

enablis.

**Accelerating
Technology Delivery in**

Financial Services

**Practical lessons from teams balancing
speed, stability and compliance.**



Our Perspective

Technology delivery in financial services has changed significantly over the past decade. Modern platforms, tooling and ways of working make it possible to deliver change far more quickly than before. Yet many organisations still struggle to realise those benefits in practice, particularly where legacy systems, regulatory obligations and operational risk all intersect.

This whitepaper explores how teams are navigating that tension, and what has helped them deliver change at pace while maintaining confidence in stability, security and compliance.

The perspectives shared in this whitepaper are drawn from our experience working alongside technology and delivery teams across financial services. We're often engaged where organisations are under pressure to move faster, but cannot afford disruption to critical systems or loss of control.

We've produced this paper because we consistently see a gap between modern delivery theory and the realities teams face day to day. Legacy platforms, established governance models and organisational boundaries all shape what is possible and progress rarely follows a straight line.

The approaches described here are not intended as a prescribed blueprint. They reflect patterns that have proven effective in specific contexts and the practical trade-offs teams make as they evolve their ways of working over time. Our recommendations are shaped by what teams struggle with most often – ownership, change control, operational readiness and confidence in release – and by what has helped them make meaningful progress without increasing risk.

Problem Statement

Technology change can be delivered much faster than it used to be. Advances in tools, platforms, methodologies and ways of working have reduced typical lead times from months, and sometimes years, down to days and in some cases hours. Cloud platforms, automation, CI/CD, DevOps tooling and culture, and more recent advances in AI-driven development have all played a major role in this shift.

Despite this progress, some organisations still struggle to fully adopt these advancements. As a result, they miss out on cheaper delivery, faster time to market, and the ability to realise business value sooner. This is typically driven by a combination of factors, including:



Legacy technology stacks that are not compatible with modern methods.



Regulatory and compliance requirements that limit improvements.



Internal processes, governance and organisational structures built around older ways of working.



Skills gaps in the existing workforce, where employees have substantial experience in legacy technologies and tools, but limited exposure to newer tools and practices.

These challenges are common in the financial services industry. In many ways they are a sign of success. Only businesses that have grown substantially over decades tend to face the combined challenges of legacy, scale, and risk controls.

Modernisation is difficult, but not impossible. A number of financial services organisations have already taken on this transformation and seen the benefits of delivering technology change faster, cheaper and with less risk.

enablis.

The DevOps Model

One of the most popular approaches to emerge is DevOps. At its core, it is often summarised as



**you build it,
you run it"**

But what does that actually mean in practice?

Historically, building services and running services were treated as separate functions.

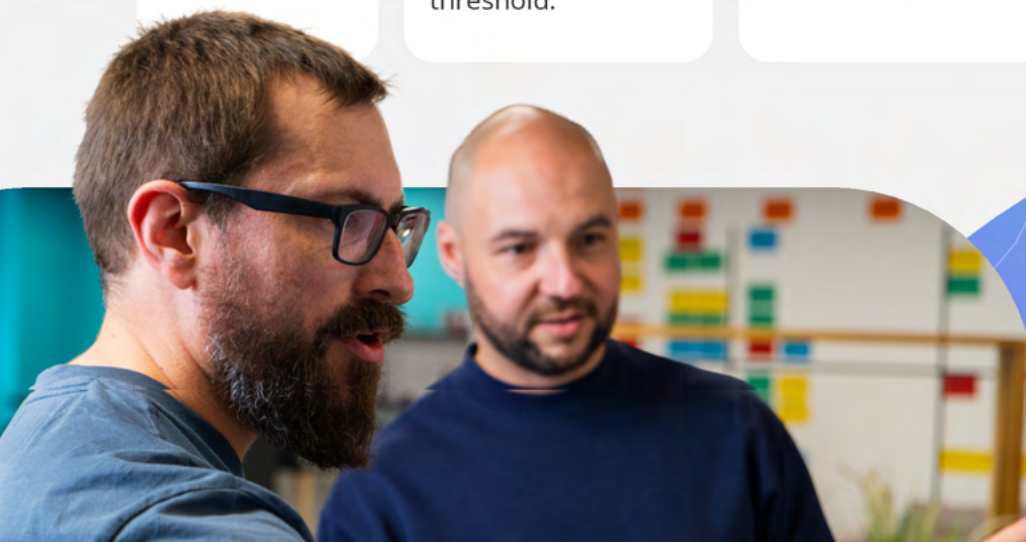
You would usually have one organisational structure and set of teams that ran, maintained, administered and supported live platforms, and another structure and set of teams focused on planned change. Those teams would gather requirements, work through roadmaps, and deliver projects and features into the live platforms on scheduled release cycles.

That split of responsibilities can work, but it usually comes at a cost. It can slow delivery, create handoffs, and institutionalise knowledge gaps between the teams. Over time, that tends to show up as lower quality and efficiency on both sides.

Traditional vs modern comparison

Area	Traditional separation of concerns	Why this creates friction	Modern DevOps optimisation
Incident investigation	Separate support teams investigate live incidents.	They have to familiarise themselves with the product, changes and code to identify the issue which takes time.	Development teams investigate live incidents. They know the product, changes and code so can pinpoint issues quickly.
Fixing live issues	Separate support teams produce fixes for issues identified.	They may miss context about why the implementation was done a certain way, introducing new issues.	Development teams fix issues. They understand why the implementation was done a certain way and are careful not to reintroduce issues spotted during development.

Area	Traditional separation of concerns	Why this creates friction	Modern DevOps optimisation
Code maintainability	Engineers produce feature code without ever having to read and understand it under pressure.	They're less aware of challenges like confusing algorithms, poor variable naming or inadequate comments/logging. This makes code harder to maintain.	Engineers have experience debugging under pressure so they write clearer, more maintainable code with simple algorithms, meaningful variable names, and useful comments.
Anticipating real-world issues	Engineers often lack hands-on experience with usability, performance and security pitfalls.	They write code that doesn't anticipate real-world problems.	Engineers are aware of usability, performance, and security pitfalls, so they code defensively and prevent similar issues from arising in new features.
Stability vs delivery	Project delivery teams under pressure to hit deadlines may prioritise releasing new features over the stability of the live platform.	Service teams often have the opposite outlook and apply a competing pressure, but are open to risks being underplayed by the delivery team.	The same teams are responsible for managing a stable service and delivering changes inline with roadmap ambitions. Being responsible for both stability and change forces careful judgment: if a change is risky, it's flagged; if it's safe and valuable, it goes live. This balance leads to better decision-making.
Risk threshold	Service teams, under pressure to run a stable platform, naturally have a low risk threshold.	This can sometimes make them blockers, slowing down feature development.	Delivery teams provide competing pressure with a higher risk tolerance, but are open to risks being overplayed by the service team.



Challenges

The benefits of the DevOps model are significant, and that's a big part of why so many organisations have invested heavily in moving in that direction. To put it into perspective, it's now very rare for a brand new technology delivery organisation to adopt a traditional operating model. In most cases they start with "DevOps by default".

So what stops organisations doing the same? Usually it's the friction that comes from existing structures and ways of working. Below are some of the common challenges that come up.

Challenge 1

Doesn't this just slow down already busy delivery teams?

It depends how the model is implemented. **You won't see the benefits if you:**



Expect a delivery team with an already busy workload to take on extra responsibilities without adding capacity.



Fold a service function directly into a delivery function under consolidated leadership without a clear plan for how skills gaps will be addressed.

A unified DevOps team covering both service and delivery should need less overall capacity than two separate functions, but only if the team has the right skills. In practice, it's usually easier to cross-train application developers into service support than it is to cross-train service support staff into application development. How you manage this transition is key.

Once a unified team is in place, you need mechanisms to balance workloads between the two remits. This matters because you don't want roadmap delivery to be blocked by the team being consumed with live support.

Techniques include:



Implementing a capacity allocation approach for service support, for example reducing sprint velocity proportionately or assigning points for BAU work.



Tracking the percentage of time spent between service and development, setting thresholds and adjusting them over time.



Incentivising the team to increase time spent on roadmap items over time by improving platform stability and making it more self-service capable.

This encourages innovation and pushes the team towards implementing efficient self-service and auto-recovery features.

The end state is a more efficient team and better service, but the journey to get there can't take shortcuts or you reduce the benefit.

Challenge 2

Does DevOps work in regulated industries?

Traditional methods can feel safer in regulated industries because they are slower and more document-heavy, but a lot of that is risk management theatre. Modern DevOps tooling can embed compliance and oversight directly into workflows, and it's often more reliable than manual gates and paperwork.

Examples of tool-led approaches that support regulatory safeguards include:

- Cohesive change management ecosystems with integrated audit history created automatically in tools. For example, Jira audit trails aligned to ITIL standards, linking to source control and CI/CD pipeline reports to show how a change was implemented, which checks it passed, and who approved it and when.
- Review and approval steps built into workflows, rather than relying on manual processes. **For example, Jira workflows that only allow a change to progress once reviewed by IT security.**
- Granular checks applied at the lowest level so important details don't get missed. Ticket-level approval ensures each change is assessed on its merits rather than getting lost in large documents covering a whole release.
- Monitoring and alerting linked into communication channels. Monitoring can be tied to Slack or Teams to bring real-time attention to issues and support rapid response.

None of this comes for free. It takes time and investment to build the right systems, but once in place, DevOps-compatible ecosystems are better equipped to meet regulatory needs while maintaining quality, stability and oversight.

Guiding themes



Auditability

Automated pipelines, workflow tooling (Jira, Azure DevOps) and Git commit history create comprehensive, tamper-evident audit trails that are often superior to manual change records.



Separation of concerns

Central teams act as reviewers rather than doers. Lower environments approve changes before production. Code review and approval workflows enforce four-eyes principles.



Security

Security checks are embedded in automated workflows rather than relying on manual gates. Dependency scanning, static analysis and infrastructure validation run on every change.



Evidence on demand

Feature flags enable precise tracking of what functionality was active for which customers and when, supporting compliance inquiries and incident investigations.

Challenge 3

Can DevOps work for organisations with legacy technology platforms?

It can be difficult to apply a DevOps model in ecosystems where legacy architecture complicates ownership. Lines between systems overlap, interdependencies are messy, and it becomes hard for any single development team to manage end-to-end responsibility.

There's no silver bullet, but it can be addressed through a structured approach:

- Take a modular approach, prioritising services or components that are easiest to decouple and have clear interfaces that mark ownership boundaries.
- Encourage roadmaps to design out obstacles in older areas of the ecosystem, for example introducing APIs between components so they can later be separated.
- Consider alternative ways to divide ownership, such as value stream or customer journey models that split the ecosystem along feature-led rather than technology-led lines.

For most substantial organisations, transformation is a multi-year journey involving sequencing teams, products and components. You can't do it all in one go. By starting with the most compatible areas first, you buy time to design out obstacles in the areas you'll tackle later.

Guiding themes



Design for failure

Architect systems and processes expecting things will go wrong. Recovery should be predictable, rehearsed, and achievable without heroics.



Shift left

Move testing, security checks and infrastructure validation as early as possible. Issues found earlier are cheaper and safer to fix.



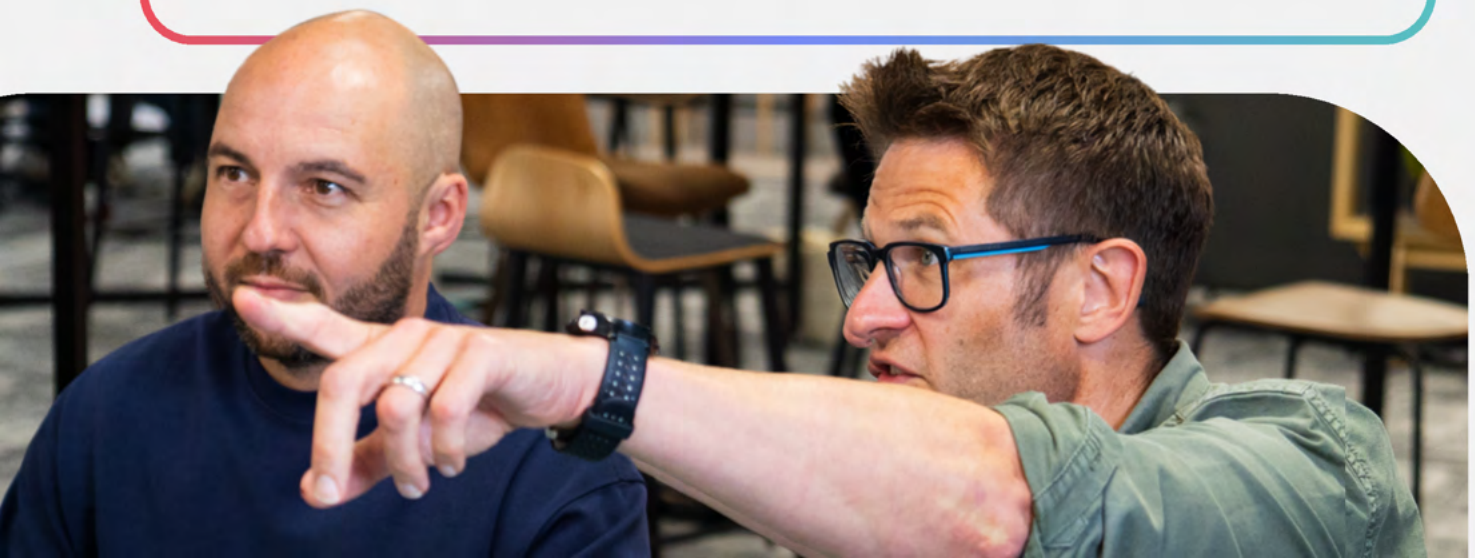
Automate for auditability

Replace manual processes with automated workflows that create immutable audit trails. This speeds delivery while strengthening compliance posture.



Release with confidence

Build release processes that are safe, reversible, and can be executed during business hours without downtime.



Delivering Transformational Change

Transforming the way a large organisation delivers technology is a major undertaking and shouldn't be treated lightly. The good news is that while the overall change takes time, you usually start to see benefits within months.

Below is a fairly textbook recipe for delivering this kind of change, with indicative timescales.

Principles



Ensure both business and technology leaders are invested and sponsor the change.



Understand this is not just a technology change, the business needs to change too.



Don't try to do it all at once. Deliver the change incrementally, in an agile way.

Approach

- 1** Select a single team or product that is best placed to spearhead the transformation. You need your best engaged people. Give them the tools they need, make sure there are perks like training and certifications, and ensure they are led by a partner who has done this journey before. This team becomes your north star.
- 2** Set expectations that this is a minimum 12-month journey for the pioneering team, and that it will take 3-6 months before they've made enough progress to share.
- 3** Document the journey. Start with value stream mapping to visualise flow and identify immediate bottlenecks. Take a scientific, hypothesis-driven, metric-led approach. For example, record baseline KPIs such as velocity, lead time to deliver and quality metrics. These may not improve straight away and can even dip early on, but by the 3-6 month mark you should start to see positive movement.
- 4** Aim for the pioneering team to do a show-and-tell to the rest of the organisation around the 3-6 month mark. By then they should have demonstrable progress and be able to explain tangible improvement backed by metrics. Use that moment to invite three further teams to volunteer for the next wave.

5 Don't bite off more than you can chew. Starting with a single team reduces the blast radius while pitfalls get worked out. After the pioneer team has reached the 3-6 month mark and delivered the show-and-tell, aim to mobilise roughly three teams per quarter to begin the same journey.

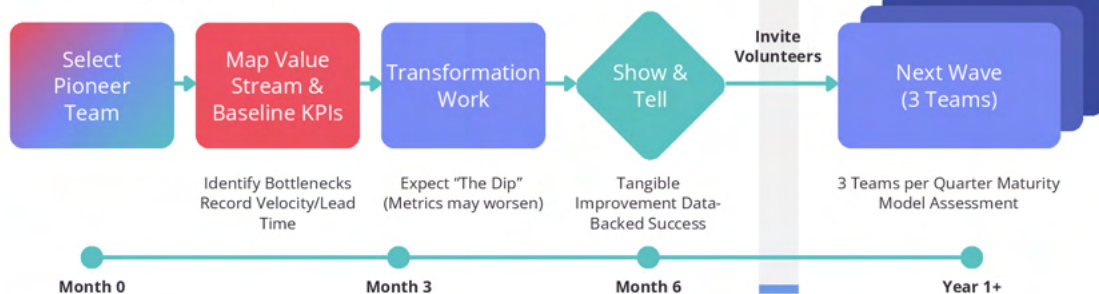
6 Try to prioritise teams that are best equipped to successfully transform. Starting with teams in a difficult position (legacy tech, capability gaps, etc) adds significant risk. It doesn't mean they won't transform, it just means it's better tackled once organisational maturity is higher.

7 Longer term, build a maturity model administered by a central team who own the model and assessment, while product teams take ownership and accountability for progressing through it. It should consider deployment times, deployment frequency, automation coverage, ability to adopt modern tools and processes, team skill level, and similar factors. Reward progress as teams move up through the levels.

Transformation Timeline & Flow

Phase 1: The Pioneer Team

"Minimise Blast Radius"



Operating Models

Whilst every business will have a unique context and starting point, there are some common patterns. The baseline is often separate service and delivery organisations, plus an array of product delivery teams supported by a few centralised functions. The remit of those central teams can play a critical role in enabling a DevOps operating model, but it can also hold it back if it is set up the wrong way.



Centralised Teams

The **"you build it, you run it"** ethos requires teams to be as free from dependencies as possible. If teams can't independently deliver change or resolve issues, and have to rely on central teams to facilitate either process, you are unlikely to achieve the pace and responsiveness the DevOps model should provide. Typical examples of areas which get centralised include platform and infrastructure, and quality assurance.

Central teams can also become bottlenecks. Demand for product delivery support is inherently difficult to forecast and capacity plan for, especially when one of the major benefits of the DevOps model is the speed at which teams can react and pivot to emergent priorities. This creates a dilemma. **Do we:**



Decentralise everything and run the risk of duplication and unnecessary spend on divergent tools, inconsistent standards, etc.



Or centralise the things we want consistent standards and approaches for, but accept the risk of slowing teams down because they are dependent on finite central resources.

Thankfully, there is a middle path out of this choice. The role of central teams should not be to do the work they are arbiters of, but to own the standards and provide supporting tools. A key tenet of a true DevOps culture should be:



**No central team dependency
should ever be on the critical path
for delivery."**

This does require product delivery teams to own the capabilities needed to do the work themselves, and that can be challenging. For example, software developers who have spent most of their career on legacy systems are unlikely to be confident configuring cloud infrastructure.

If the end state is a product team that can self-serve all aspects independently, that may take some time to achieve. But there are several stages of maturity between fully centralised services and that ideal end state.



Stages of Maturity

Baseline

Stage
1

- Centralised teams
- Siloed responsibilities
- Single points of failure
- Waterfall processes and gate delivery required to manage capacity
- Loss of autonomy and agility of product development teams
- Capacity planning difficult
- Clear segregation of responsibilities but at the cost of efficient delivery

Tactical

Stage
2

- Secondment of centralised resources into product development teams
- Removal of silos
- Improves autonomy and agility of product development teams
- Skills gap starts to be addressed as embedded practitioners upskill colleagues
- Single points of failure persist and capacity planning even more difficult

Mature

Stage
3

- Re-organised into stream-aligned teams based around end service or capability
- All product development team members have some distributed infra/devops capability
- No single points of failure
- Eases capacity planning challenges
- Central teams own standards and tooling to help accelerate delivery teams e.g. pipelines, ephemeral environments

Elite

Stage
4

- Entirely self sufficient product development teams
- T-shaped engineers with cloud/QA capability as standard
- The existence of DevOps maturity models with teams empowered and incentivised to drive progress through
- Community contribution back to central tools and standards

If you find yourself at stage 1, it is not possible to jump straight to stage 4, and trying to do so usually introduces problems and leads to a loss of confidence in the model. Instead, maturity needs to be nurtured as a gradual process, with senior sponsorship and a shared understanding that the end state will involve difficult steps along the way.

Whilst it is never easy to advocate acceptance of things going wrong, and a degree of accountability does need to remain, it is important that teams are encouraged on a journey which will require people to leave their comfort zone. That is unavoidable.

We may now look at each of the steps in more detail.

Stage 1 to 2: Establishing Foundations

Moving from centralised, gated delivery to embedded capability and basic automation.

✓ Focus on teams and people who are ripe for change.

✓ Focus on teams with technology that is conducive to modern methods.

✓ Second centralised capacity into product development teams.

✓ Set the remit of seconded capacity to:
a) ensure the work required gets completed, whilst also b) cross-skilling into the rest of the team.

✓ Be transparent about the goals so the rest of the team learns from the seconded resource.

✓ Provide training to accelerate growth and demonstrate investment in the vision.

✓ Ensure the team sees this as an opportunity to learn, develop and broaden skills.

Architecture and Documentation

Ensure architecture is sufficiently documented before development begins. Technical, security and release stakeholders should be bought-in and confident in the approach. This early alignment prevents costly rework and builds trust, which is needed for faster release cadences later.

Define an outline release approach covering production staging, hidden production, beta and full release. Agree upfront what compliance evidence is required at each stage, including non-functional testing and security assessments.

Development Process Fundamentals

Adopt a continuous delivery model using short-lived feature branches committed frequently to main. Avoid long-running branches which create merge difficulties and synchronisation problems. Larger features should be assembled incrementally across multiple stories, enabled only when all components are ready.

Maintain the main branch at production release quality at all times. Every commit to main should be deployable. This requires discipline around code review, automated testing, and ensuring incomplete features are disabled via feature flags rather than left half-built in the codebase.

Implement feature flags to control functionality rollout. Flags are maintained as application configuration, but can be updated on-demand via API, changing application behaviour in near real-time. This enables safe staging of functionality before customer release and supports rapid rollback if issues emerge.

Environment Foundations

Establish a standard set of non-production environments designed for specific purposes:



Dev

Typically local, for individual developer work



Test

One per delivery stream, for integration testing



Staging

Combined environment for release candidates and non-functional testing

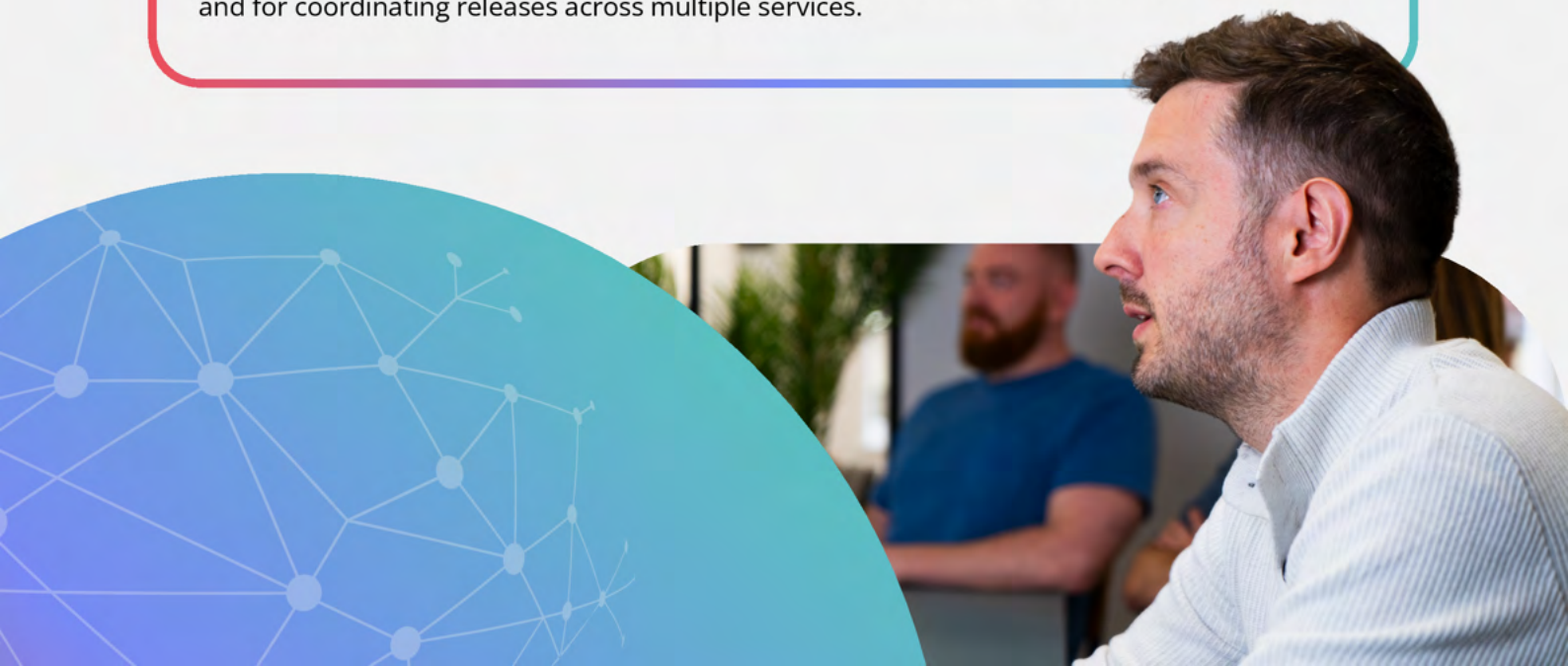
All environments must be controlled using Infrastructure as Code. Core infrastructure such as load balancers and database clusters should be validated as far left as possible. Enforce a no surprises culture. Avoid any production-only configuration or infrastructure that cannot be tested elsewhere.

Establish clear rules of engagement between teams regarding environment usage, including ownership, coordination and data reset expectations. Where multiple teams or services need to integrate, define integration points early and ensure dependent services are available in test environments from the outset. This prevents late-stage integration surprises and builds confidence in cross-team dependencies.

Basic Quality Gates

Implement automated code quality checks including linting, static analysis and dependency vulnerability scanning. Use a secure artefact storage solution such as Artifactory to store created artefacts with cryptographic proof, and to control shared dependencies.

Apply semantic versioning consistently to all built artefacts. This creates a clear, traceable link between source code, built components and deployed releases. This is essential for audit trails and for coordinating releases across multiple services.



Stage 2 to 3: Building Maturity

Distributed capability, elimination of single points of failure, and production-like practices everywhere.

✓ Use lower environments to give newly trained decentralised teams a safe practice space.

✓ Consider a license-based model where teams earn (and can lose) the right to do daily releases for pre-approved change types.

✓ Evolve workflows to capture review steps properly.

✓ Apply four-eyes policies on change approval to maintain separation of concerns.

✓ Use centralised code review processes for specialist areas, e.g. security.

✓ Make critical use of versioning and auditable tooling.

Testing Strategy

Alongside unit and integration testing, implement API-level system tests designed to run in test environments for rapid validation of application behaviour. Where a user interface exists, implement a small number of end-to-end journey tests. Keep these minimal, as they are slower to execute and maintain.

Ensure representative local development environments. Engineers should be able to start required services locally and test against them, reducing dependency on shared environments.

Establish a controlled, ideally synthetic, test data environment. Test cases can be added as required, with data reset from scripts on a regular basis (overnight and on-demand). This avoids reliance on special manual data setups and enforces data management discipline.

Release Process Maturity

Design the release process so it can be executed safely during business hours without requiring downtime. If issues are discovered, use a well-tested rollback process.

Adopt canary releasing. Deploy changes to a small proportion of traffic initially, monitor closely, then ramp up in increments. This limits blast radius and provides early warning of problems.

Release little and often. For some systems this means weekly, for others multiple times per day. The goal is to avoid accumulating large change deltas which carry greater risk and are harder to troubleshoot.

Implement strict backwards compatibility across APIs, particularly where supporting native mobile clients and third-party integrations. Adding new attributes is non-breaking, removal or semantic changes require versioning. Use a simple URI path parameter versioning scheme.

CI/CD Pipeline Enhancement

Build representative smoke testing into deployment pipelines to verify environments immediately following release. Replicate the same release process across non-production environments, including releasing under moderate load, to rehearse production deployment.

Implement production-like observability and monitoring in non-production. When issues occur, engineers should use the same troubleshooting approach and toolchain they would use in production.

Repository Strategy

Carefully consider mono versus multi-repo approaches. Monorepos can suffer from merge contention (resolved using merge queues) and can obscure what components have changed. However, they simplify creating consistent point-in-time test environments based on a single semantic version tag.

With multi-repo approaches, use a release manifest to collate versioned components. Each component should follow semantic versioning conventions: major versions for breaking changes, minor for new functionality, and patch for fixes. This enables release managers to reason about compatibility and risk when assembling releases, and provides clearer signals to consumers of shared libraries or APIs. Avoid mixing component versions across releases.

For hotfixes, create a branch from the relevant semantic version tag, merge changes, and create a new patch version tag. This avoids maintaining long-running release branches.

Where using a monorepo with multiple team ownership, implement CODEOWNERS to require review and approval from specific owning teams.



Stage 3 to 4: Achieving Excellence

Self-sufficient teams, continuous improvement, and contribution back to standards.



Teams own the maturity model for their area.



Organisation acts as a beacon for DevOps best practice.



Evolution of central tools is driven in an open-source style model.



Encourage outbound contributions: talks, blogs, etc.



Communities of practice become governors of standards.

Recruitment of high calibre technologists is elevated through the appeal of the proposition.

Governance at Speed

Establish a release license approach for high-frequency releasing. Traditional CAB processes become onerous when releasing daily. A release license defines pre-approved release criteria. Standard releases are submitted to CAB for scheduling and notification only. Formal CAB approval is reserved for changes that break the license terms: significant customer journey changes, breaking API changes, or introduction of new systems.

Implement automated change documentation. Use Git commit history to cross-reference change tickets and compile release notes containing details from the ticketing system. This creates an immutable audit trail linking every production change to its authorisation.

Periodically audit the change process: inspect CAB records, cross-reference to tickets, review requirements traceability, merge approvals and test coverage. This demonstrates control to regulators without slowing delivery.

Operational Excellence

Design out synchronised changes across dependent teams. Infrastructure and database changes should be automated as part of the release process using Infrastructure as Code and automated migrations. Eliminate manual dependent changes entirely.

Following production release, block the staging environment temporarily to support investigation of production issues that need replication.

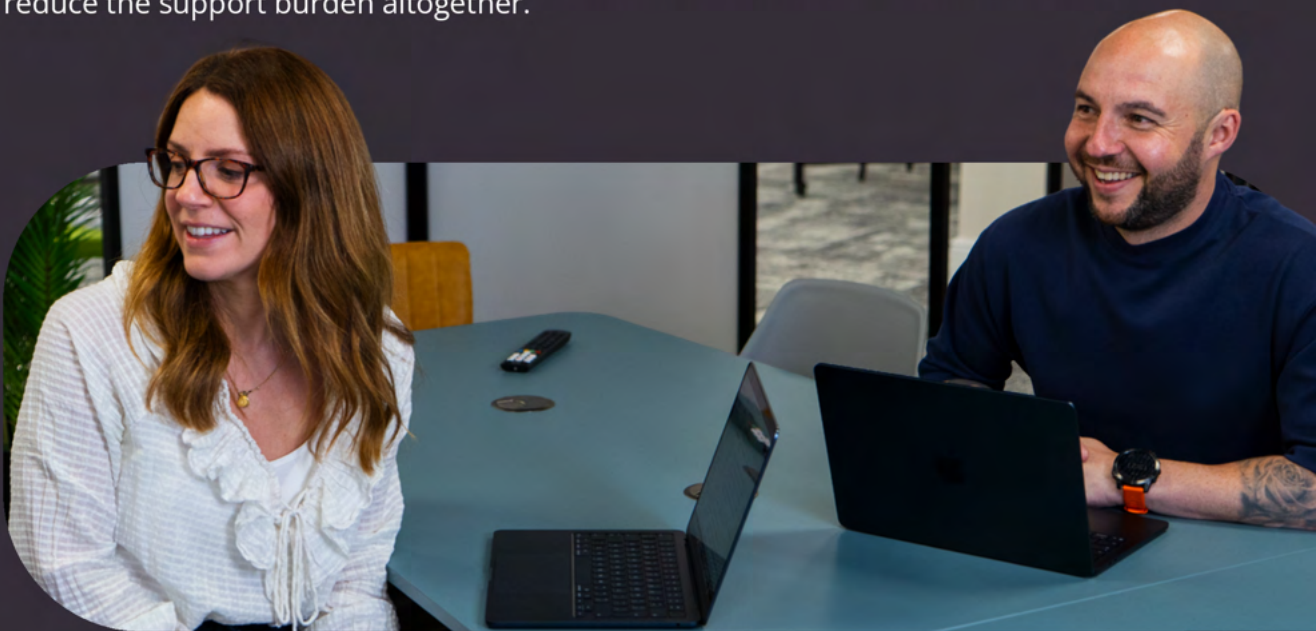
Regularly demonstrate features during pre-production phases. Showcase non-functional capabilities such as system recovery and security measures to build stakeholder confidence.



“you build it,
you run it”



The teams that build systems should also live with them in production. When engineers are responsible for keeping what they build running, design decisions change for the better and code becomes easier to understand, maintain, and fix. Planning for this means being honest about capacity—factoring BAU and operational work into delivery expectations, and paying attention to how much effort goes into support versus new development. Over time, teams start to adapt naturally, investing in better observability, self-recovering systems, and smarter ways of working that reduce the support burden altogether.



Case Studies

enablis.

Spotlight on:

A UK Bank

Context



A UK retail bank was informed by its incumbent provider that hosting for its Business Intelligence platform would be withdrawn. This triggered a major transformation programme. The organisation had never previously operated its own end-to-end banking platform, so this was not simply an infrastructure migration. It required fundamental changes to technology, operating models, and ways of working.

Rather than treating this as a forced move, the bank used it as an opportunity to modernise its platform, begin a cloud journey, and explore how a DevOps way of working could be introduced alongside wider business change.

Challenges



There were plenty of reasons to think a cloud and DevOps approach wouldn't work. The system was built around monolithic legacy applications from third-party providers. From an organisational point of view, they didn't have processes designed to cope with the idea of multiple releases per day. Key quality checks from IT Security and Infosec were baked into gated processes, with approvals granted at large, infrequent meetings. A lot had to change: the technology, the tools, delivery processes, and the way operational business processes enabled and supported change.

Approach



The first step was identifying a single platform that could act as the spearhead for change, including wider business change, and finding a partner who could bring cloud and DevOps approaches into the programme.

Legacy technology challenges, while not ideal, could be abstracted and built around. The new AWS platform was built using Terraformed Infrastructure as Code (IaC). The same scripts were used to build sandbox, test, and eventually production environments, which later enabled an ephemeral development environment capability.

Cloud-native services such as Control Tower were introduced to support regulatory governance and oversight.

Change control also had to evolve. Dynamic workflows were introduced to support real-time, in-flight review and approvals, bringing stakeholder departments into the process rather than sitting outside it. Monitoring platforms were built with security, service operations, and data insights teams acting as key stakeholders. Physical office space was adapted to embed dashboards and provide clear visibility, so critical alerts could be seen quickly and trigger action immediately.

Supporting departments gradually adapted their own processes to work with a faster delivery pace. This didn't just make them ready for other systems to follow the same route, it also helped them actively encourage it because it made parts of their roles easier.

Once launched, the team adopted a "you build it, you run it" model. This combined scrum-based feature delivery with a more reactive Kanban-style approach for allocating capacity to BAU service support. Metrics were captured to understand how much time was being spent on different types of support activity. That data then supported decisions on feature development by showing where it would pay back. For example, building self-serve password reset tools reduced administrative burden in a measurable and predictable way.

The objective was to continually reduce the resources required to run the platform, while still adding new features at pace.

Outcomes



The new platform launched successfully and became the north star for teams across the business. With processes tailored to support more granular, real-time change management, central teams such as IT Security, Infosec, Change, and Service Management became advocates for other teams and systems to adopt similar methods.

All new products were built along the same lines, with business stakeholders benefiting from quicker, smoother, and more predictable delivery into live. In the year that followed, they delivered PSD2 APIs and launched a new mobile app, all while adhering to DevOps principles.



Spotlight on:

A UK Building Society

Context



A UK building society required a new native mobile banking application for its savings customers. The solution included a modern API layer integrated with legacy core banking platforms and existing enterprise service interfaces where appropriate.

The programme carried significant regulatory obligations due to Open Banking requirements, including Strong Customer Authentication. It was also intended to act as a pilot for modern engineering practices such as DevOps, CI/CD, and containerised deployments.

Challenges



The project had to bridge modern mobile development with established legacy infrastructure, while meeting non-negotiable regulatory and compliance requirements.

The organisation was traditionally cautious about technology change. This was shaped by historic costs of change, and concerns around knowledge transfer and long-term maintainability. Stakeholders needed confidence that modern delivery approaches could operate inside existing governance frameworks before fully committing.

Approach



An internal alpha phase was used to prove the core technology choices and secure stakeholder buy-in before scaling the team.

Security was designed around a zero-trust model, using JWT tokens and device-signed API requests, aligned with PSD2 and SCA requirements.

The team adopted a single main branch model with feature flags. This enabled weekly releases, while still allowing new functionality to be staged safely in production for internal validation. Test automation was comprehensive across unit, integration, and API levels, with smoke tests run after every release. Test environments were reset daily to a known state, removing reliance on special data fixtures.

Infrastructure as Code enabled repeatable CI/CD deployments to Kubernetes across all environments. Guard rails were put in place so production releases could only be performed by a dedicated release manager role. Database schema changes were managed through automated Flyway migrations.

A release licence model enabled regular approved releases, with CAB reserved only for significant functional changes. Observability metrics and structured logging were built in from the start, enabling a “you build it, you run it” hypercare model.

Success depended on wide engagement across architecture, operations, change, and business teams, not just the delivery team.



Outcomes

The project delivered a secure mobile banking experience and demonstrated that accelerated delivery and modern infrastructure practices could operate effectively within a risk-conscious building society environment. The patterns and controls established also provided a practical template for subsequent digital initiatives.

Track Record

The Enablis portfolio of current clients includes:



People & Experience

enablis.



Peter Cotton

Chief Customer Officer

Twenty years in technology delivery, from engineering through architecture to running major projects and transformations. Peter has worked across multiple industries, including a period in Financial Services where he led the Sainsbury's Bank DevOps transformation, alongside projects to fulfil PSD2 requirements and launch new mobile apps.

Ed Marshall

Chief Technology Officer

Extensive financial services experience across Sainsbury's Bank, Tesco Bank, Tandem Bank, Experian and Starling Bank. Ed was the technology lead for Sainsbury's Bank's DevOps transformation, driving the cloud architecture and integrated toolset that connected stakeholder teams to technology delivery.



Simon Roberts

Technology Delivery Director

Has driven major technology change across a number of building societies including Yorkshire Building Society (YBS) and Leeds Building Society (LBS). Simon led the YBS PSD2 and mobile projects and drove adoption of a DevOps methodology. He also led architecture on the initial phases of the LBS core banking replacement.



Contact Us

Enablis is here to help you innovate faster, build stronger, and achieve more – without the fluff, just results.

 hello@enablis.co

 LinkedIn

 1st Floor, 1 York Place,
Leeds LS1 2DR

 +44 (0)113 323 9900

