Building and operating a high-performance platform for a retail organisation

By Josh Roberts
Analyst at BJSS

When it comes to global retail, a high-performing platform is more than just a technical achievement – it's a competitive necessity. This article dives into how we built a resilient, scalable, and continuously deployable platform, balancing automation with cultural transformation.

We explore the strategies, challenges, and breakthroughs that enabled us to support thousands of deployments with minimal disruption. But technology alone isn't enough – success hinged on shifting mindsets and reimagining ways of working.

What does it mean to be an 'elite' platform team?

DevOps Research and Assessment (DORA) and SPACE are well-known metrics used to measure the performance of Platform or DevOps functions in an organisation. To be recognised as "elite" by DORA standards, teams need to excel in four specific areas:

- **Deployment frequency:** How often an organisation successfully releases to production
- **Lead time for changes:** The time it takes for a commit to make it into production
- **Time to restore service:** How quickly organisations recover from a failure in production
- **Change failure rate:** The percentage of changes that fail in production

To achieve DORA elite status, teams must meet the following criteria:

- **Deployment frequency:** On-demand (multiple deploys per day)
- **Lead time for changes:** Less than one day
- **Time to restore service:** Less than one hour
- **Change failure rate:** Less than 15%

Only about 20% of teams achieve elite status across all these metrics. These kinds of high-performing behaviours are far more common in companies that are technology-first, such as the tech giants and e-commerce players who operate without any physical retail presence.

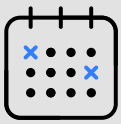
Take a look at some of the posts on this topic from household names:

- How Netflix Became a Master of DevOps
- Etsy: The Secret to 50+ Deploys Each Day
- Continuous Deployment at Facebook
- Amazon: Automating safe, hands-off deployments

For organisations where technology isn't the primary business model, achieving elite status can be much more challenging. As an engineering consultancy, we are often brought in to help non-tech-first companies – such as government departments, utilities, and, in this case, a major global retailer – to transform their digital platforms.

Of all the DORA elite metrics, achieving true continuous deployment – where changes merge into production on demand – proved to be the most difficult.

Why DORA Elite?



Traditional release

Increases complexity and risk

vs

Frequent releases

Reduces risk and increases speed

Most of the DORA metrics are straightforward. Nobody wants changes to production that break the system. If issues arise, you naturally want to restore service as quickly as possible. However, deployment frequency and lead time for changes can feel more alien to some teams that aren't used to working in this way. Many organisations are comfortable releasing changes once a sprint or once a month – they follow a rigidly defined plan, adhere to a Change Advisory Board (CAB) process, and everything seems to work fine.

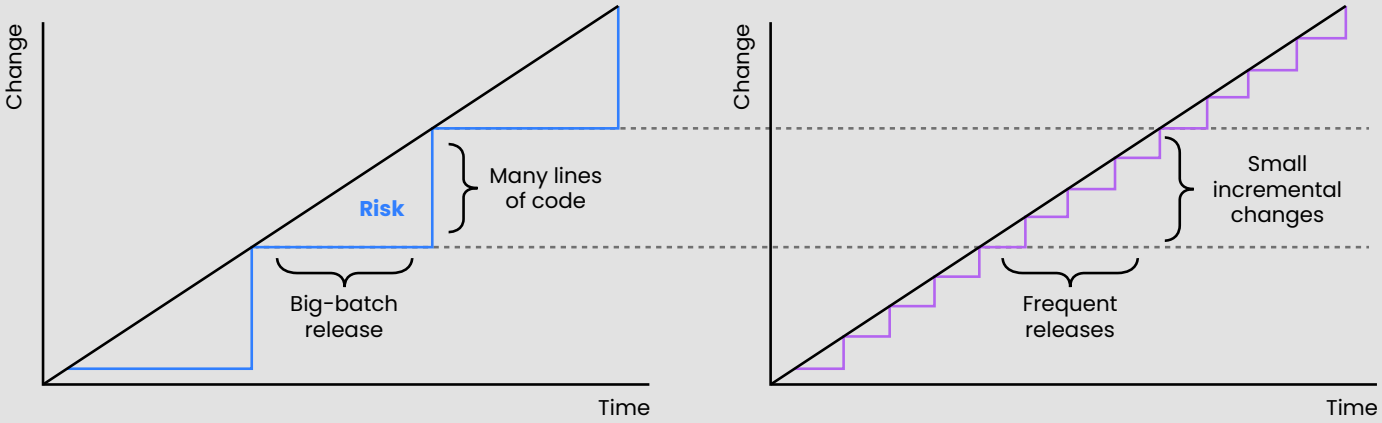
The problem with this approach is that releases with lots of changes create complexity, and complexity creates risk. By releasing small, frequent changes, you significantly reduce the risk and simplify troubleshooting.

In fact, when looking at the value added by management approval to releases, the DORA 2019 report highlights:

"We found that external approvals were negatively correlated with lead time, deployment frequency, and restore time, and had no correlation with change fail rate. Approval by an external body (such as a manager or CAB) simply doesn't work to increase production stability... However, it certainly slows things down."

CABs and similar processes can be valuable, especially when managing particularly disruptive changes. However, for many routine changes, organisations can benefit from streamlining approvals through automation and cultural shifts.

SPACE is another set of metrics that we can apply alongside DORA. DORA focuses primarily on deployment performance, SPACE by contrast measures aspects of team collaboration, project management, and developer productivity that are not directly addressed by DORA metrics.



It seems counter-intuitive at first to think that more releases make production safer. The basis of the principle is that the less you change at once, the less likely it is to go wrong, or that any QA process won't have identified an issue.

Understanding the client's challenges

The retailer we're working with has a huge brick and mortar presence – with thousands of stores across the world, but they were having serious issues with their legacy digital platform. It was a monolith with difficult monthly releases that required downtime and extensive manual testing.

Post-release incidents were frequent, often requiring hotfixes or rollbacks. Where rollbacks were required, even this wasn't simple – extending the total amount of downtime while managing the release.

While obviously a very poor outcome for the client, this was also an extremely stressful environment for the engineers to work in and maintain.

The client wanted a complete re-platforming of their digital offering using a MACH (Microservices, API-first, cloud-native, headless) architecture. They also aspired to adopt CI/CD practices, aiming for changes to move from being development complete to being deployed in production within just 60 minutes.

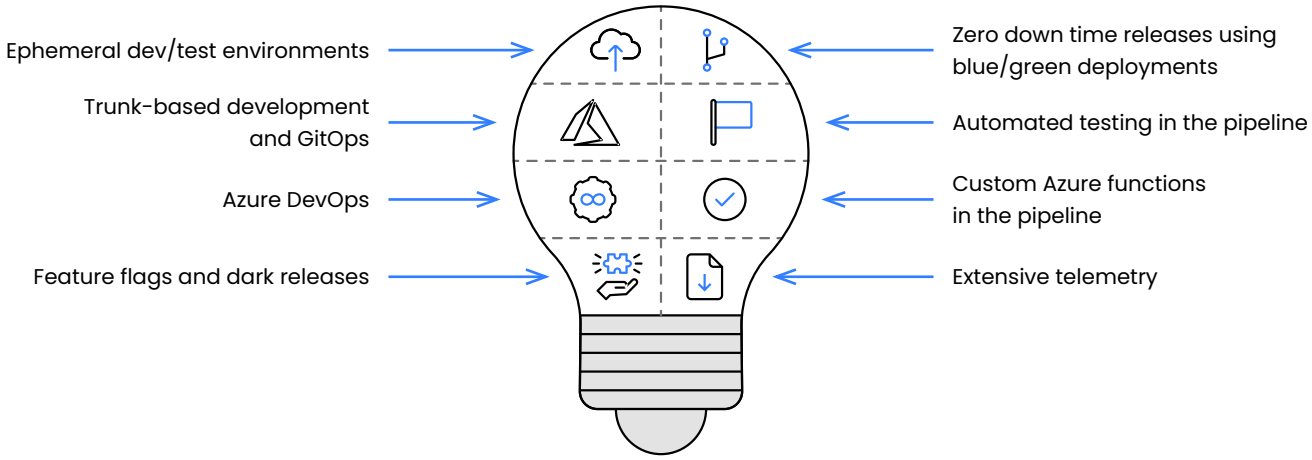
The system handled a wide range of functions for the retailer:

- **Content management**
- **E-Commerce**
- **Product management**
- **Appointment bookings**
- **Store management**

Different teams and suppliers worked on replacing various parts, with staggered timelines for replacement across different markets. This meant we needed a strategy to 'strangulate' the monolith and decommission different parts of it, in different markets, at different times, depending on the needs of the market and the progression of the disparate development teams.

The technical challenges

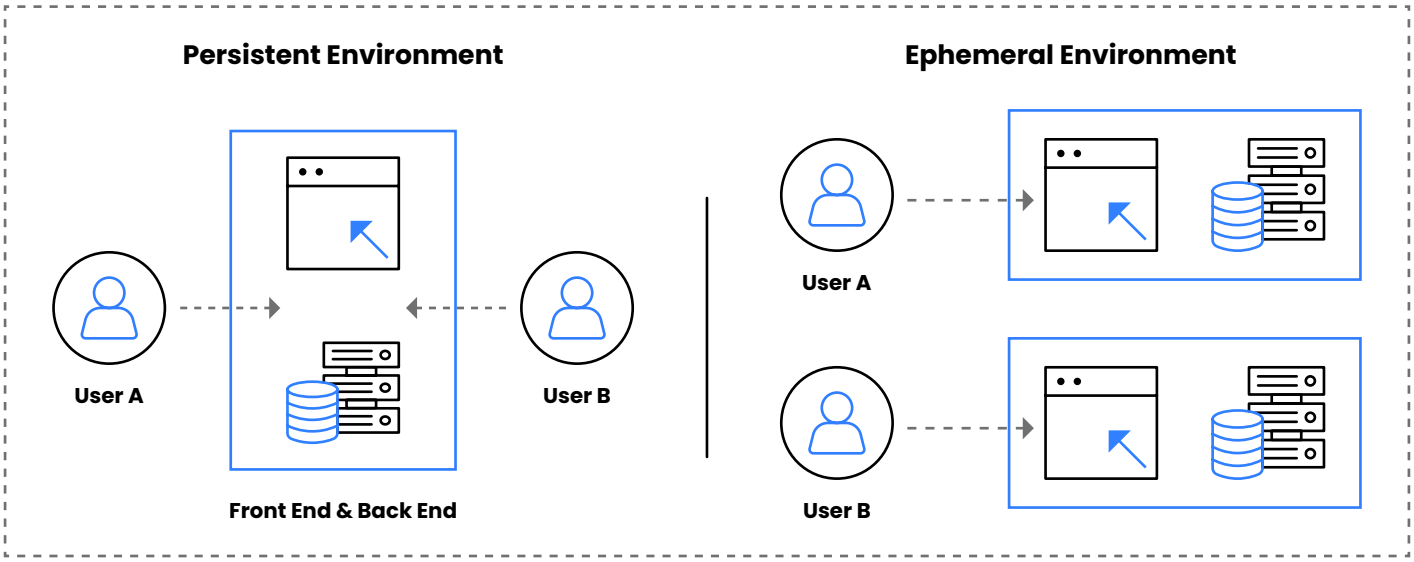
The diagram below summarises the key technical solutions that enabled us to hit those DORA elite metrics.



Ephemeral environments

We automated the creation and teardown of hundreds of ephemeral environments daily. Each environment was created when a developer proposed a change, and their change was deployed into it for testing. Automated tests ran in these environments, ensuring that changes met quality standards before merging into the main branch. These environments also allowed developers to manually test or demo their changes with stakeholders if needed. This approach eliminated the classic "well, it worked on my machine" excuse.

In the diagram below, the User is a developer. They all get their own environment.



Trunk-based development

We adopted trunk-based development, storing all infrastructure, configuration, and application code in Git repositories. Component repositories held application code and used Terraform for deployments. To maintain a clean history, we enforced strict branch naming rules and squash merges, linking every commit to a Jira ticket for traceability.

All branches merged into a single main branch and shared the same environments, ensuring consistency. This structure also allowed us to replicate environments easily – a significant advantage for deploying into new markets with slightly different configurations. Additionally, using Terraform ensures that any manual changes made to production are automatically reverted, keeping the system aligned with the desired state. This reinforces consistency and reliability in our deployments.

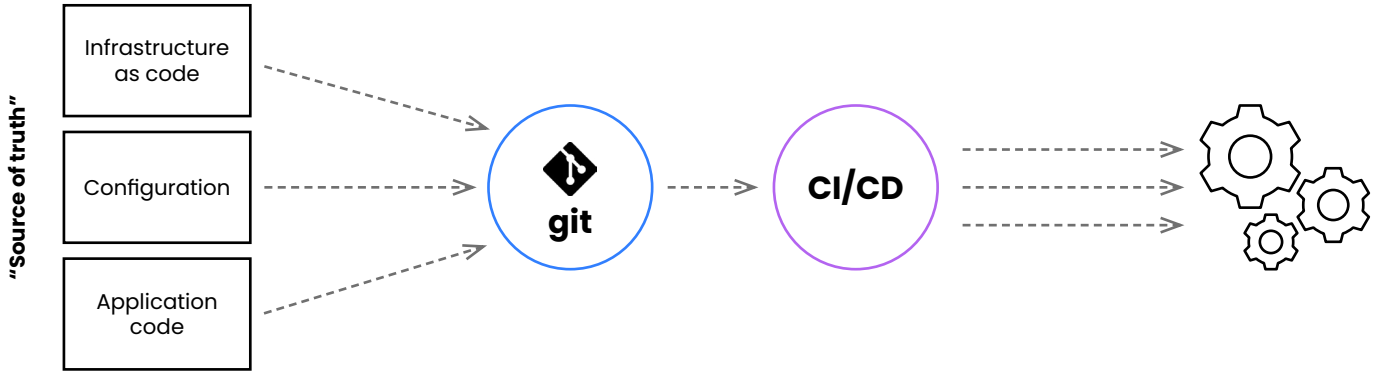
GitOps

On the GitOps side, we bundle multiple components – along with their infrastructure and the tests that validate that infrastructure – into a single version representing the system as a whole.

Every change, whether it's to the tests, configuration, infrastructure, or software, is built, bundled, and released in the same way. This approach provides absolute assurance to the release management team that every change to the system is tracked, versioned, and managed. It also simplifies the release process, as all changes follow the same procedure, allowing us to focus on technical concerns only when they cross system boundaries.

The way we version all components and roll them into a single whole system version makes it trivial to roll back, it also allows us to easily spin up ephemeral environments at a point in time and determine if a system behaviour (such as performance) changed between different versions.

In the diagram below, updates to the source code trigger a pipeline, which runs a series of tasks to update runtime environments so they match the source.



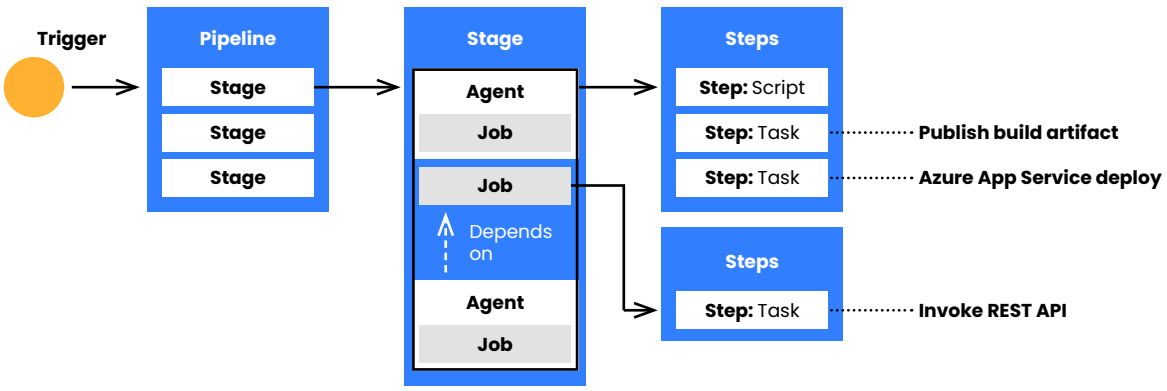
Azure DevOps

We use Azure DevOps (ADO) to manage our pipelines, repositories, and releases, but we don't use all of its features. Given the size of the client organisation and its well-established tools and ways of working, we collectively decided to align with existing practices rather than introduce significant changes.

So, for instance, we avoid using ADO's project management tools, even though its integration with builds works out of the box. Instead, we prefer Jira, as it's the client's existing tool and is widely adopted. This required us to build several custom integrations between JIRA and ADO to improve developer productivity. For example:

- Linking ephemeral environment URLs to Jira tickets
- Tracking if a ticket is closed to tear down environments
- Associating build IDs, deployment IDs and pull requests to Jira tickets

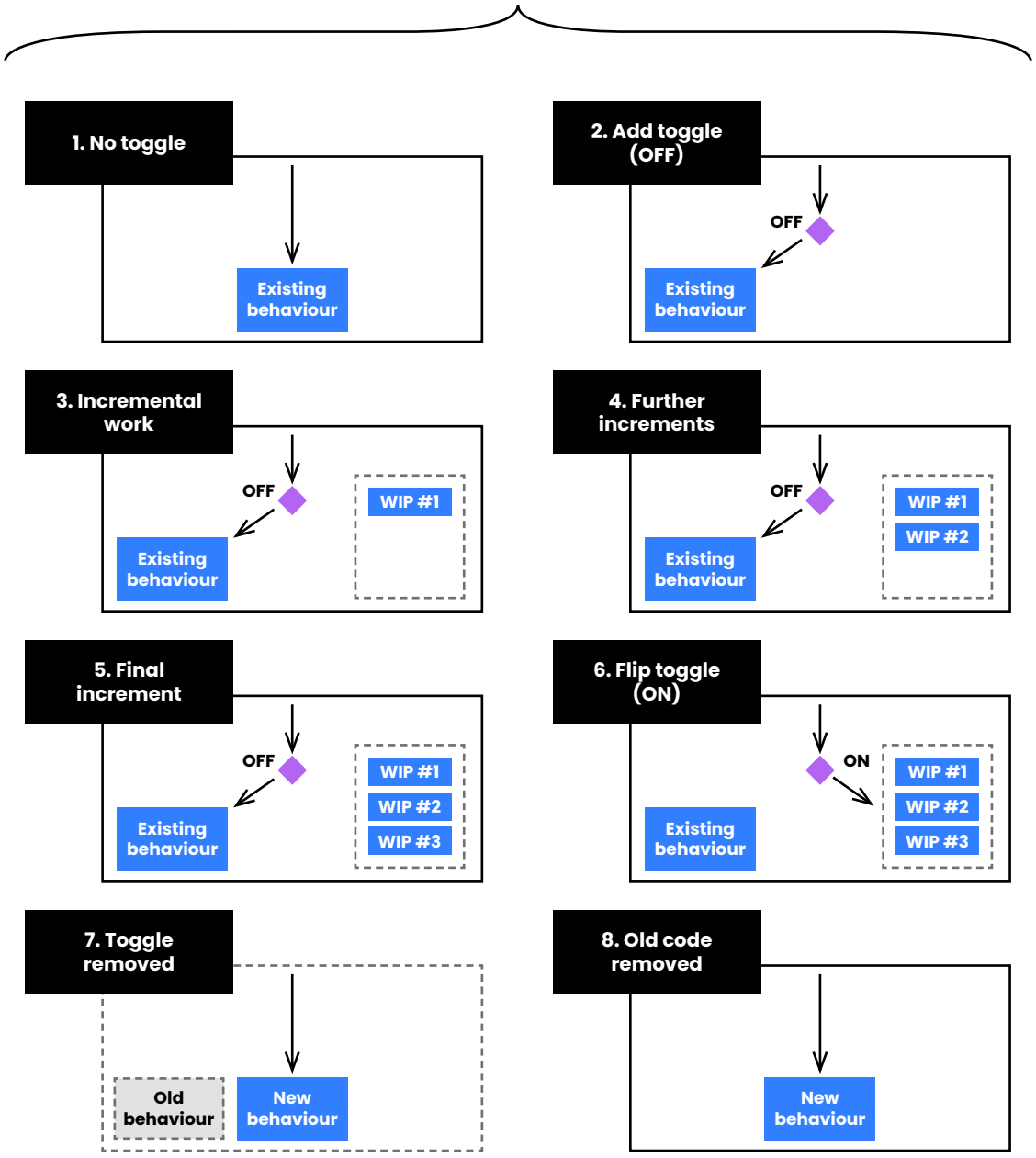
All of our pipelines were multi-stage, and our mainline deployments included multiple environments.



Feature flags and dark releases

- We used Cloudflare Workers to enable controlled routing and dark releases
- Teams could release new journeys to production well before their official go-live date
- The Worker router ensured that end users were only directed to the new application under specific conditions
- This allowed us to continuously deploy into production, test internally, and ensure everything worked before a full rollout to end users
- Final cutovers became trivial – all it took was flipping the routing
- Once fully deployed feature flags also further enabled gradual rollouts of enhancements. This allowed us to make incremental changes safely, ensuring we could roll back quickly if necessary

This diagram, from BJSS Platform Architect Stuart Bass, shows how we use dark releases to build up an application in production that's hidden and then eventually switch to it and remove the old application.



Zero downtime releases

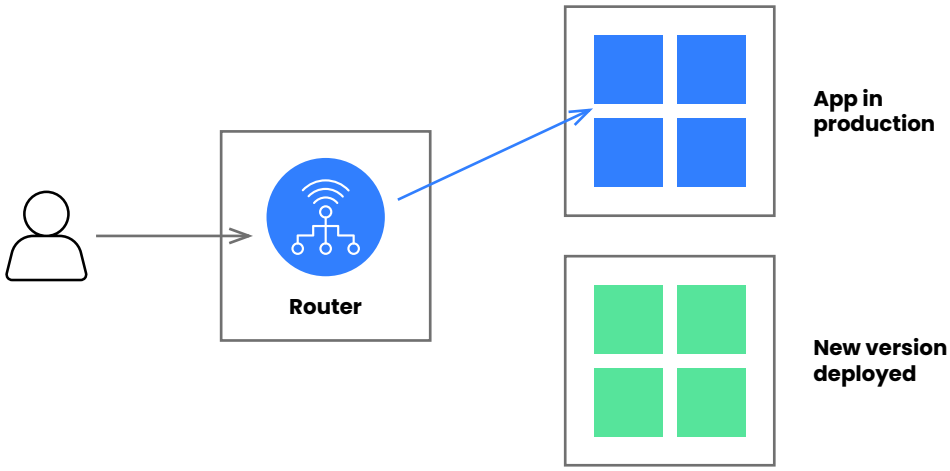
To support continuous deployment without disrupting users, we adopted a blue/green deployment strategy. Cloudflare Workers managed all of our routing and directed traffic to the "live" version of the application while we deployed updates to a non-live instance. Once the updates were validated, we switched the traffic to the new version seamlessly.

This approach worked well but had one critical limitation; any changes to the Worker itself had to be made outside of business hours due to the significant impact they could cause.

To address this, we leveraged Cloudflare's Preview/Publish functionality, which allowed us to run tests on the 'preview' instance of the worker before promoting it to Production.

This functionality also proved valuable for our frontends, which we migrated to Cloudflare Pages from Azure Storage Accounts. Cloudflare's Preview/Publish feature for Pages effectively provided built-in blue-green deployment. We only needed to ensure that our tests targeted the correct instance before requesting promotion to production.

The blue 'App in Production' is the one being used by end users. We deploy a new version to 'green' and then switch the routing using the Cloudflare Worker.



Automated testing

Eliminating reliance on manual testing for most changes is essential to increasing delivery velocity. We run tests with varying coverage across all environments, using different frameworks based on developer preference. Component tests are executed during application build in an ephemeral test environment, often numbering in the hundreds or thousands per workstream, providing comprehensive coverage.

If these tests fail, the change cannot be merged into the main branch. The component tests are backed by real world Azure resources which are optimised to be spun up rapidly. This means we get tests on a resource which is akin to production.

Smoke and integration tests are run on the deployed ephemeral environment to provide additional coverage after code deployment. Once merged into the main branch, smoke and integration tests are run on our Mainline QA environment.

Re-running these integration tests allows us to validate any delta that might have crept in between the creation of the ephemeral test environment and the deployment into the mainline branch.

Afterward, only smoke tests are executed on staging and production environments. These smoke tests are always run in a separate pipeline stage and will block deployment if they fail.

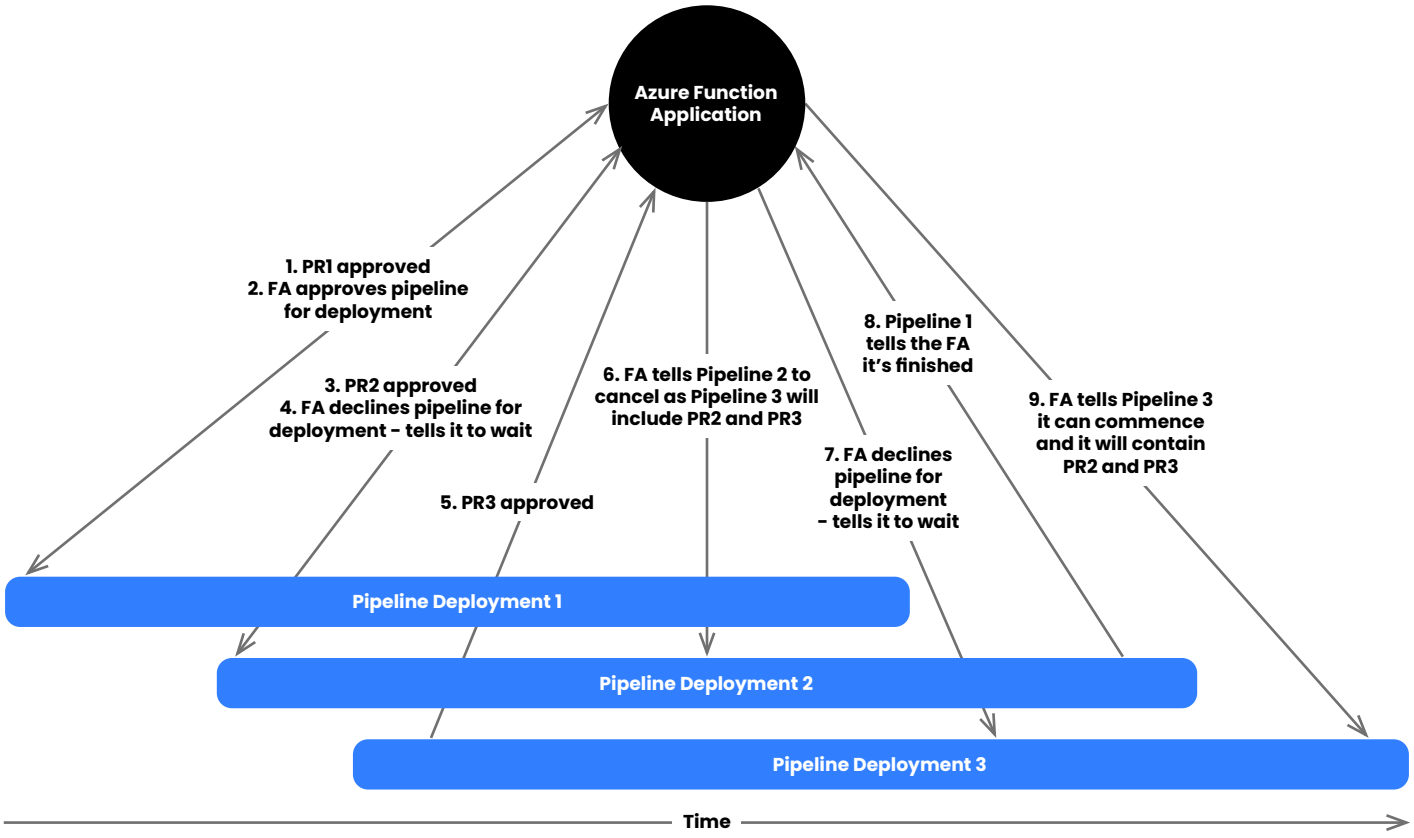
Various test frameworks are used, such as JEST, Selenium, Playwright, and Cypress, depending on the team's preferences and expertise. However, the core principles of testing remain consistent across pipelines, regardless of the code being deployed or the team responsible.

Custom pipeline functions

To gain the client's approval, we agreed on a set of business processes for Continuous Deployment, which I'll detail later. These processes required us to build custom software to execute within the pipeline. The key functions included:

- A check to identify if a developer had marked their change as non-negligible in a Jira ticket, which blocked the change from merging into the main branch without explicit release management approval
- A locking function to ensure that when multiple developers attempted to merge changes simultaneously, the merges would queue appropriately, one after the other
- A validation function to ensure pull requests were associated with valid Jira tickets
- A "big red button" feature, enabling both technical and non-technical users to quickly suspend continuous deployment with a single click, halting any further deployments to production
- Additionally, we automated the generation of release notes, which had previously been created manually on a daily basis. This process was no longer feasible with multiple daily releases.

This diagram explains how our locking function works when multiple developers try and merge their changes into the same pipeline at the same time.



Extensive telemetry

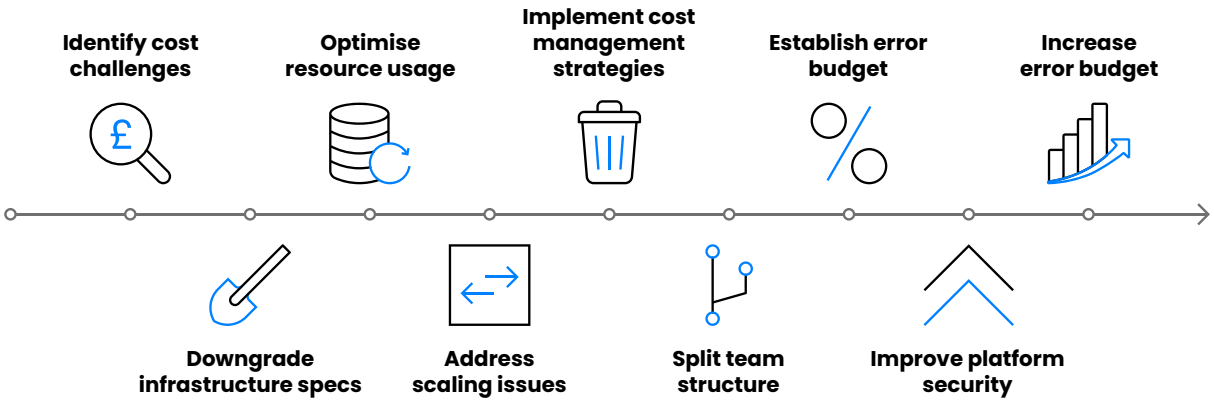
Describing the full scope of monitors across all our pipelines would require an article of its own. However, the key shift has been the evolution of the "release engineer" role.

Previously, this role was responsible for managing production releases, often just approving a release with a button click. Now, the role focuses on responding to monitors and alerts when something goes wrong during deployment.

For example, a deployment stage might fail due to an unstable CDN or unavailable Terraform provider. Release notes may fail to generate if someone bypasses naming convention validation. Additionally, test failures often require investigation to identify the root cause.

These alerts trigger into monitored BJSS and client Teams channels. Named individuals are responsible for responding to them. Where the alert is set to a high severity, it can also trigger a JIRA ticket to the service desk so that we can link an event to an incident.

The challenges



Building a platform like this presented numerous challenges. As a consultancy, one of our primary concerns was managing costs for our client.

As the use of ephemeral environments grew, so did the potential for escalating costs. To control this, we downgraded the spec of the provisioned infrastructure to something closer to a Raspberry Pi, which was sufficiently performant for most dev/test environments. For developers requiring higher performance, they could select a more robust profile.

We also optimised resource usage by having ephemeral environments share more expensive resources. For example, instead of provisioning hundreds of expensive Postgres database servers, one for each environment, all environments connected to a shared server which could then provision databases for each environment.

This approach led us to our second major challenge; scaling. We encountered several limits in Azure, such as the number of access policies per key vault, number of permitted resource groups per subscription, and storage accounts permitted per subscription.

To overcome this, we created pooled subscriptions across multiple Azure regions. Each dev/test ephemeral environment was provisioned in one of five pooled subscriptions, which also contained shared resources specific to that region.

Managing the cost of these environments required rigorous housekeeping. While most of the time, resources were properly decommissioned, the scale of daily environment creation meant that missed decommissioning could lead to significant costs over time. We introduced some key fail-safes:

- If a branch hadn't been updated in a few days, we would automatically remove the associated environment. the developer had 30 days to spin it back up again if needed.
- We set up monitors and alerts for environments that had been active beyond a certain threshold, prompting manual review to determine if they were still needed. This also helped identify gaps in the automated housekeeping process.

As our team and platform grew, we had to reassess our structure. We split the team into two main groups, one focused on 'build' activities and the other on 'operations.'

The distinction was fairly loose, but the key difference was that the 'build' team had protected time for long-term initiatives, while the 'operations' team was responsible for reacting to alerts and managing performance.

The operations team's main responsibility was ensuring the performance of our test environments, associated pipelines, and developer experience. While they were also responsible for production, that environment was stable and rarely required intervention.

To further support the operations team, we implemented an error 'budget.' Initially set at 80%, the operations team was expected to drop everything and investigate if more than 20% of dev/test environments failed to deploy. Once the root cause was identified, they would either fix it quickly or work on a longer-term solution with the build team. As we refined this process, our platform's stability improved, allowing us to increase the budget to 95%.

We also want to continue to optimise our build and deployment pipeline speed. Our next major challenge at the time of writing this is separating our frontend and backend deployments for one of the major applications. At this point in the application's lifecycle the vast majority of changes are made in the frontend and not the backend. Yet each deployment pushes out a brand new version of a chunky backend.

By separating these, the goal is to get the frontend changes – over 80% of the change made on this application – to deploy to production within 15 minutes after it has been committed into the main branch.

We now deploy to production tens of times a day, across multiple pipelines and development streams, into different subsidiary brands and markets. We also have a setup that makes deploying to new infrastructure regions and markets a relatively trivial exercise.

We supported hundreds of developers working concurrently, using thousands of ephemeral environments each month. We automatically deploy and destroy tens of thousands of resources every month. Our extensive monitoring allows us to proactively respond to scaling issues and ensure we maintain a stable and performant platform for our developers.

We've had no significant disruptions to production as part of our releases, and even minor issues are identified and resolved within minutes or hours – either by fixing forward or rolling back a single change.

Whenever I talk about Continuous Deployment at BJSS, the people who find it most exciting are the engineers, who want to discuss the technical implementation.

While I think what we've built is great, ultimately, there are many extremely talented engineering teams at BJSS and across the industry who can understand what we've built and replicate or tailor it to their needs.

As with all things technology and transformation, the implementation is only half of the story. Getting us to the point where we could trust automation to fully manage our deployments required a huge amount of influencing, coaching, and education – for both developers and managers, at BJSS and the client.

Cultural and business change

While cultural change is as crucial as the technical changes that helped us achieve "elite" status against the DORA metrics, it is deeply context dependent. Unlike technical implementations, which can often be replicated with a reasonable assurance of success, cultural shifts are influenced by people, their experiences, and the unique dynamics of their teams. **That's why cultural change is not a universal blueprint and isn't easily replicable.**

Every team I've worked with has required a tailored approach, and even within the same team, strategies evolve as they grow. Writing about this risks oversimplification, especially in an era where oversold one-size-fits-all solutions abound. What follows is a record of what worked for us in achieving continuous deployment (CD).

Ultimately, the biggest determinant of success was the calibre of people involved. At BJSS, we're fortunate to work with intelligent, capable engineers, analysts, and leaders. High-performing teams aren't built on processes alone – they're built on great people. So, before diving into specifics, I'll say this: if you don't trust your people or they lack the capability to take ownership of their work, no cultural change initiative will ever succeed.

Being set up for success

Early in the programme's design phase, client and BJSS senior leaders and architects set out their vision for re-platforming away from the legacy system. This vision included a strong aspiration for continuous deployment, ensuring that the goal was embedded from the outset – both in the architecture of the applications and the underlying infrastructure that would deploy them.

Having the opportunity to design a greenfield platform was a fortunate starting point, and securing the client's buy-in early was critical.

Anyone involved in frontline delivery knows that implementation can sometimes diverge from the initial strategy. However, we gained the confidence of key senior technical stakeholders, and it was up to us to demonstrate that we could turn the strategy into reality.

By implementing the foundational technical solutions described above, we reached a point where we could release some of our pipelines daily. This achievement demonstrated to the client that continuous deployment wasn't just an ambitious goal but an achievable one.

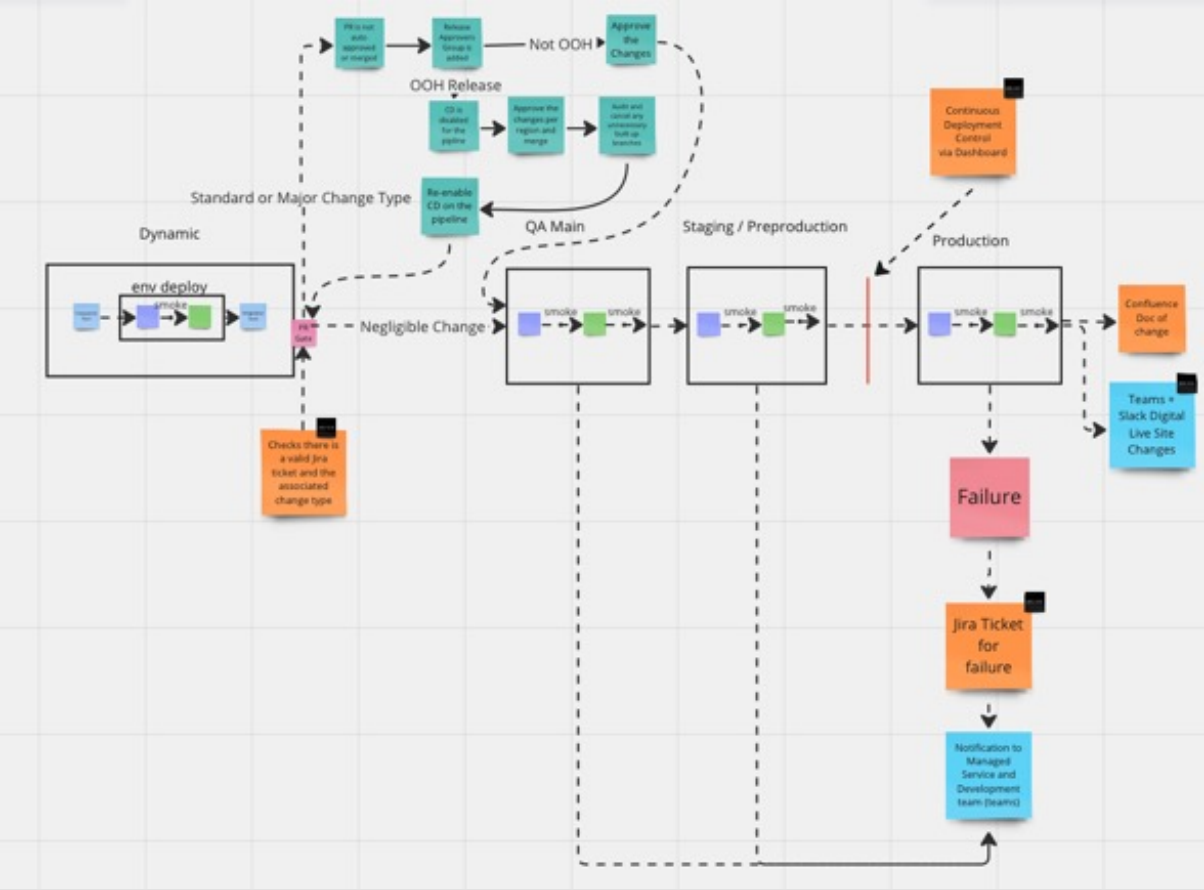
While we touch on how we achieved daily releases, this section focuses heavily on our journey from daily releases to on-demand deployments – the final "elite" DORA metric we needed to meet.

The process

We reached a point where we were releasing daily across multiple pipelines for different business streams. Each morning, an engineer joined a call with a release manager to identify pipelines with changes, trigger releases, and monitor the process. Releases were done sequentially for manageability, even though parallel releases were technically feasible. In 95–99% of cases, these releases went smoothly. However, the process was frustrating: one engineer was effectively lost to this task for half a day every day.

Daily releases initially helped build confidence with the client's release manager, who had previously overseen manual monthly releases that often failed. Over time, daily releases became routine and added little value. Despite this, human oversight persisted, their presence was effectively a 'fail-safe' that we'd grown accustomed to having.

To evolve, we needed a new process. A group comprising the client release manager, BJSS's service management team, a platform architect, platform engineers, and developers from a pilot workstream came together to map out a new workflow, this is what came out of the workshop.



The guiding principles were:

1. **Validation at the pull request stage:** Developers could experiment freely in ephemeral environments, but merging into the main branch triggered validations to ensure changes were suitable for an automatic release into production
2. **Redefining the path to live:** QA, staging, and production – our path to live environments – were all treated as production-grade. Failures at any stage were taken seriously and investigated
3. **Gatekeeping for non-automatable changes:** We needed mechanisms to flag changes unsuitable for automation, including processes for managing off-peak releases requiring downtime
4. **Emergency off-switch**: A centralised, easily accessible stop mechanism was essential for halting deployments in critical situations. This feature was designed for both technical users and non-technical managers
5. **Audited and automated announcements:** Release notifications to the business were automated to reduce manual overhead, especially as release frequency increased.

Choosing the first team

We began with a lower stakes workstream to pilot the process and the tooling for continuous deployment – the content management system (CMS). While content is important, its revenue impact is less direct than systems like e-commerce or appointment booking. A failed CMS release might inconvenience internal users but wouldn't immediately affect the bottom line.

Crucially, the selected team already prioritised automated testing and considered ownership of the release pipeline a shared responsibility. They actively responded to failures, collaborated with platform engineers, and viewed daily releases as an inconvenience. This cultural foundation – trust, accountability, and engagement – was essential for piloting CD.

Building trust and going live

Rolling out our first CD deployment felt like launching a new product, and we approached it as such.

- **Live demos:** We demonstrated the process and features to the release manager, including edge cases
- **Acceptance criteria:** Clear criteria were established to validate readiness
- **Backlog management:** Non-critical bugs and enhancements were logged for future iterations. We worked hard during the development phase to limit scope creep and stick to an MVP
- **Early life support:** Nominated individuals provided focused support during the initial rollout phase
- **Transition to business as usual:** After a stable period, support responsibilities rotated among the team

Treating the platform as a product helped us gain stakeholder trust, avoid scope creep, and focus on delivering a minimum viable solution. The real learning began post-launch – as with any product, and that's where we wanted to focus on getting to. There's a lot written out there already on treating platform as a product, and I could probably write a separate article on how we've tried to do this for our platform as a whole.

High-trust environment

Psychological safety – the freedom to express concerns, ask questions, and make mistakes – was critical. Continuous deployment can be intimidating, especially for developers accustomed to manual processes. We created an environment where developers could:

- Repeatedly ask the same questions without fear of judgment. This helped us to update our documentation if it wasn't clear enough for the developers
- Feeling comfortable to mark tickets as requiring manual release if they felt uncertain. Even if the ticket was suitable for automation, this triggered a discussion and allowed us to work through the criteria with the developers
- Participate in discussions about failures and fixes to build confidence over time

This high-trust environment reduced anxiety, encouraged autonomy, and fostered collaboration between developers and platform engineers.

Adapting to team needs

As we rolled out CD to more teams, we tailored the process to their specific requirements:

Flaky tests

One of our workstreams faced a recurring issue with tolerated test failures. After much debate, we implemented a monitoring system that triggered alerts in a public Slack channel. These alerts tagged the last committer and requested an investigation into the failure.

The primary debate was whether this approach might be perceived as too accusatory toward individuals. Ultimately, we decided to emphasise the principle of ownership, a cornerstone of our continuous deployment strategy.

If you were the last committer and a test failed – regardless of whether it was caused by your change or something else – further deployments to the main branch were paused until the issue was resolved. Resolution could involve a forward fix, a revert, or, if it wasn't easy to solve, convening a team to identify the appropriate solution.

Our priority was to avoid a culture where failures were dismissed as "interesting" or assumed to be someone else's responsibility. This approach fostered accountability and significantly improved test reliability.

Third-party dependencies

In our staging environment, where third-party integrations were less stable, we implemented monitors to detect test failures. We had a function that paused a deployment if the tests failed, enabling testers to review the failure and decide whether to bypass it because the root cause was due to a faulty third-party system – the associated change did not require a test against that third party system.

Testers were required to justify their decisions in a dedicated Slack channel where they were alerted. This process kept change flowing at pace when failures were outside of our control. It also fostered discussions about the validity of the failure and the decision to bypass, while also prompting us to identify improvements to mitigate our reliance on unreliable third-party systems.

Evolving roles

The shift to CD transformed roles across the board. Release engineers transitioned from manual approval to responding to telemetry alerts. The release manager, previously heavily involved in daily calls, became a strong advocate for CD principles across the client's organisation. Ownership of deployments became a shared responsibility between developers and platform engineers, with everyone focused on delivering changes safely and efficiently.

The release manager and a dedicated engineer to support releases became points of escalation, rather than day to day operators of a manual process.

Lessons learned

- **Ownership is key:** Developers and platform engineers shared responsibility for ensuring changes reached production safely
- **It's not done until it's in production:** The definition of done was updated. A change was considered complete only after merging into production and functioning as expected, rather than just merging into the main branch
- **Post-mortems drive improvement:** Detailed reviews of incidents helped identify lessons and prevent recurrence
- **Documentation matters:** While promoting interactions between people is critical, to achieve a stable business as usual process, accessible, clear documentation builds confidence and autonomy among developers. Having some basic documentation to start with that we could iterate on over time with developers as they used the platform was helpful.

Cultural change is as much about people as it is about processes. At BJSS, our success with continuous deployment stemmed from a combination of great people, a collaboratively designed process and an environment of trust and collaboration. By treating the platform as a product, fostering psychological safety, and emphasising shared ownership, we built a system that not only worked but excelled.

While the specifics of our journey may not be universally applicable, I hope they provide some inspiration to move your team to use continuous deployment.

You can find out more about BJSS on our website, or get in touch here.