WHITEPAPER

MIND THE GAP

Common Pitfalls When Modernising Legacy Applications onto Kubernetes

FOREWORD

There's a quiet myth that's been allowed to spread unchecked across boardrooms and architecture whiteboards alike, that Kubernetes is the moderniser's silver bullet. That if you can just get your legacy application onto Kubernetes, much of your transformation work is already done. But this isn't just wrong; it's dangerous.

Over the past few years, I've watched as well-meaning teams have treated Kubernetes not as a platform to grow into, but as a proving ground where legacy apps are flung en masse in the hope they'll land somewhere operational. They rarely do. And when they don't, the post-mortems always sound the same: it took longer than expected, cost more than budgeted, and we still don't quite know why the app fails under load or doesn't scale predictably.

Modernisation isn't about containers, clusters, or GitOps pipelines. It's about control. Control over how your applications behave. Control over how failures are managed. Control over how change is introduced without breaking what's already working. Kubernetes can give you that control, but only if you earn it first.

This booklet was born from hard truths, gathered from projects that stumbled, realigned, and slowly got back on track. It's a guide for those willing to take a slower, smarter path toward modernisation. One that respects the complexity of legacy systems, understands the trade-offs of hybrid deployments, and recognises that reliable operations always come before elegant architecture.

If you're expecting a best-practice checklist, you won't find it here. But if you're looking for a reality check; something to challenge the instinct to automate before you understand, or to design for a future you haven't proven is reachable; then you're exactly where you need to be.

Modernisation is possible. But it must be earned, one validated step at a time

The Portainer Team

CONTENTS

THE PROMISE AND THE PERIL

The idea of modernising a legacy application is compelling. It's the kind of initiative that gets heads nodding in board meetings and makes transformation teams feel like they're building the future. And when Kubernetes enters the conversation, the buzz intensifies. After all, Kubernetes represents modern infrastructure done right; container orchestration, scalability, portability, automation. What's not to like?

But here's the catch: Kubernetes doesn't make your legacy application modern. It makes the environment modern. And if your application isn't ready to live in that world, all Kubernetes does is expose every brittle edge, every hardcoded assumption, every tightly coupled dependency your app has been hiding for years. Suddenly, what was once a "working system" becomes a collection of architectural liabilities, surfaced and magnified by the realities of running in a distributed, dynamic platform.

This is where so many modernisation projects falter. They assume that containerisation is the hard part, and that once the app is inside a container, the rest is just deployment scripting. They forget that the way an application behaves in a traditional, long-lived VM or bare metal server is fundamentally different to how it behaves in a volatile, container-based runtime. When things start breaking, they blame Kubernetes, or the CI/CD tooling, or the infrastructure team. Rarely do they realise the problem started with a faulty assumption: that you can modernise simply by shifting platforms.

Modernisation is not about tooling. It's not about Kubernetes, GitOps, Helm, or Istio. It's about understanding. Understanding your application's actual behaviour. Understanding how it interacts with its environment. Understanding what will break when you move it. And only then (only when you've validated those interactions) do tools like Kubernetes start to pay dividends.

This ebook is a field guide to the real pitfalls you'll face along that journey. Not the textbook kind, but the kind that turn real projects into delayed, over-budget cautionary tales. If you're serious about making Kubernetes work for legacy applications, you need to understand not just where you're going, but how to get there without falling into the traps that lie between.

TOO FAR, TOO FAST: THE LEAP THAT BREAKS EVERYTHING

There's a natural enthusiasm that accompanies a modernisation project. A sense that once you've decided to move forward, you should move quickly. After all, the sooner you reach the new world, the sooner you can retire old systems, unlock developer productivity, and start delivering on the promised ROI. But in the world of legacy-to-Kubernetes migrations, moving too quickly is often the exact reason a project begins to unravel.

It starts innocently. A team agrees to containerise the app and deploy it into Kubernetes. They stand up a cluster, build some Helm charts, plug in a GitOps controller, and begin connecting CI/CD pipelines. In parallel, they replace the app's configuration system, overhaul secrets management, re-architect network ingress, and start planning for service mesh integration. On paper, it all looks like progress. But in reality, this approach introduces dozens (sometimes hundreds) of simultaneous changes to the application's environment. And when the app doesn't behave as expected (and it rarely does), there's no clear way to isolate the root cause.

The result is a project that becomes paralysed by uncertainty. Is the problem with the container image? Is it the pod security policy? Did the networking rules change something? Is it a DNS issue? Is there a race condition introduced by the CI pipeline? Each layer added without validation becomes another foggy lens between you and the actual issue.

The tragedy is that none of these tools or practices are inherently wrong. They just don't belong at the starting line. When you leap too far, too fast, you overload your ability to diagnose failure. You introduce architectural ambition before achieving operational stability. And worst of all, you burn team confidence. Every new issue feels like another reason to question the platform, the tools, or the decision to modernise in the first place.

KUBERNETES REWARDS CAREFUL ITERATION, NOT OVERCONFIDENCE.

The right move isn't to leap into the deep end, but to step into the water slowly, validating each change and making sure your app can still swim. You're not just changing the environment, you're rewriting the operating model. And if you skip the fundamentals, the platform won't catch you when you fall.

THE RIGHT PLACE TO START

Ask any enterprise team how they plan to modernise a legacy application, and you'll often hear a version of the same answer: "We'll move it in stages." It's a perfectly rational response. After all, replatforming a complex monolith or multi-tier application in one go is a recipe for disaster. So the plan becomes: peel off one service, containerise it, run it in Kubernetes, and leave the rest of the system where it is, on VMs, in traditional networks, or even still on-prem.

But while this approach is logical, the hybrid phase it creates is anything but simple. Suddenly, your application lives in two different worlds, each with its own assumptions, tooling, and limitations. On one side, you have Kubernetes: ephemeral, dynamic, namespace-isolated, identity-shifting. On the other, you have legacy infrastructure: static, stateful, and often burdened with tightly coupled dependencies and hidden operational quirks. Bridging these two worlds is a systemic design challenge.

Cross-world communication introduces a host of risks. Network latency becomes a variable with real consequences. Firewall rules that were never designed for intra-app traffic now need to support real-time calls across two very different environments. DNS resolution might fail because Kubernetes uses internal names unfamiliar to legacy systems, or vice versa. TLS handshakes may break if both sides don't trust the same certificate chain. Even something as basic as a shared file system might not be compatible across containerised and non-containerised workloads.

Service discovery adds further friction. Kubernetes relies on dynamic DNS resolution and service endpoints that may be spun up or down on demand. Legacy components, meanwhile, may reference hardcoded IPs or hostnames that don't change. Your app can behave inconsistently, simply because the environment it runs in alters the assumptions baked into its communication paths.

The hybrid model is not inherently flawed, in fact, it's often the only viable path forward. But treating it as a transitory state doesn't mean you can take shortcuts. Every partial step must be treated with the same seriousness as a production deployment. Every integration must be validated not just for connectivity, but for behaviour, performance, and reliability.

If you're embracing a phased migration strategy (and most teams should) then you're also signing up for complexity. That's not a problem; it's just reality. And the more clearly you see that reality, the more confidently you'll navigate it.

PREMATURE AUTOMATION: THE ILLUSION OF PROGRESS

There's a certain satisfaction in writing automation. When pipelines start to trigger, containers build themselves, and deployments roll out with a single commit, it feels like modernisation is finally happening. Dashboards light up. GitOps syncs. Infrastructure responds. But automation, seductive as it is, can give a false sense of confidence if it's built too early.

The urge to automate early is understandable. It's seen as the only "right" way modernize, a way to enforce consistency and eliminate human error. But that logic only holds when you're automating a process you understand deeply. When you're still trying to make the application work reliably in Kubernetes, building automation around it is cart before horse. You're just making it faster to deploy broken things.

Kubernetes exposes all the latent fragility in legacy applications. From missing readiness probes to poorly handled restarts/shutdowns, from bad assumptions about local disk to hardcoded IP dependencies, these are issues that automation can't fix. They can only be revealed through hands-on observation. When you deploy manually, you see what fails, when it fails, and how it fails. That knowledge is critical. Without it, your automation is just wrapping paper around a mystery box.

Moreover, automation becomes yet another variable when something breaks. Is the pod crashing because the app is broken? Or because the deployment script passed the wrong environment variable? Or because the pipeline built an image with an outdated dependency? Troubleshooting becomes a whack-a-mole exercise, with no clear baseline to return to. You can't debug a moving target.

There's also a very real human cost. Teams often spend weeks (maybe even months) automating deployments only to find that the application doesn't behave as expected. All that effort is now sunk into a delivery pipeline that delivers a non-working app. The team is demoralised. Time is lost. Worse, no one is quite sure what to fix next, because the automation hides as much as it reveals.

Instead, the smarter approach is to earn your automation. Manually deploy the application into a representative environment. Use tools to observe the system's behaviour clearly. Watch how it behaves under load. Simulate failures. Restart it. Scale it. Kill it. Log into the pods. Explore what happens when a dependency is unavailable or when a pod reschedules to a new node. Every quirk you uncover is one less mystery your pipeline will have to work around later.

Only once the manual process is well understood (when the app is stable, predictable, and reliable) should you begin to automate it. And even then, automation should be built in layers. Start with build and packaging, then add deployment, then finally add orchestration logic, such as canarying, rollbacks, or sync policies. That sequencing ensures you're never automating blind.

AUTOMATION IS CRITICAL, BUT IT MUST COME AFTER UNDERSTANDING.

Otherwise, it's just a polished version of the same old uncertainty. And in a modernisation project, false confidence is far more dangerous than honest failure.

THE TRAP OF PERFECT: ARCHITECTURE BEFORE RELIABILITY

In most modernisation projects, there's a moment early on when someone draws the "target state" diagram. It's an impressive thing: multi-cluster, multi-region Kubernetes with GitOps automation, zero-trust networking, circuit breakers, metrics, tracing, and full progressive delivery. Boxes and arrows flow across the screen with enviable confidence. Everyone nods. The diagram is beautiful.

But diagrams don't run applications.

And this is where many teams lose their footing. They become so enamoured with the end state that they skip the messy, manual, essential work of making the application run well today. They want the perfect architecture, fully automated, with all the bells and whistles, but forget that the foundation of that architecture is a reliable, well-behaved application. Without that, no amount of automation, infrastructure, or "platform engineering" will make the system stable.

The trap is believing you can design your way to reliability. That if you just make the architecture sophisticated enough (introduce a service mesh, or bake in retries, or enforce memory limits) then the app will rise to meet it. But in reality, most legacy applications don't care how elegant your platform is. They care about what they need to function: access to the right services, a predictable runtime environment, reliable storage, and well-understood failure behaviours. If those aren't in place, the rest is decoration.

This obsession with "right the first time" thinking creates enormous project risk. Teams waste months building out deployment blueprints, writing operator specs, and codifying patterns that haven't yet been tested against real workloads. They bake in assumptions about how the app should behave instead of validating how it does behave. And when things start breaking in production (or worse, just underperforming) it's too late. The foundation was never proven.

There is a better approach, though it requires humility. Start simple. Deploy the app manually into a controlled environment that mimics production. Observe everything. Does the app crash when rescheduled? Does it leak memory over time? Does it handle load in a container the same way it did on a VM? Can it survive a node reboot? Can it tolerate a dependency being slow or unavailable?

These are the questions that tell you whether your app is ready to be automated and scaled, not whether your platform is ready to do it. Kubernetes is brilliant at orchestrating predictable workloads. But it is utterly unforgiving to those that aren't. The work of modernisation is to make your application predictable first: reliable, recoverable, observable. Then, and only then, should you start wrapping it in layers of automation and resilience.

PERFECTION WILL COME.

But it must be built on certainty, not speculation. Focus first on proving the app works. Not just once, but repeatedly. Only when it behaves consistently under known conditions should you reach for the architecture diagram. Until then, it's just wishful thinking in a pretty box.

TEST FIRST, AUTOMATE LATER: THE FOUNDATION OF MODERNISATION

In the rush to modernise, most teams begin with pipelines. It's almost instinctive, standing up a CI/CD system, pushing container builds into a registry, and automating deployments into a shiny new Kubernetes cluster. But this enthusiasm masks a deeper problem: if you're not testing what you're deploying, you're not modernising. You're just moving faster into uncertainty.

Before you write your first deployment script, before you commit to a GitOps flow, you need something far more valuable: confidence. Not vague confidence in your tooling or infrastructure, but precise, deliberate confidence in how your application behaves in this new world. And confidence is earned through testing, real testing.

Modernisation without testing is like refitting an aircraft mid-flight and assuming it'll keep flying. In reality, containerising an application introduces a thousand small behavioural shifts: process lifecycle changes, network interface behaviour, storage persistence quirks, and failover edge cases that never surfaced in a static VM. You might get lucky once, but you won't get reliable.

What you need before any automation is a comprehensive testing framework tailored to your application. Not just unit tests or API checks, but validation that reflects real-world usage:

- Can the app recover cleanly from a pod crash?
- Does it maintain session consistency during rescheduling?
- Does it gracefully handle dependency latency or failure?
- Does it consume memory in line with expectations?
- Can it be horizontally scaled without breaking internal logic?

These aren't theoretical questions. They're the day-to-day realities of operating on Kubernetes. And they're best answered early, manually, and repeatedly.

Start by creating a synthetic production environment, a controlled setting where you can mimic the key elements of your live setup: namespaces, networking, storage classes, resource limits. Manually deploy your application here. Run your tests not once, but as often as you can. Don't optimise for elegance. Optimise for learning. Each test tells you something about how the app interacts with the platform, and each failure is an opportunity to adapt before it's baked into an automated pipeline.

Critically, these tests become the bedrock of your eventual automation. When you do reach for pipelines, those same tests now serve as your guardrails. They provide the feedback loop that tells you whether a deployment is working, not just whether it completed. They turn automation from a blind fire-and-forget process into a controlled, iterative flow.

The value of testing isn't just technical, it's strategic. It builds alignment between Dev, Ops, and Platform teams. It uncovers implicit dependencies that would otherwise become production issues. And it teaches your organisation how to operate, not just deploy.

So before you automate, validate. Before you commit to a GitOps flow, commit to a repeatable testing process. And before you scale, ensure that the core experience of your application in Kubernetes is known, stable, and measurable.

Because modernisation is not about speed. It's about confidence. And confidence, in this world, is measured in tests passed, not pipelines built.

FINAL THOUGHT: EARN THE RIGHT TO MIGRATE

By now, it should be clear that modernising a legacy application onto Kubernetes isn't a technical task, it's an organisational one. It's as much about building confidence in your approach as it is about building containers. It's a process of steadily earning the right to take the next step, not leaping forward based on assumptions or ambition.

Modernisation must be treated as a series of deliberate, validated moves. And with each move, the goal isn't just to get the application running, it's to prove that your approach is reliable. That your team can migrate a component into Kubernetes, keep it running stably, handle issues when they arise, and do it all without derailing schedules or draining budgets. That's where the real value lies, not just in technical success, but in predictable delivery.

When you take a stepped, cautious approach, you generate something far more valuable than code: you build credibility. You prove to your executive stakeholders (both IT and business) that this isn't just another overhyped transformation initiative. It's a plan grounded in realism. A plan that respects complexity and mitigates risk. A plan that delivers results you can trust.

Each successful component migration becomes a proof point. Each application brought into the new platform becomes a visible milestone. And with every milestone hit on time and on budget, you accrue what can only be described as "modernisation credit", the intangible but powerful trust that gives you room to move. That credit can then be spent wisely. Not recklessly, but strategically.

Maybe you want to adopt a service mesh to improve east-west traffic observability. Maybe you want to shift to serverless patterns for batch jobs. Maybe you want to re-architect how your teams work with infrastructure entirely. But those moves, those bigger, bolder bets, should come after you've proven the foundation is solid. After you've shown you can migrate, stabilise, validate, and support.

Too often, teams try to spend that credit before they've earned it. They chase modern patterns too early and lose momentum when things go sideways. But if you build up that credibility gradually (by doing the hard, manual, sometimes boring work first) you'll eventually have the trust, the data, and the track record to propose more ambitious changes. And when you do, you'll have the full weight of organisational backing behind you.

SO DON'T RUSH. DON'T CHASE ARCHITECTURE DIAGRAMS OR BUZZWORD CHECKLISTS.

Focus instead on making each step of your modernisation journey repeatable, predictable, and understandable. Migrate slowly. Test thoroughly. Validate everything. And in doing so, you'll earn not just the right to migrate, but the mandate to lead your organisation confidently into the modern era.

PORTAINER.iO