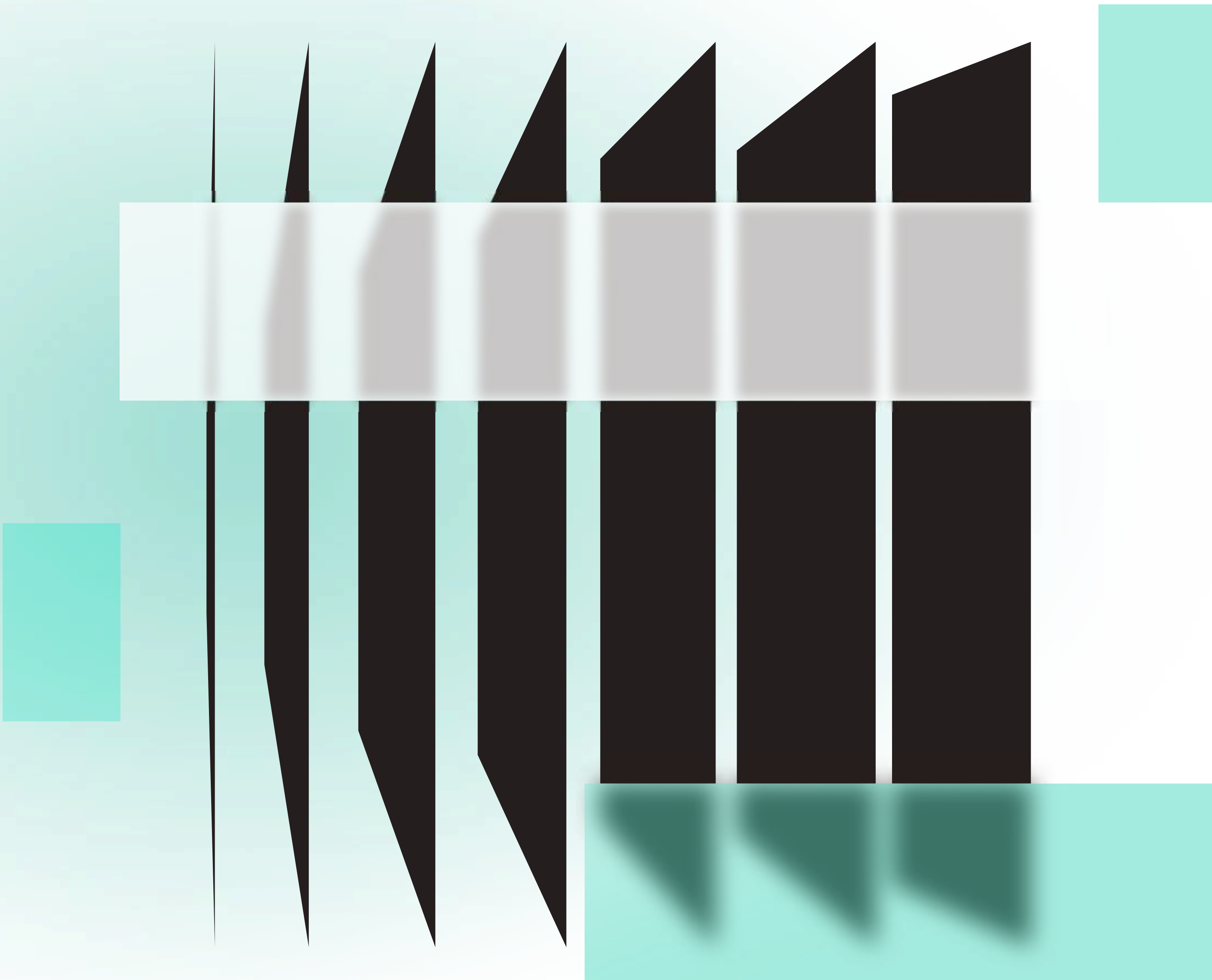
SCENARIO BOOK

Progressive Delivery

Accelerating the flow of value to customers requires the frictionless delivery of software in the form of infrastructure, code and data. Where are your sticking points?



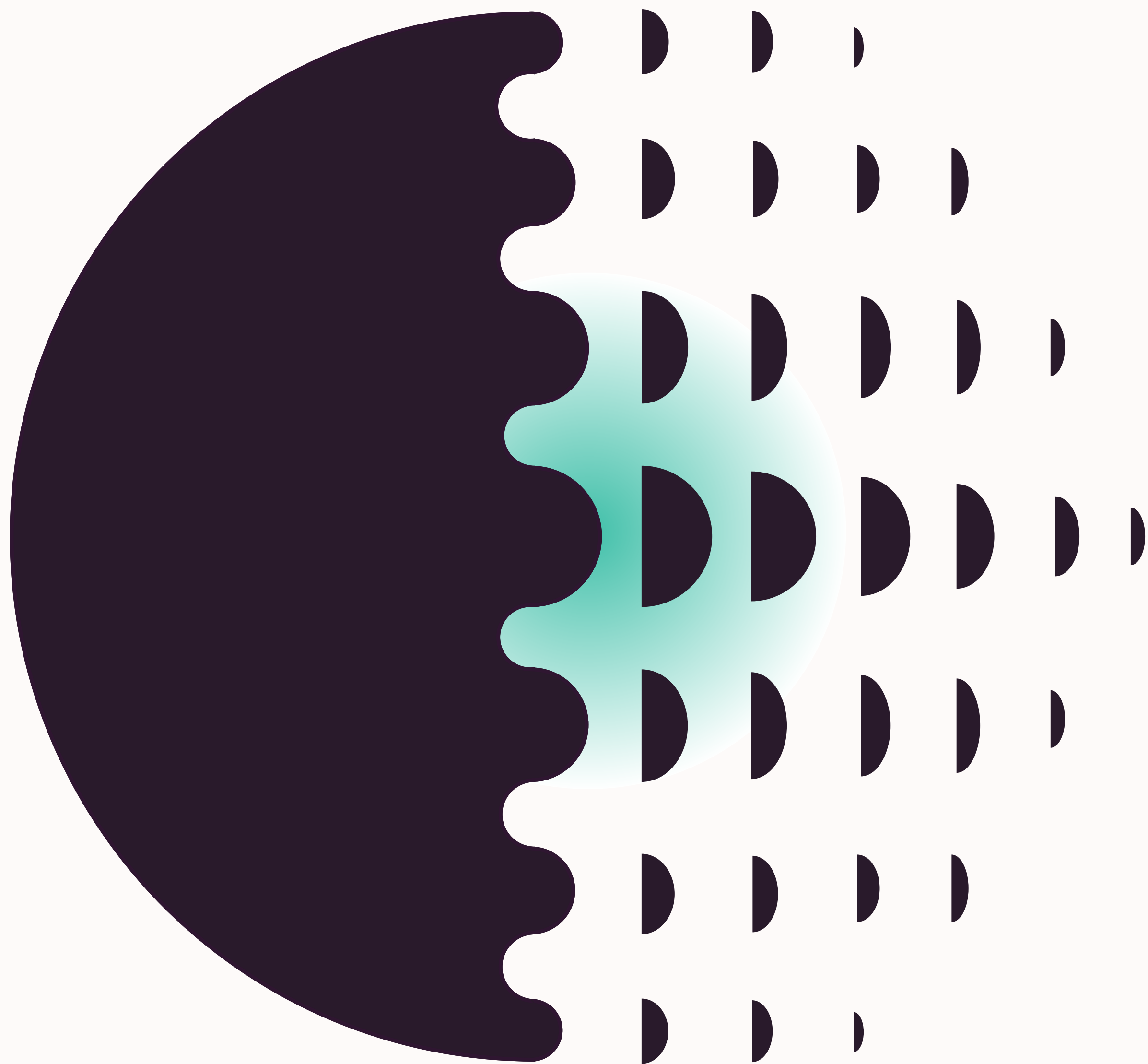
HYPR

Your scenario?

You believe that your technical delivery practices are constraining the flow of customer value because:

- You're not releasing software at will
- Releases are too large
- You're not confident that releases can be done safely and without downtime or disruption to your customers
- You're storing up changes until there is enough confidence to deploy but this creates a significant cost of delay
- You know that work on testing, observability, monitoring, security, compliance and analytics are still considered at the end of the delivery process, rather than truly being a part of it
- Your teams are becoming frustrated and demotivated by the interruption to flow
- It takes a long time to find and recover from problems
- You can't turn features on and off or release to small, targeted customer segments
- You don't have data about how your customers use your software product features after they are released

"There are times when it feels like we are paralysed, unable to release what we want, when we want. We need to remove the constraints urgently and reach a new level of maturity"



What’s happening and why?

The inability to release at will is the result of historical constraints in your delivery system. Enterprises haven’t adapted the system quickly enough to take advantage of new technology and practices. We see the following patterns in our work:

Legacy technology

Monolithic architecture makes individual components difficult to release separately. So the release you want to make to improve value for one component is intertwined with – and constrained by – other parts of the architecture. This lack of separation dramatically slows down the flow of value to targeted customer segments.

Big bangs, sporadic flow, bigger risk

Issues with legacy tech means that releases are necessarily much bigger than they need to be and so happen less often. There’s no continuous flow of value with ‘big bang’ releases.

Big deployments are also much more complex, come with a higher risk of failure and have a greater impact on the operational part of the business. Manual intervention in deployment can be a recipe for a fat-fingered disaster.

Practices and processes

Speed is continuously prioritised over engineering discipline and this may result in operational and quality practices coming late in delivery or being ignored. For example:

- Containerised development workflow not used
- Infrastructure as Code not used to automate infrastructure
- Lack of layered test automation or poor quality tests
- Security and compliance checks not automated
- Inefficient code branching and merging strategy
- Feature flag management not in place to separate deployment from release
- Lack of DevOps thinking and pipeline management

People and capability

We find that organisations generally have too few trusted engineers with the right Progressive Delivery capabilities and access to production. This creates a bottleneck and increases risk if certain processes are done manually.

What is Progressive Delivery?

CI/CD augmented

Continuous Integration (CI) and Continuous Delivery (CD) are well-known terms but Progressive Delivery adds a dimension of risk control and feedback. It allows you to release different elements and features of your product to defined audiences.

Rather than features being deployed dark and turned on as a whole, Progressive Delivery enables the delivery of features to groups, regions or other demographics to assess fitness-for-purpose, usage etc.

There is a growing trend towards testing value hypotheses in production. So the aim is faster delivery into production with the ability to do more testing in production. Using Progressive Delivery techniques, this is much more achievable.

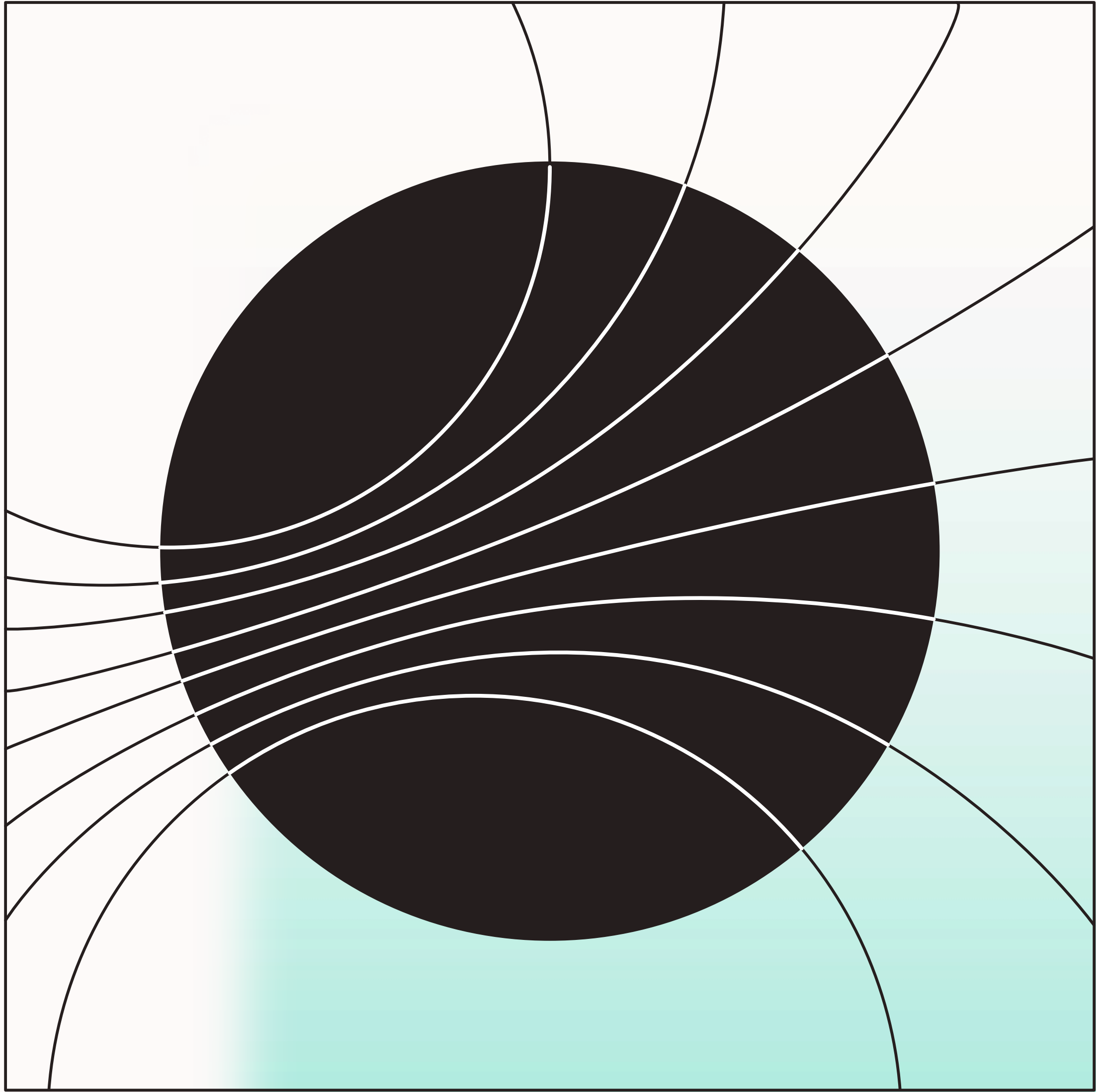
Delivery to production is a technical concern while releases to customers is a business concern. Product managers can be empowered by having the tools to enable and disable features by location or audience demographics.

What does good look like?

Progressive Delivery should allow you to frictionlessly release code to production when you want. It may not be every minute, but it's hard to argue that you shouldn't have the option. You should also be able to release only what you need to in order to meet your goals with different customers.

Getting to this point is not a magic leap, but it does require systemic thinking across several dimensions of software delivery. These include:

- Containerised development workflows that allow all team members to learn and experiment with fast feedback
- The ability to create and evolve cloud infrastructure through infrastructure as code from containerised local development through to production
- Architecture with appropriate patterns including separation of concerns and a clear expression of domains from a business perspective
- Independently releasable services/components aligned to domains
- Leveraging a layered approach to test automation where quality and feedback is built in as early as possible
- The notion of quality is extended to include the automation of security, compliance, audit and documentation
- Evolvable delivery pipelines including templates to encourage consistency and reduce cognitive load on teams allowing them to focus on customer value
- Appropriate gates based on organisational context to support governance requirements
- Feature flag management to allow teams to deploy to all environments with decoupled controls on release. Release can be by feature to a defined audience of customers. This final capability being the essence of Progressive Delivery
- Use of DORA or flow metrics to guide improvement



We can help you get there

We're experts in helping enterprises mature their delivery practices to accelerate the flow of customer value. Through our work, we have developed the patterns and approaches that help us understand what good looks like in your context and the roadmap to get you there.

Discovering your context

We use Situational Analysis to discover, among other things:

- The maturity level of delivery – your ability to release code into production, from software design to customer use
- The key constraints in delivery
- Your technology stack and toolset
- Your code, data and quality practices

We use our Improvement Model to baseline your maturity against benchmarks. You'll see the gaps that need addressing.

Starting your journey

The practice gaps you should focus on closing might include, for example:

- Containerising local development
- Improving layered test automation
- Automating infrastructure, deployment, testing, compliance and security in pipelines
- Decoupling deployment from release with feature flags
- Helping platform teams create valuable reusable artefacts that product teams love to use
- Progressively delivering features to test with the right audiences
- Modernising architecture

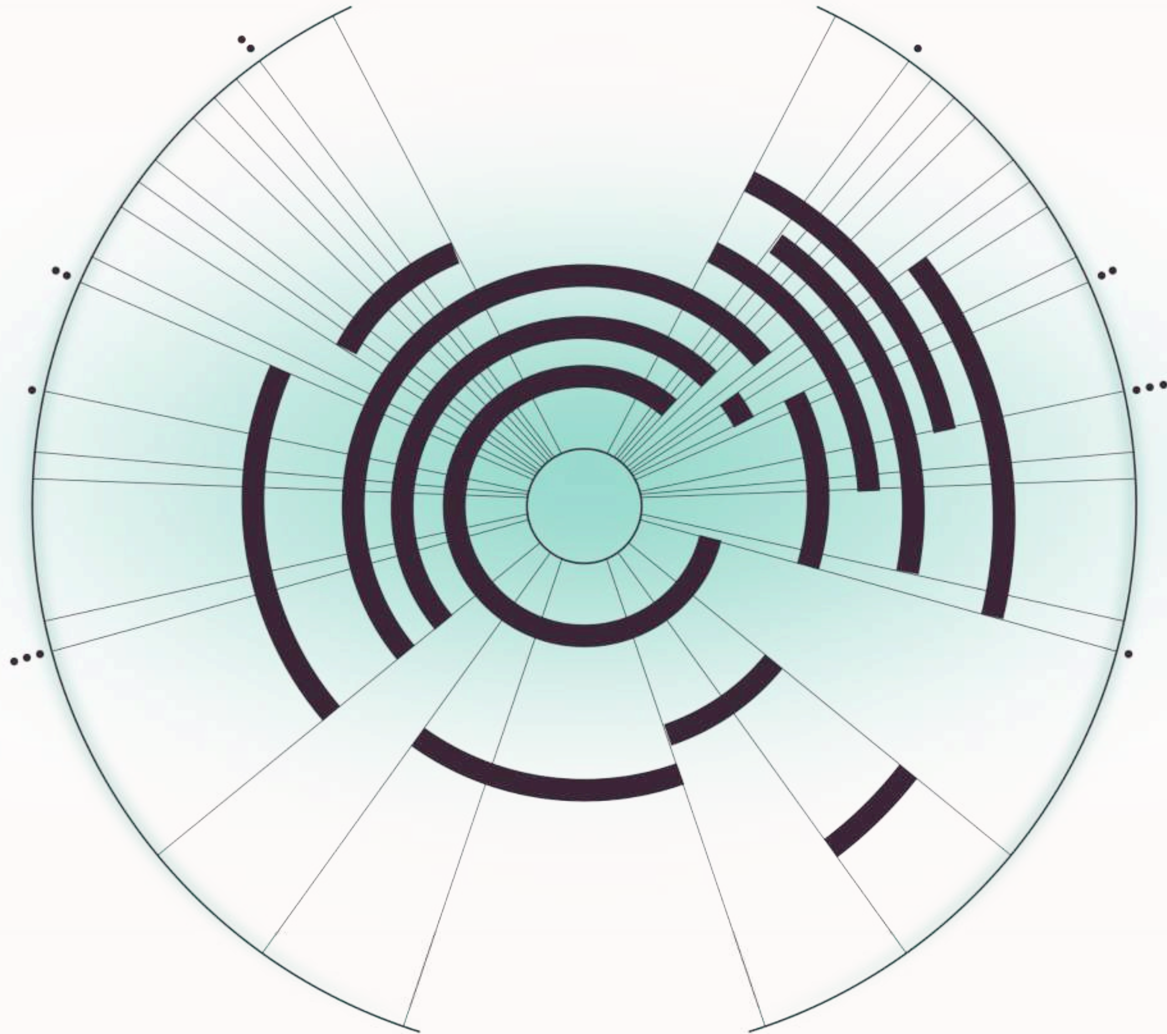
Hybrid Teams – improving while delivering

We aim to improve the appropriate areas in stages through our 'Hybrid Team' model. We provide a small team of experts to work alongside your people on a real piece of work. We introduce the disciplines that improve delivery and, at the same time, give you the capability uplift that can then be taken across other teams. It works incredibly well. For the full benefits of this approach, see our dedicated Hybrid Teams Scenario Book.

Next steps

We'd love to help you mature your delivery practices. Every engagement is different and we can work together to create a Hybrid Team model that works in your context. Don't hesitate to call us to explore this further.

The Appendix at the end of this Scenario Book provides delivery practitioners with further detail on how we think about the key Progressive Delivery practices



Why HYPR?

We can help you keep you on the right side of technology change and make the decisions that ensure your system accelerates the flow of customer value. Call us now...

What makes us different?

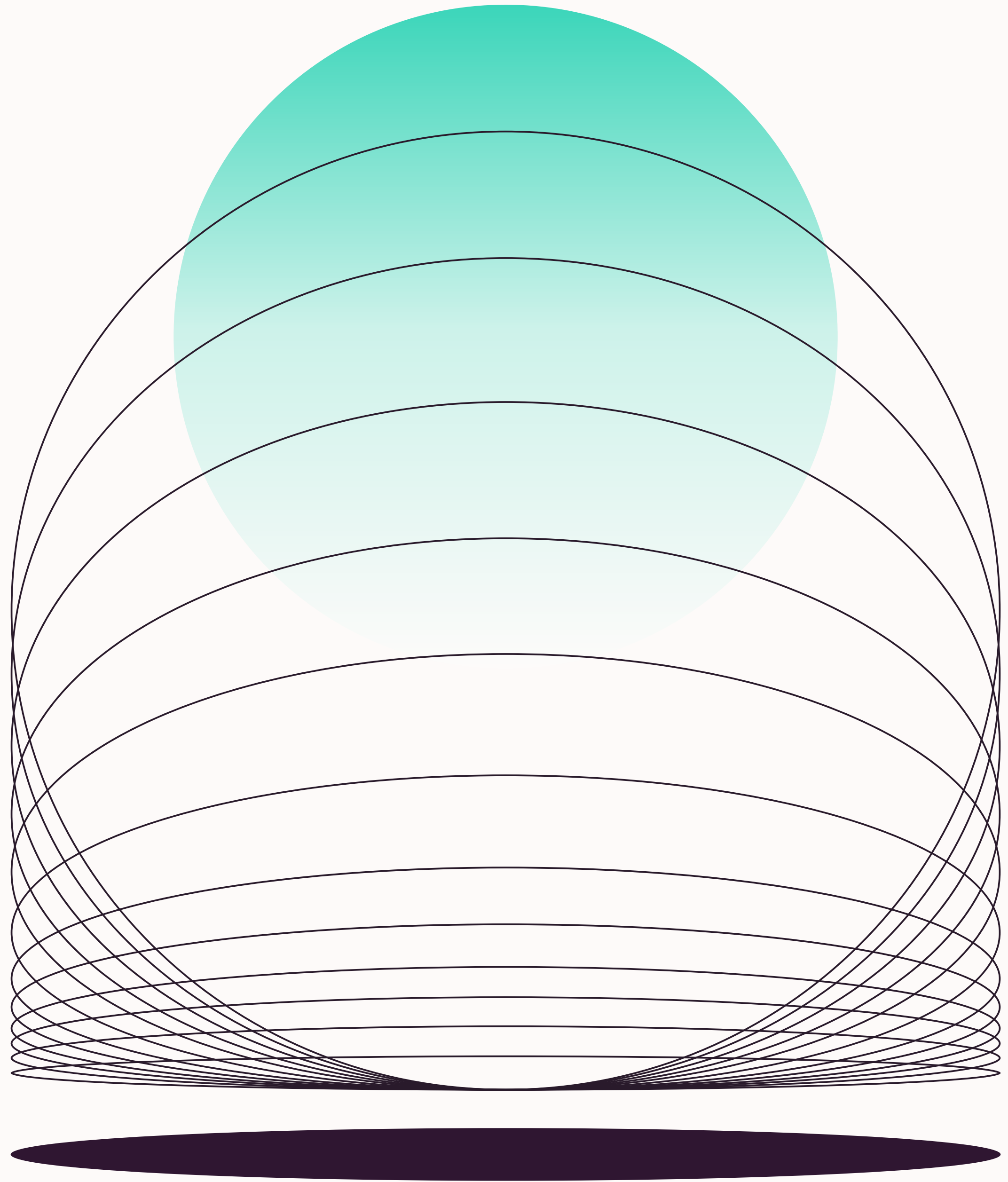
Focus on flow – Progressive enterprises are focusing on finding and removing delays from their system through the practice of Value Stream Management (VSM). We're a leading VSM consultancy helping enterprises in NZ and Australia.

Systems thinking – We take a systems-thinking approach to avoid local optimisations that contribute little to the whole.

Focus on your people – Technology and people are one system and two sides of the same coin. We focus as much on the social constructs and human networks as we do on the tech.

Transition not transformation – Your enterprise operates in a VUCA (Volatile, Uncertain, Complex, Ambiguous) world. It needs to keep flying while making changes. We know from experience that transition is the only way you can do both.

Our people – We're a diverse team with shared purpose and values. We have extensive skills across our consulting lines, from the very best software engineers to strategic experts able to engage at board level. They have lived at the coalface of change.



More about Progressive Delivery practices

We aim to improve your practices and help your teams develop new skills across every aspect of delivery. Here's more detail on how we think about some of them...

DevOps (in general)

DevOps is a 'culture' best defined by observable behaviours, including psychological safety and working more collaboratively. DevOps shifts teams away from departmental project thinking to thinking about flow. It removes the divide between development and operations. A mature DevOps practice reduces tension between delivery throughput and system stability through automation and knowledge sharing. Benefits from implementing DevOps include:

- **Creation of a continuous learning and improvement culture**
- **Higher frequency and quality of deployments**
- **Greater innovation and risk-taking ability through safe-to-fail experimentation**
- **Better ability to identify the approaches that deliver faster time-to-market**
- **Reduced lead time for fixes**
- **Greater understanding of severity and frequency of release failures**
- **Improved Mean Time to Recovery (MTTR)**
- **Reduced key person risks and bottlenecks through decentralising control**

Creation of resilient, self-healing practices

To operate effectively at scale, software is typically distributed. Distributed systems provide many benefits but also unique challenges. Resilience can be thought of in two dimensions – technology/engineering resilience (maintaining the efficiency of function) and ecological resilience (maintaining the existence of function). Creating resilient, self-healing systems requires consideration of the following:

- **Understanding the highest-value improvements that can be made**
- **Helping technologists understand that distributed systems are often non-deterministic. This helps create the right culture for change**
- **Helping to improve the flow of information between development and operations**
- **Understanding the role of architectural patterns in system resilience**
- **Understanding that complexity is the enemy of resilience**
- **Establishing Chaos Engineering Initiatives to help change behaviours towards hypothesis-based experimentation**
- **Leveraging cloud technologies to eliminate single point of failures and fast recovery from any incident**

Automating delivery pipelines

Continuous Delivery pipelines automate infrastructure, testing, deployment and release. A good practice here is to treat the platform itself as a product for which customers are the development teams. A product platform must be easy to use, easy to maintain and provide the fast, visible feedback that teams need to deliver, but also the flexibility to support diversity in teams.

Using multiple pipelines – typically one per software component – also allows faster rollout of updated versions of each software component, rather than having to roll out the entire product as one single unit. Pipelines involve a lot more than just building and testing code in the modern development stack. They now need to incorporate security, malware and governance checks and move to continuous security, malware and governance postures, with additional focus on fit-for-purpose. Pipelines should provide the confidence that the product is secure, meets regulatory requirements and that features work as expected. They should also include proactive, automated updating of third-party libraries for which new security patches are available.

Layered Test Automation is essential in an effective pipeline. As with all technical practices, if not adopted properly, they can be ineffective and expensive. Experience helps guide a team to the best approach in their context, taking into consideration technology stack, current maturity, programming languages, software architecture, tools and delivery demands. See the more detailed section below on **Layered Test Automation**.

'Shifting left' should also be encouraged. Engineers then own the pipeline from source code to production and keep the pipeline fast by removing or re-evaluating long-running processes. There are no 'snowflake' environments because all environments are the same. Creating environments that are as similar as possible as production, cost-permitting, allows teams to practise and identify problems before deployment.

While delivery is continuous, release should be **'on-demand'**. This can be accomplished by using appropriate system design to make changes available at the right time and/or incrementally to assess overall system impact. Potential techniques may include feature toggles and similar, as well as authorisation based on policies or activities.

These pipeline practices help teams know that features work today, tomorrow and in a year's time. Improved feedback/fast feedback cycles promotes fast learning and gives teams confidence to evolve and refactor mercilessly, drastically cutting development time. Delivery to production can be reduced from months to days or even shorter.

Continuous Integration and Continuous Delivery (CI/CD)

Continuous Delivery encourages teams to build software that can be deployed at any time. Deployment of code need not be continuous but practices in this domain should provide confidence that code that is built and packaged can be deployed safely and reliably.

Central to Continuous Delivery is a delivery pipeline through which all code is sent. This pipeline incorporates commit, acceptance and production stages. As teams mature, both application code and infrastructure utilise pipelines to support the Build Quality in practice.

There is a separation between delivery and release, the release being when a feature or features are actually provided to users. Separating these concerns helps identify where modern practices can be successfully adopted in order to simplify delivery processes and release value faster. Being able to deploy continuously and release on-demand enables the business to deliver business value in a controlled way at just the right time.

Layered test automation

Manual testing has value in exploratory testing and smoke testing but it's not enough. Layered test automation, once applied well, speeds up the whole development process. Automated tests provide more immediate feedback when code is changed. They reduce the need for manual testing of simple scenarios so that testers can do more exploratory testing. They support refactoring, which is often needed to keep code clean and coherent, providing fast feedback if a change has unintended impacts. Comprehensive test suites of automated tests have multiple levels – see The Practical Test Pyramid.

Although Automated Testing is no longer optional, many organisations still struggle with Automated Testing or even quality practices in general. Some organisations don't have any automated tests, yet are happy to spend over 50% of their development capacity on fixing defects, whether they be found internally or externally.

The first step is always the hardest, but once automated tests are added to the Definition of Done (DoD) and once the tests are included in the Continuous Delivery Pipeline, there are a large number of types of tests that can be automated. Unit tests, now sometimes also referred to as 'micro tests', test a single function or class. If there are dependencies, these can be mocked. The code may need to be refactored to enable mocking, usually by using the concept of dependency injection (DI). Average duration per micro test is typically a micro second.

Integration tests target a class or software component (eg. micro service), with their dependencies available as well. Tests in this space typically take longer to run (at times a few seconds) and they can be more expensive to maintain, so generally there are fewer of them. Other areas that can be automatically tested include performance, security, architectural compliance, etc. Even resilience when third-party systems may not be available can be tested automatically by simulating system outages or error conditions.

In general, testing through the interface is less ideal. Tests are often costly to write and maintain. Often, the test suite becomes brittle and there tend to be intermittent test failures which may require investigation and manual intervention. Generally, the best option is to test at the lowest/smallest level if possible and move up to more expensive tests only if and when it is the only option. Tests through the user interface should remain the absolutely rare exception.

It should also be noted that automated tests should not be written by dedicated people or teams. Instead, ideally, tests are written first, eg. based on acceptance criteria that form part of a feature. And those automated tests should be written by the developers who then implement the feature. Test Driven Development results in much simpler and better implementations and designs than test-first, let alone test-last. Quality must be built in rather than being tested in. Automated tests are just one of many quality practices that assist with this endeavour.

Performance engineering goals

It is important to set realistic system performance goals. There is often a tendency of non-technical stakeholders to request that the system is as fast as possible, but performance always comes at a cost. By working with stakeholders to set objective, measurable goals, the performance of a system can be measured and the system engineered to meet just those goals, minimising the risk of over or under-engineering the system. Teams working on the system will need data from system telemetry to see the impact of changes in achieving those goals (avoiding regressions/understanding improvements).

Infrastructure as Code

Manual testing has value in exploratory testing and smoke testing, but it's not. Representing Infrastructure as Code has become the norm, in particular with the increase in popularity of public clouds such as AWS (Amazon Web Services), Azure and Google. Instead of making changes to the cloud infrastructure through a console or a portal, the desired state of the cloud infrastructure is described in a generic language. Tools are then used to apply the required changes and migrate to the desired state. Since these files are text files, they can easily be versioned to enable better lifecycle management of the infrastructure. For example, new resources can be provisioned automatically by changing a few lines of code.

It's often desirable to have a production environment (PROD), as well as other non-production environments such as User Acceptance Testing (UAT) or development (DEV). With infrastructure as code, it's much easier to maintain multiple environments and keep infrastructure in sync. In addition, separating PROD from other environments assists with protecting the business, as well as customers' data and with meeting regulatory requirements.

Infrastructure as Code also plays a role in disaster recovery. In the unlikely event of an environment becoming unavailable, it's possible to create a new environment, perhaps in a different region, very quickly. Other environments – such as temporary environments for product evaluation or demonstration purposes – can be also created and managed easily.

We're ready to help

HYPR

Are your delivery practices constraining the flow of customer value? Do you want to release software at will? With our help, you can make the shift to Progressive Delivery. Call us now...

Gillian Clark – Director
gillian@hypr.nz
+64 21 642 079

Ajay Blackshah – Chief Practices Officer
ajay@hypr.nz
+64 21 747 633

Gareth Evans – Director and Chief Engineering Officer
gareth@hypr.nz
+64 21 0227 6744

Written by our team
Edited by Sian Hoskins
Design by Anam Berdugo, HYPR Creative Director

Thanks to all the clients and 'Friends of HYPR' who provided feedback and the pioneers of ideas and models that help us see things in new and different ways.

HYPR INNOVATION

GENERATOR NZ
Level 1, 22-28 Customs St East
Auckland Central
PO BOX 106-229
Auckland City 1143

www.hypr.nz

© HYPR INNOVATION LTD 2024 All rights reserved