability, and developer experience.

Too often, these layers are treated as post-deployment details or left to be figured out ad hoc. The result is fragile workloads, unscalable platforms, poor developer uptake, and platform teams buried in support tickets.

The real challenge is not the Kubernetes cluster. It is everything that surrounds it. That includes adapting legacy software, handling vendor applications not built for containers, reskilling operations teams, enforcing new security boundaries, and providing a usable interface for developers or partners.

This ebook breaks down each of those layers and outlines what is truly involved, so you can plan, budget, and staff for what you are actually taking on.

BECAUSE KUBERNETES DOES NOT JUST CHANGE WHERE SOFTWARE RUNS

It changes how your entire IT organization operates.

REFACTORING LEGACY APPS, OR ADAPTING WHAT YOU CAN'T CHANGE

Before you ever touch Kubernetes, the first challenge is deciding what you plan to run on it. Most enterprise applications (whether custom-built, vendor-supplied, or inherited from an outsourcing partner) are not immediately compatible with containerized environments.

For in-house systems, this typically means reworking traditional architectures. Monoliths with shared memory models, hardcoded IP addresses, local file system dependencies, or tightly bound config files do not lift into containers cleanly. Apps may require restructuring into smaller services, externalizing state to managed storage, swapping environment-specific logic for runtime config, or reworking how services discover each other.

Even basic issues like signal handling for graceful shutdown, health probes for liveness and readiness, or resource constraints like JVM memory versus pod limits can require significant code and config changes.

COTS software and third-party-developed apps come with their own problems. Many are not packaged as containers. Those that are might lack proper Helm charts or Kustomize manifests. Even when vendor charts exist, they often assume full-cluster privileges, unscoped secrets, or outdated ingress models. You may need to rewrite manifests, isolate workloads, or negotiate changes the vendor is not ready to support.

When you rely on outsourcing partners, runtime assumptions and delivery formats vary. You will need to establish SLAs that specify Kubernetes compatibility. Code must be containerized. Manifests must follow your policies. Delivery must include observability hooks.

This is not just replatforming. It is reengineering. It often happens before Kubernetes is even deployed.

Treat this phase as a dedicated project. Assign technical leads. Allocate testing environments. Assess every app against a readiness matrix.

ONE OF THE MOST COMMON CAUSES OF KUBERNETES FAILURE IS ASSUMING APPLICATIONS ARE READY FOR IT WHEN THEY ARE NOT.

STANDARDIZING YOUR CONTAINER BUILD AND PACKAGING STRATEGY

Once you have container-ready applications, the next challenge is packaging them consistently and securely. This is not a developer detail or a tweak to a Jenkins file. It is a foundational control layer. Without standardization, you get drift, risk, and operational chaos.

Start with base images. Will you allow teams to choose Alpine, Ubuntu, Red Hat UBI, or distroless? Or will you define pre-approved images curated by the platform team? This choice affects security, patching, and compliance.

Build automation is next. Do you use Docker directly, or tools like BuildKit or kaniko for safer builds in CI? Will you enforce structure with templated GitHub Actions, Tekton pipelines, or shared Jenkins libraries? You'll need Dockerfile linting (Hadolint), image signing (cosign), and SBOM generation (Syft or Grype).

Scanning is non-negotiable. Use Trivy, Grype, or commercial platforms like Prisma Cloud to catch known CVEs before runtime. Decide whether to block on critical vulnerabilities, and who owns fixing them.

Images need to be stored. Will you use AWS ECR, GCP Artifact Registry, or self-hosted Harbor? What are your tagging strategies? Are images immutable, and who can delete or promote them?

Secrets must be injected securely. You can start with Kubernetes Secrets, but most move to Sealed Secrets, SOPS with GPG or KMS, External Secrets Operator, or Vault. These tools add encryption, access control, and syncing with Git.

COTS apps often come with insecure image defaults. Outdated base images, hardcoded creds, or unauthenticated registries. These need review. Outsourced teams must follow your container build policy. Contracts should include base image requirements, CVE thresholds, and manifest delivery.

THIS IS YOUR CONTAINER SUPPLY CHAIN. IT NEEDS GOVERNANCE.

DESIGNING AND OPERATING YOUR KUBERNETES RUNTIME PLATFORM

Standing up Kubernetes is quick. Running it properly is not.

Whether you use EKS, AKS, GKE, OpenShift, or something self-hosted like kubeadm, Talos, or Rancher, you are responsible for the platform it becomes.

Start with tenancy. Do you run one large cluster with isolated namespaces, or deploy multiple clusters for separation by team, environment, or region? Will dev, test, and prod live together?

Authentication and access must be integrated with enterprise identity. Most use SSO via Azure AD, Okta, or LDAP. RBAC policies define what users and service accounts can do. OPA Gatekeeper or Kyverno layer in policy enforcement, preventing risky workloads or privilege escalation.

Networking requires a CNI plugin such as Calico, Cilium, or AWS VPC CNI. You'll define ingress through NGINX, Traefik, or Istio Gateway. Internal DNS and TLS termination must be handled cleanly. Traffic control, service discovery, and egress controls follow.

Storage uses CSI drivers. You'll provision from cloud services (EBS, GCE PD, Azure Disks) or use on-prem platforms like NetApp, Ceph, or Portworx. Map storage classes to performance tiers. Define snapshotting and backup policies early.

Operations need processes. Who handles upgrades? What is the cadence? Do you enable auto-repair? What's the recovery plan if a control plane fails?

COTS apps may break cluster policies. Some require elevated privileges or host-level access. These need sandboxing or isolation.

YOUR CLUSTER IS NOT A TOOL. IT IS A PRODUCT. TREAT IT AS SUCH.

REBUILDING CI/CD PIPELINES FOR KUBERNETES AND GITOPS

Kubernetes breaks the assumptions behind most existing CI/CD pipelines. Traditional flows expect to deploy binaries to virtual machines, not container images to orchestrated clusters. That means your pipelines need a full redesign.

Start with image building. CI jobs must now build Docker or OCI images from source, tag them consistently, scan them, and push them to a private registry. Tools like GitHub Actions, GitLab CI, Jenkins, or CircleCI must be reconfigured. Add linting, SBOM generation (Syft), vulnerability scanning (Trivy), and optional image signing (cosign).

Next comes deployment. In the Kubernetes world, deployment should be declarative. Helm, Kustomize, or plain manifests define state. That state belongs in Git. GitOps tools like Argo CD or Flux then pull those definitions and sync them to the cluster.

This shift from push-based CD to pull-based GitOps introduces new concerns. Repo structure becomes critical. You need to separate application code from deployment logic. Environments must be represented as branches, directories, or entire repositories. Secrets must be managed via SOPS, Sealed Secrets, or External Secrets Operator.

Promotion, rollback, and approval workflows also change. Developers no longer run kubectl or SSH into servers. They submit merge requests to promote to staging or production. Observability must be wired into this flow so teams can validate deployment outcomes.

COTS vendors may not provide GitOps-compatible manifests. Some offer Helm charts. Others ship only Docker images. You may need to wrap those artifacts into your standard templates and host your own deployment configs.

Outsourced development contracts should now include delivery of container images, deployment manifests, and GitOps compatibility. If the external team cannot deliver into your pipeline, you'll end up manually bridging the gap.

CI/CD IS THE DELIVERY BACKBONE.

If it doesn't evolve, everything else collapses under manual work and inconsistency.

SECURING INTERNAL COMMUNICATIONS WITH ZERO TRUST NETWORKING

By default, Kubernetes is flat. Every pod can talk to every other pod. That is convenient, but insecure. If a single pod is compromised, lateral movement is easy.

Zero trust changes that. It assumes no pod or service is trusted. Every connection must be explicitly allowed, authenticated, and monitored.

Start with NetworkPolicies. These define which pods can talk to which, using labels and namespaces. If your CNI supports it, enforce isolation at every layer. Plugins like Calico and Cilium are widely used for this.

Then add identity. Service-to-service authentication requires a mesh like Istio, Linkerd, or Consul Connect. These tools inject sidecars or proxies into pods and enable mutual TLS, workload identity, and telemetry.

Next, look at ingress. Gateways must terminate TLS securely and block unauthorized access. Tools like cert-manager help manage certificates. Ingress controllers such as NGINX or Traefik should be configured to enforce rate limiting, IP filtering, and proper header propagation.

You also need to monitor east-west traffic. Use eBPF-powered tools like Cilium Hubble, Tetragon, or commercial systems such as Tigera, Isovalent, or Datadog.

COTS apps can be problematic. Many assume open internal networks and cannot handle sidecars or TLS. You may need to isolate them in permissive namespaces or run them behind reverse proxies that add security externally.

Outsourced teams may not know how to test their services behind network policies or inside a mesh. Your onboarding process must simulate production topology early and test conformance.

ZERO TRUST IS NOT JUST A SECURITY GOAL.

It is an operational principle. It forces clarity on how systems communicate and what is allowed.

REDEFINING OBSERVABILITY FOR EPHEMERAL INFRASTRUCTURE

In Kubernetes, everything is ephemeral. Pods restart. IPs change. Nodes come and go. Traditional monitoring systems struggle to keep up.

To regain visibility, you need container-native observability. That means rethinking metrics, logs, and traces.

Start with metrics. Prometheus is the standard. Use kube-state-metrics to expose cluster state. Use node-exporter for host stats. Instrument apps with OpenMetrics and label everything by namespace, pod, and service. Store long-term metrics in Thanos or Cortex if needed.

For logs, Fluent Bit and Fluentd are common collection agents. Aggregate logs into Elasticsearch, Loki, or Splunk. Logs should include pod metadata and be structured consistently across teams and services.

Tracing is the most underutilized pillar. Distributed tracing, via OpenTelemetry and backends like Jaeger or Tempo, allows you to follow a single request across services. This is essential in a microservices architecture.

Grafana remains the go-to dashboarding tool, but you will need to build and maintain your own dashboards for developers, SREs, and auditors. Alerting should be tied to service-level indicators and routed into Slack, PagerDuty, or Opsgenie.

COTS applications rarely emit usable telemetry by default. You may need to wrap them with sidecars, parse stdout logs, or monitor them externally with synthetic probes.

Outsourced teams must be required to expose metrics and logs as part of their deliverables. Otherwise, you'll have code running in production with no way to understand what it is doing.

OBSERVABILITY IS NOT ABOUT TOOLS.

It is about the discipline of understanding your systems. In Kubernetes, that discipline must be baked in from day one.

ADOPTING EVERYTHING-AS-CODE FOR CONTROL, CONSISTENCY, AND SCALE

Manual changes are the enemy of reliability. In Kubernetes, the only safe path is to declare everything as code.

Start with infrastructure. Use Terraform or Pulumi to provision clusters, networks, load balancers, and cloud resources. Store state securely, enforce change reviews, and track drift.

Next comes application configuration. Helm and Kustomize are common, but others use Jsonnet or CDK8s. Templates must support overrides for dev, test, and prod, and enforce linting and schema validation.

Secrets must also be declared, though encrypted. Tools like SOPS, Sealed Secrets, or External Secrets Operator ensure secrets can live in Git, yet remain unreadable without access to private keys or cloud KMS.

Policy-as-code closes the loop. OPA Gatekeeper or Kyverno lets you define and enforce rules: no privileged containers, mandatory resource limits, approved registries only.

Then comes GitOps. Argo CD and Flux continuously reconcile Git state into the cluster. Git becomes your source of truth. You no longer push to production—you commit to it.

COTS software complicates this. You may need to wrap vendor deployments into your templating system. If a vendor only provides a YAML blob, break it into templated, parameterized components.

Outsourced teams must adopt your as-code approach. Contracts should require all deployments, secrets, and infra changes to be delivered via Git, in formats you support.

EVERYTHING-AS-CODE ENABLES ROLLBACK, AUDITING, AND TEAM COLLABORATION.

Without it, Kubernetes becomes fragile. With it, you gain control.

PLATFORM ENGINEERING MAKES IT USABLE

Once Kubernetes is live, a new problem emerges. No one knows how to use it.

Developers struggle to write manifests. COTS vendors deliver misaligned charts. External teams ping your SREs for basic help. The system works, but only if you have specialists hand-holding every deployment.

Platform engineering solves this.

Your platform team builds golden templates, reusable scaffolds, and paved paths. They provide internal CLIs, GitHub Actions, or portals like Backstage or Portainer. They define what “good” looks like and make it easy to follow.

This team also curates documentation, integrates with secrets management, standardizes ingress patterns, and owns the developer experience end to end.

Without this, Kubernetes becomes a high-friction environment. Teams build workarounds. Inconsistency spreads. Reliability drops. The burden shifts back to the platform team, who now juggle tickets instead of building.

COTS and external teams benefit from platform engineering too. If you offer clear examples, standard templates, and Git integration, adoption is smoother. If not, you get fragmented delivery, broken deployments, and escalating support needs.

PLATFORM ENGINEERING IS NOT OPTIONAL.

It is the difference between adoption and entropy.

IT'S NOT ONE PROJECT, SO DON'T PLAN IT LIKE ONE

If there is one takeaway from everything you've read so far, it's this: Kubernetes is not a tool you roll out. It is a transformation you lead.

It will touch how your applications are structured, how your teams work, how infrastructure is provisioned, how software is delivered, how risk is controlled, and how operations are scaled. Whether you develop everything in-house, buy software off the shelf, or work with outsourced partners, Kubernetes will force a re-evaluation of assumptions baked into your stack and your org chart.

You are not just adopting containers. You are introducing eight major initiatives:

1. Refactoring legacy applications or adapting COTS
2. Standardizing container builds and registries
3. Designing and operating the Kubernetes platform
4. Rebuilding CI/CD and enabling GitOps
5. Implementing zero trust networking
6. Building container-native observability
7. Adopting everything-as-code and Git as truth
8. Enabling developers through platform engineering

Each of these is substantial. Each needs ownership, resourcing, and sequencing. The reason so many Kubernetes initiatives miss the mark is not a technology failure; it's a planning failure. When something this impactful is managed like a feature upgrade, the outcome is predictable: budget overruns, platform debt, team frustration, and ultimately, stalled adoption.

THE UNCOMFORTABLE TRUTH ABOUT WHAT YOU'RE ACTUALLY DOING

After walking through these eight projects, let's be honest about what's really happening. You're not adopting a container platform. You're transforming your entire operating model. The technology is the easy part - it's the organizational change that breaks most enterprises.

When I sit down with IT leaders twelve months into their Kubernetes journey, they all say the same thing: "We thought we were updating our infrastructure. We didn't realize we were changing everything about how we work."

Take your initial estimates and multiply them. Budget by 3-4x. Timeline by 2-3x. Team size by 2-5x. Ongoing operations by 5-10x. That reasonable $500K, six-month project becomes $2M and eighteen months. If things go well.

THE PATH THAT ACTUALLY WORKS

Here's what successful organizations do differently: they embrace a simple principle - slow is smooth, smooth is fast.

They don't try to tackle all eight projects at once. They pick one, nail it, then move to the next. Start with containerizing a single non-critical application. Get it running reliably. Learn what breaks. Fix it. Document it. Only then move to the next app.

Once you have containerized apps, build your supply chain. Once that's smooth, stand up a simple cluster. Each project becomes the foundation for the next. Each success builds confidence and knowledge.

This feels slow. Your steering committee will push for parallel workstreams to "accelerate transformation." Don't listen. The organizations trying to do everything at once are the ones in crisis twelve months later, with nothing working properly.

Sequential progress beats parallel chaos every time. You can show concrete wins: a successfully containerized app, a working security pipeline, a stable production cluster. These victories buy you political capital for the next phase.

WHY THIS MATTERS

Kubernetes delivers on its promises - infrastructure portability, self-healing applications, instant scaling, and eventual developer productivity. But only if you respect its complexity.

Start with one project. Get it right. Learn from it. Then expand. This isn't giving up on transformation - it's being smart about achieving it.

Six months from now, you want to be the organization with two projects completed successfully, not the one trying to salvage eight failing initiatives.

REMEMBER:

slow is smooth, smooth is fast. The choice is yours.

AFTERWORD

Here at Portainer, we have spoken to thousands of customers struggling to undertake a Kubernetes transformation, and all reaching out for help.

Whether it's cognitive overload, staff burnout/frustration, team members resigning in droves, or IT leadership getting fed up with spiraling costs or constant go-live delays. This all boils down to a lack of foresight, a lack of recognition to the organizational change required. Most commonly, however, is "conference driven platform design" leading to unnecessary complexity and dozens of tools introduced far too early simply "because its what everyone else uses". Resist this at all costs.

We hope this insight helps you guide a successful transformation project, because after all, a project failure reflects badly on the technology as a whole, and we don't want Kubernetes blamed for no good reason.

PLATFORM FIX

Co-authored by Neil Cresswell, Portainer
and Steven Wade, Platform Fix

PORTAINER.iO