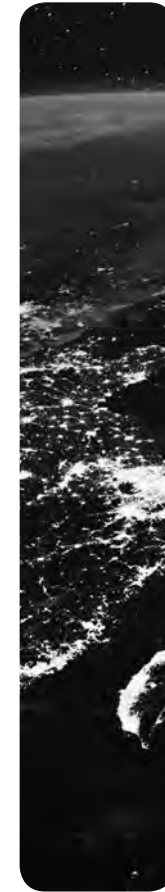
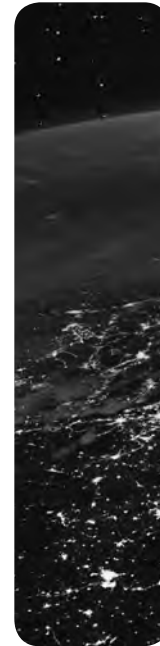




resill!on

Quality: Engineered. Tested. Assured.

Navigating agility
in the software
development
lifecycle:
an engineering approach



Setting the scene

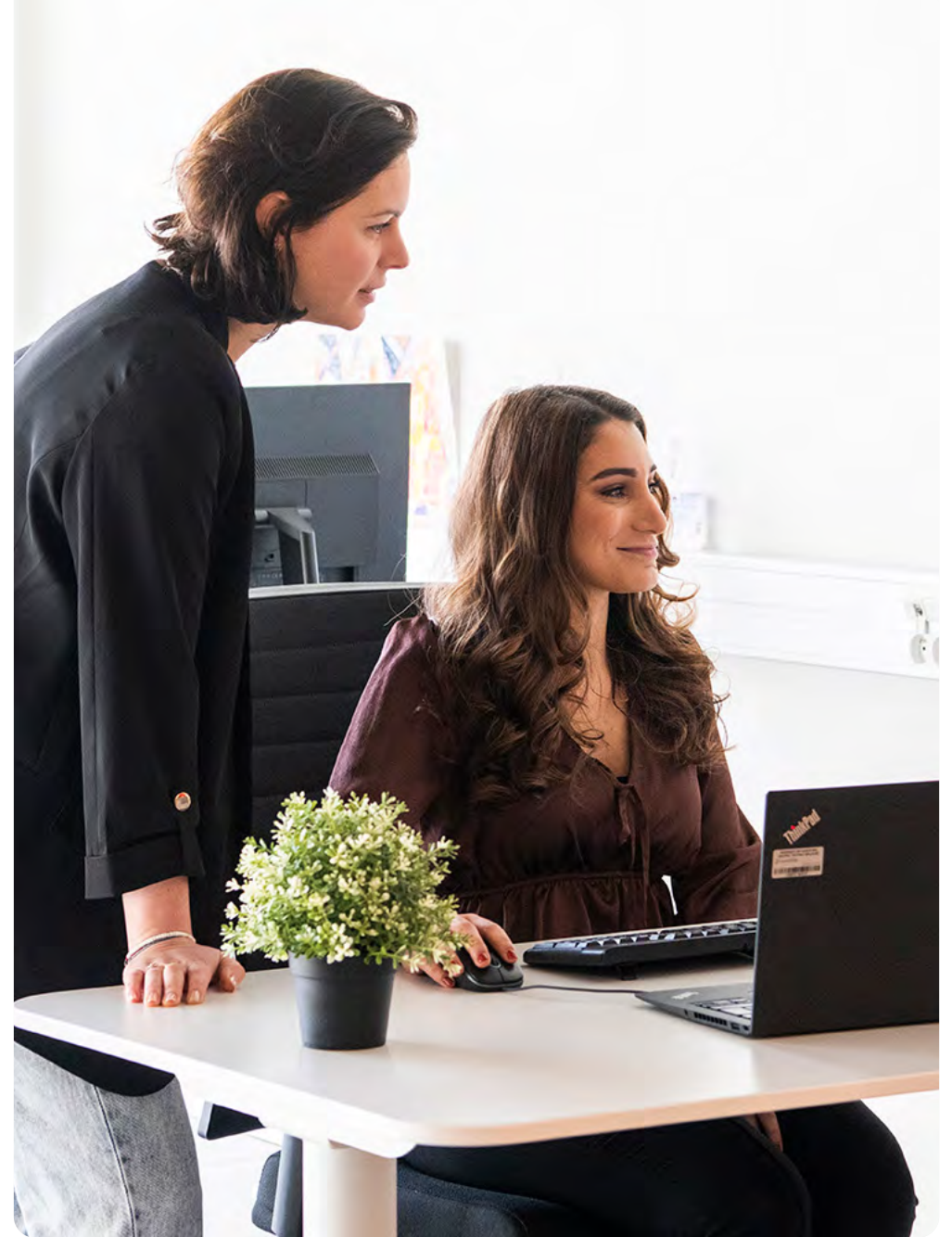
In our previous paper ‘Navigating agility in the software development lifecycle: a strategic approach’ we emphasised the importance of incremental delivery and an ‘inspect and adapt’ mindset. We also introduced the four DevOps Research and Assessment (DORA) metrics: Throughput (lead time for changes, deploy frequency) and Stability (change failure rate, mean time to recovery).

Being an elite performer on these four metrics, for example ranking highly on throughput and having excellent stability, can be interpreted as speed and quality going hand-in-hand.

We also concluded in the previous paper, that organising work into feature teams is good practice for scaling agile software delivery.

In this paper, we’re now going to look at things from an engineering perspective. What practices should software delivery teams adopt in order to be successful?

Similar to the ‘strategic perspective’ paper, we’ll focus on the principles rather than any specific implementation methods such as Scrum, XP or Kanban. Instead, it examines the engineering principles which have proven their value for DORA elite performers.



Introduction

Independent and multidisciplinary teams

In order to keep costs under control, generate quick and consistent business value and achieve competitive advantage, organisations that develop their software in-house need to consider quality, cost and agility from an engineering perspective.

The 'inspect and adapt' mindset should be the mantra of every software delivery team. In order to adopt this mindset as independently and effectively as possible, the team must be multidisciplinary. Each team should have all the required knowledge and capabilities: business knowledge, product knowledge, development skills, quality assurance, infosec and the ability to deploy to production (either by embedding the skills into the team, or via a self-service portal).

Alongside these build and deploy competencies, the team also needs to be able to incorporate and analyse production metrics. Logging or alerting mechanisms used to measure business value efficiently and fix issues quickly are also part of the team's responsibility.

To collaborate and communicate effectively, the team size should follow the 'two pizzas' rule: the team must be small enough that it can be fed with just two pizzas. In effect, this means no more than ten people per team.

Being both independent and multidisciplinary, without being overstaffed, is an important step towards creating a scalable organisation with high throughput and high stability. The ambition being an organisation where quality engineering is everyone's responsibility.

Feedback loop

One of the essential elements of the 'inspect and adapt' theory from the Scrum Guide is establishing a feedback loop (reference 1). A feedback loop must operate within a sufficiently rapid timeframe, akin to the feedback loop observed in system control theory.

There are **three activity areas** which all need a fast response:

- i. **Upstream to downstream:** From business idea to business value generation in production.
- ii. **Downstream to upstream:** Measuring the business value and behaviour in production and rapidly feeding this information back to the development team.
- iii. **The inspect and adapt culture** needs nurturing because it acts as the impetus to keep the feedback loop in motion. It enables the creation of fast and consistent business value while keeping cost under control.

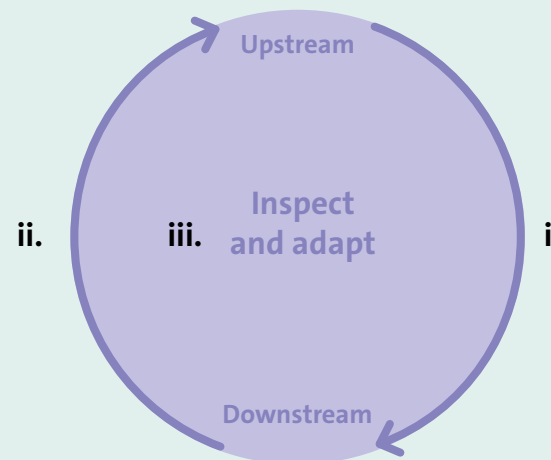


Figure 1 Inspect and adapt as a continuous improvement feedback loop. The feedback loop needs to be fast to be effective as a control mechanism, similar to system control theory.

Upstream to downstream: from business idea to business value generation

1. Process time and lead time

In software engineering, process time* is a measure of how long it takes from code being committed in a version control system until that commit is deployed in production. 'Process time' only takes into account the 'active' processing time and discards the 'idle' or 'waiting time'. In contrast, 'lead time' includes all time, from business request to customer delivery (reference 2).

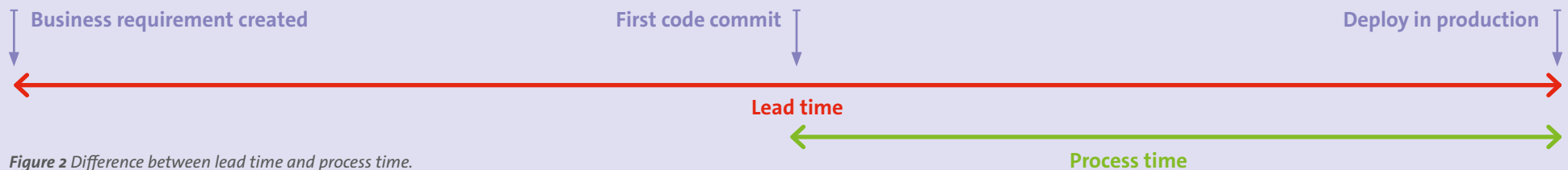


Figure 2 Difference between lead time and process time.

In our previous paper on 'a strategic perspective', we covered the DORA metrics. The 'process time' is the same as 'lead time for changes' in the first DORA throughput metric. It's a metric for the mean development time of a software delivery team.

2. Batch or iteration size

To achieve optimal stability and throughput from the business requirements stage through to business value generation in production, the priority should be to optimise the batch size. The inspect and adapt feedback loop only works as a control system when each iteration is sufficiently large, but not too big to impede throughput.

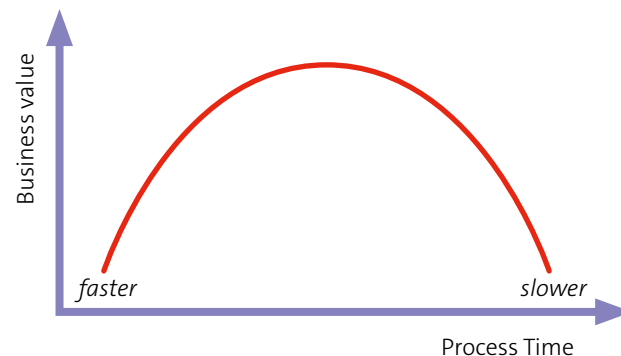


Figure 3 Trade-off of going too fast with not enough business value generation and too slow by moving batch sizes which are too large per iteration.

A pragmatic rule to determine the optimal batch size of deliverables is as follows: each iteration should only be as efficient and quick as it needs to be in order to generate the desired business value. There is a balance between going too fast, and not realising sufficient business value, versus going too slow, and missing out on risk-adjusted return on investment. The team must also measure the business value using adequate logging or other detailed metrics. In a situation where an iterative deliverable does not generate the expected business value, the loss is then limited to the quick, cheap deliverable of just one iteration. The knowledge gained from the 'lost' iteration indirectly generates business value because it gives a view of what features do not meet the needs of the end users.

* Process time is often referred to as "cycle time".

3. Always deployable

To consistently deliver a high level of business value, the engineering goal is to always have code in an 'always deployable' state. This is not achievable without continuous testing and validation at each stage of the software delivery process.

By having quality assurance baked in at every stage, there's no need for an elaborate testing and stabilisation phase before production deployment. The following sections summarise some of the best practice adopted by leading organisations according to the Accelerate and DORA metrics (reference 3 and 4).

a. Everything is code

Alongside feature code, the software delivery team must also consider the following by-products as code:

- build scripts
- automated (and manual) test suites
- deployment related scripts
- continuous integration setup scripts and configuration files
- software configuration management (SCM) files
- infrastructure-as-code scripts and any other infrastructure related configuration files
- logging templates and logging configuration files
- dashboard / reporting templates
- product documentation.

Treating these by-products as code in the version control system, preferably as close to the feature code as possible*, allows you to move towards full reproducibility. It's a step towards having executable scripts rather than lengthy (non-executable) internal deployment documentation or other guidelines.

Software source code commonly features a pull request flow, with code reviews and inspections as part of the daily routine. By having all by-products in version control, you get a similar pull request flow, before then merging them into the main / trunk development branch. This allows for experimentation, for example exploring how to stabilise a new way of deploying on a feature branch.

* Some organisations adopt a mono / single repo for all their code. This falls outside the scope of this paper.



b. Trunk-based development

Maintaining feature code and the build / deployment by-products in an ‘always deployable’ state, will be challenging if more than one code branch needs to be kept up to date over an extended period.

Trunk-based development means that a single branch (historically called ‘trunk’, but now often referred to as ‘master’ or ‘main’), is the single point of truth of the most recent stable development. Short-lived feature branches rebase their code frequently on this main branch, as well as frequently merging into it. Merging feature branches at least once a day means a faster feedback loop, ensuring an ‘always deployable’ state.

With trunk-based development, we recommend using the following rule:

‘Never commit on a broken build, unless to fix it’.

Whenever the main branch does not build or deploy, alert all developers and restore the main branch to a deployable state before continuing regular development activities.

c. Build once, run everywhere

The build artefact from each independently deployable component or service must be reproducible and immutable*.

After a software artefact has been built, it must be kept in an artefact store from which it can be deployed onto any environment. Any separately released software component must be built once, and after that, run everywhere.

Furthermore, a deployment on any environment must be reproducible. This is more straightforward if the deployment scripts are kept in version control as a by-product.

d. Continuous Integration (CI) and Continuous Delivery (CD)

Guaranteeing that software is in an ‘always deployable’ state resides in the domain of Continuous Integration, Continuous Delivery and Continuous Deployment.

Continuous Integration refers to the automated integration of code changes, made by multiple developers working together in a shared code repository. After a code commit on a feature branch is pushed, or when feature branches are merged into the trunk / main branch, a CI system is typically triggered to build, test and run a software artefact. Historically, testing as part of CI, was often limited to unit testing and system tests.

Continuous Delivery goes beyond CI, by also including automated system integration testing and automated deploys on environments more closely resembling production (where system interactions similar to production take place). A CD pipeline stops before a deploy into production. The final step – deploying into production – is not automated and requires human approval.

Continuous Deployment goes beyond Continuous Delivery, by deploying automatically into production as well. There is no human interaction after the software feature branch is merged into trunk / main. A CI / CD pipeline is created with several automated quality gates. If they’re all green, the build will be available in production rapidly after a code merge into the trunk / main branch. If the CI / CD build is red, the rule ‘never commit on a broken build, unless to fix it’ comes into play.

In the book ‘Continuous Delivery’ (reference 5), a pragmatic definition of what is considered ‘rapidly’ is given. In this instance, rapidly means that the developer still has cognitive focus on the code that is being merged. If you consider that a defect is usually cheapest to fix at the time it is introduced, then a robust CI / CD pipeline is a good investment.

* Immutable: it cannot be changed or tampered with. The only way to change the build object is by creating a new build object from the first step (with the necessary changes).

e. CI/CD best practices

Some elements of best practice for any organisation looking to implement or extend a CI / CD pipeline:

- Non-functional testing has a role to play at all stages of a CI / CD pipeline. This ensures that performance, security and any form of auditing checks become part of daily business as usual and not just tagged on to the end of a project cycle.
- As mentioned in section 3.3a, the CI / CD pipeline code should be version controlled and preferably follow a similar code review process as feature code. A (modified) pipeline build is triggered after a pipeline code change (albeit without a functional change) to ensure that no regression occurs in the pipeline.
- The traceability and repeatability of a deployment pipeline is important. Make sure that the same build can be retriggered consistently. When all by-products are version controlled, configuration management files and deploy scripts can easily be restored later to help with root cause analysis if a post-mortem investigation is required. Build log files and test reports of each pipeline run must also be stored for a sufficiently long time. For software development in regulated sectors, these logs and reports can serve as evidence for audit and compliance purposes, without much human effort.
- Although not strictly part of a CI / CD process, gated code commits help assure an always deployable trunk / main branch. Best practice is to use short live feature branches that rebase frequently on trunk / main and to only allow fast-forward code merges to trunk / main. If a merge does not produce a green build artefact (usually early in a CI / CD pipeline), consider a post-merge script that automatically rolls back the merge activity.
- Being in an always deployable state with small, quick changes, makes it challenging to create up-to-date release notes and customer-facing documentation manually. They too should become part of the build pipeline process. By adopting conventional commit messages (see reference 6), release notes can be deduced from commit messages. Documentation can be kept up to date by storing it close to the feature code in version control. A peer review before a code merge should check that documentation changes are indeed included.
- All regression tests that affect the system that is being built, must be run as part of the CI / CD pipeline.

As well as providing confidence when automated tests pass, continuous delivery must also give assurance that all the required configuration is ready in order to deploy the build artefact on the production environment. That is why adopting an ‘inspect and adapt’ mindset requires identical environments: a local developer’s workstation, a team-controlled test server, an acceptance or staging integration environment or even the production server must have identical setups. To ensure similarity between environments and to keep it workable, service virtualisation (or stubbing or mocking) can be used alongside API contracts and contract testing. Communication between all independently deployable components must follow the same protocol on each environment.

An extra benefit of having identically configured environments is that additional environments can be enabled (‘spun up’ in a cloud native landscape) quickly and disabled when no longer needed. When adopting a modern cloud strategy, the same setup and configuration scripts can be used for scaling capacity in production when the traffic load is high.

In addition to having infrastructure setup and configuration files identical, monitoring and logging setups must also be identical for all environments. This reduces the risk that monitoring and logging in production won’t work and increases the speed of diagnosing issues, because each member of any software delivery team is familiar with the logging system.

f. Continuous testing

To have high throughput and stability with code in an 'always deployable' state, QA engineers must be included in the team. In a continuous delivery model, QA engineers ensure that continuous testing happens throughout the pipeline, as well as monitoring and analysing test coverage.

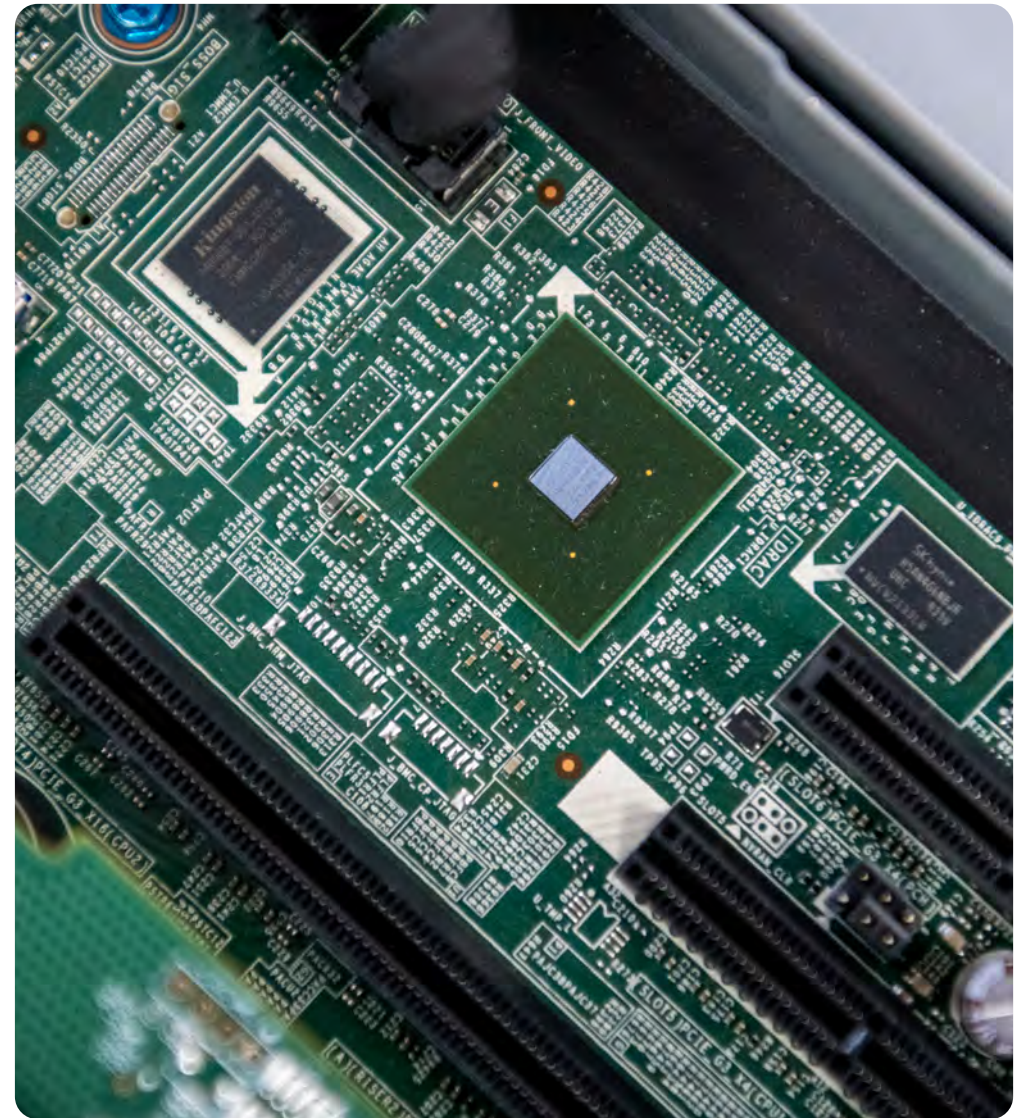
Their 'quality mindset' challenges other team members to bake quality in at the most appropriate test level and they can contribute by writing or reviewing fast, reliable and automated test suites. They can also be on stand-by to provide additional risk-based, exploratory, manual testing, thereby preventing unforeseen system usage.

With 'everything as code', test automation follows the same pipeline flow as feature code, by adopting pull request approvals and identical coding standards as feature code.

In addition to automated pipelines that are triggered after a code commit or code merge, further regular triggered pipelines should be set up, e.g. nightly scheduled stress testing or disaster recovery testing.

Continuous testing is about testing systems starting right from a code commit, moving on to automated testing across multiple test levels and continuing all the way to production-like systems with production-like load testing, 24 hours a day, 7 days a week.

Testing and QA activities considered with an 'inspect and adapt' mindset with high throughput and stability will be the scope of the following paper focused on software delivery from a quality perspective.



g. Deployments and responsibility

The goal is to have a low-risk release and deployment approach that ensures a balance between speed, quality, and responsibilities in the deployment process.

Low risk in the long run, requires releasing from the trunk / main, with the possibility of creating a release branch afterwards for hotfixes, but only if the trunk has breaking changes. If the deployment process is not included as part of an automated artefact promotion within the build pipeline, then it should be designed as a self-service activity, executable with a single click. This process should be identical across all environments: Developer workstation, Low level test or integration environments, higher level acceptance environments and production.

After each deployment, checks must be executed to ensure that all functional systems and endpoints are up and running. Developers and testers must be empowered to deploy on any internal environment without handing over deployment tasks to a separate team. Production deployments should ideally be done by the teams who own the application. Any risk mitigation and control mechanisms should be implemented early in the CI / CD pipeline as part of artefact promotion quality gates. The deploy should be done as quickly as possible, aiming for as close as possible to zero downtime for the end users.

A broken deploy, in the sense that everything is code (see section 3.a), must be handled like a broken build: all commits stop, except to fix the deploy. Here, the trunk / main branch (if deployment code is in a separate repository) should be kept in a releasable state at all times.

Other deployment practices to consider are:

- try a canary release or dark launch to a small group of end users by using feature toggles
- blue / green environment deploys, that alternate production traffic depending on which environment has the latest changes
- logs of all deployment activities should be recorded for auditing and compliance purposes.

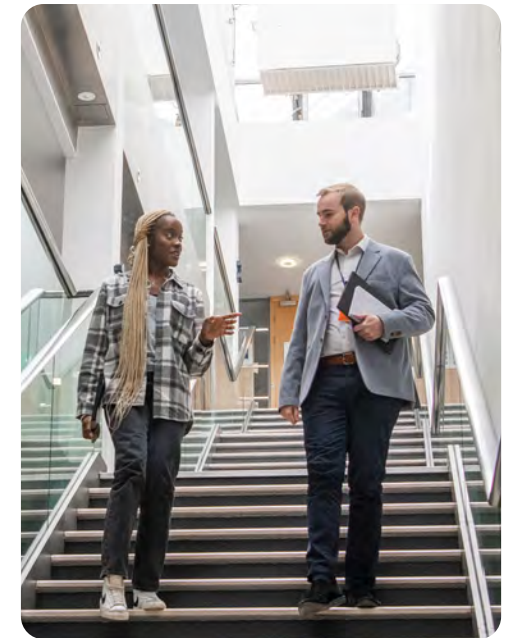
Releasing new features should be as simple as marketing announcing that new functionality has become widely available. The functionality should already have been tested in production (hidden from users). A feature release does not require a change of application code or build, only environment or runtime configuration. Releases can be scheduled during business hours, or end-users can choose when it's convenient for them to upgrade their system (e.g. point-of-sale system in a store).

Big bang releases should never be undertaken, not for fresh launches and not for long, complex migrations. Architectural patterns exist, but they go beyond the scope of this paper.

h. Shared services and platform engineering

When it comes to high throughput and stability, shared services become an enabler for software delivery teams.

For example, an identity and access management service which is security, performance and audit compliant can be re-used by all delivery teams. 'Platform Engineering' where dedicated teams facilitate this for other software delivery teams is common practice nowadays.



4. Pay off technical debt

When projects have accumulated technical debt, it's hard to overcome the interest on 'debt payment'. According to research (see reference 8), technical debt has an immediate cost by wasting 42% of developers' time.

There is no clean definition of code quality, but code health (an aggregated metric based on 25 factors, see reference 8) can be considered as its measurable twin.

The code health of each individual file can be measured and categorised using a green / yellow / red classification to indicate the health status of each file. The authors of the study in reference 8 found that qualitative healthy code has lower actual time-in-development*:

- on average, green code is twice as fast going into production than red code.
- variation (the uncertainty): Feature implementation on red code can be up to nine times slower.
- red code contains 15 times more defects.

Keeping the code base healthy pays off in terms of fewer defects. A healthy code base is a low hanging fruit in many engineering organisations. Technical debt derived from poor architectural choices is more difficult to fix.

* Software engineers did not have to do manual time tracking. The time-in-development was automatically calculated by how long stories or tasks were in predefined states in the ticketing system. The stories or tasks were then linked to code changes in version control.



Downstream to upstream: measure business value and learn

When dealing with complex* software systems, it's crucial to get feedback quickly. This is why it's vital to focus on quality right from the start. A strategy known as 'shift-left' is required in the flow from upstream to downstream. But there is more to it. By allowing software development teams to deploy their work in production by themselves, you not only get fast feedback from downstream to upstream but also build confidence in efficient deployments. This continuous cycle of feedback and adjustment promotes a culture of learning and improvement, leading to superior, more efficient software delivery.

1. Dev and Ops

Culture and people play a pivotal role in ensuring stability across all environments, including production.

The responsibility is shared among all participants in the process, rather than being solely on the shoulders of a dedicated operations team. Software development teams should be part of the on-call duty team, ready to provide support when needed. This shared responsibility and involvement promote a sense of ownership and accountability, leading to more robust and reliable software systems.

When a service in production becomes unstable over a longer period, it's beneficial for the development team that built the service, to work closely with the operations team. The rationale is that an issue with one service should not monopolise all of the operations team's time, which could otherwise be spent on other services. The collaboration between developers and operations can lead to valuable insights, which can be used to improve the design of other services for better operational efficiency in production.

For software delivery teams, the definition of 'done' should extend beyond code completion. It should include successful deployment into production and subsequent observation to ensure that the feature that's been delivered is stable and functioning as intended. Moreover, it is crucial to verify that the feature is delivering the expected business value.



* A complex system is a system where no single person is able to oversee the whole and understand how all pieces work together.

2. Metrics

Throughput and stability are key. High stability, which corresponds with a fast Mean Time To Recovery (MTTR), helps mitigate risk in production and builds confidence by enabling rapid diagnosis and fixes.

A central approach to production metrics and monitoring is crucial.

This includes:

- having identical metrics in production and all pre-production environments
- making monitoring dashboards accessible to everyone in the organisation
- sharing technical logs widely to avoid data silos
- generation of (production) metrics as part of daily engineering work.

Including production performance metrics alongside investment in proactive alarms management and anomaly detection is also important. Metrics and logs should be stored for post-mortem improvements, embodying the 'inspect and adapt' principle.

Metrics should also be applied to testing and pre-production deployment activities. For example, alarms could be set for test execution staking longer than expected. Testing dashboards, including historic progress charts, should be publicly exposed. If test suites fail frequently, anyone who sees this should be able to ask why and how they can help. This approach stimulates a culture of shared responsibility and continuous improvement.

Metrics should be 'full spectrum'. They should be relevant to various stakeholders and cover different aspects of the system. For business stakeholders, relevant data such as concurrent sales transactions, end-user churn rate, and average attention span should be highlighted. Application or service-specific data, like the average time of service calls and warning or error rates, provide insights into the performance of individual components. Infrastructure-specific metrics, such as database load, network performance,

and file system outages, help monitor customer facing applications. Abnormal high load and exceptional program terminations, can be related to InfoSec events and are crucial for maintaining security. End-user tracking, including user analytics, provides valuable insight into user behaviour and preferences. Any other gaps identified should also be addressed to help achieve an 'inspect and adapt' culture and contribute to continuous improvement.

Data analytics can play a pivotal role in identifying and addressing technical bottlenecks that may be impacting business value. For instance, timeout screens shown to end users could lead to missed sales transactions. By analysing such incidents, it becomes possible to link technical issues directly to business outcomes and take corrective action. It's useful to identify learnings from leading companies in the field, in particular looking at their site reliability engineering practices. Adopting proven strategies can accelerate progress and efficiency. The goal is to discover and resolve problems before they have a significant impact on business value. This proactive approach can help maintain a high level of service quality and customer satisfaction.

3. No blame culture

For example, trying to prove quickly that your team is not at fault during an outage, a practice called 'mean time until declared innocent' can be counterproductive and foster a toxic environment. Instead, teams should constructively work together across departments to diagnose, fix and prevent issues. This creates a positive culture where everyone shares responsibility. Using a scientific approach to analyse cause and effect can further enhance this process. This not only resolves the immediate issue but helps to prevent similar problems in the future.

Inspect and adapt culture as enabler for business value control

1. Allow experimentation

Organisations should cultivate an environment where product development is entrusted to multidisciplinary, empowered delivery teams.

This approach enhances customer outcomes through ‘inspect and adapt’, underpinned by throughput and stability. Teams should be empowered to conduct experiments in production, independent of management approval.

Consistent with the risk-adjusted return on investment discussed in the first paper on strategic perspective, it’s important to acknowledge that not all experiments deliver the desired business value. Rapid iteration is key in influencing the business outcome. In some instances, it may be necessary to discontinue underperforming product features. This is risk adjusted return-on-investment in action.

The objective is to build the most cost-effective and fastest experiment, and then measure to see if it generates the desired business value. The right product metrics must be in place to facilitate a swift and iterative ‘inspect and adapt’ approach. It’s crucial to involve UX engineers in the multidisciplinary software delivery teams. Adopting methodologies such as A / B testing and customer surveys can provide valuable insights into user behaviour and preferences, thereby informing and enhancing the product development process.

2. Never stop learning

When adopting good practices from upstream to downstream and using appropriate feedback mechanisms from downstream to upstream, every incident is a chance to learn and become more resilient to the same type of issue.

Any failure is an opportunity to learn and self-improve. You might add new automated tests to trigger the conditions of past incidents or include extra metrics on pipeline and production environments which generate alerts from which recovery mechanisms start operating. Care must be taken that lessons which are learned locally by a team, are shared with global teams.

To expect the unexpected and never stop learning, organisations can decrease fault tolerances to find weaker failure signals for example by learning from near-failures instead of actual outages. Organisations can also check their resilience by injecting anomalies in (pre-)production systems. Mature organisations rehearse server and system outages to test their resilience and automated recovery systems.

Latent defects can be discovered through anomaly or controlled failure injection. These are not directly observable defects, but issues that occur when trying to handle an anomaly. For example a new server instance can be activated to take over a service after the anomaly, but automated traffic routing might make it non addressable. The most important aspect is training (self-improvement) to become more resilient and prepared for high throughput and excellent stability.

Besides the technical point-of-view, the resilience of business processes is important. If a critical business system (e.g. the identity provider login system) becomes unavailable, do people know what to do or do people start improvising (e.g. employees start spamming their manager via private messages) causing a risk for business continuity.

3. Free flow of information

Using process documentation that's shared via a wiki, intranet page or shared document often results in out-of-date material or information overload when trying to find the correct data.

Documentation should be part of daily work, so try to go for executable documentation. Write deploy scripts that people can easily read. If a change is needed, the executable script becomes the documentation. The goal should be to have all documentation as executable code in version control. All teams that need it, always have the latest version without relying on wider comms. These scripts can even have their own CI / CD steps to catch issues, before any harm is caused to the consumers of the script.

4. No knowledge bottleneck

With 'throughput and stability', independent software delivery teams build the services, and should be closely involved in running them in production. This meets the organisation's wider goals and helps generate business value.

In cases where a software delivery team has their own niche technology stack that very few others in the organisation understand, this becomes an organisation-wide risk.

This team creates a knowledge bottleneck, they fail to translate local knowledge into global knowledge and the wider organisation realises no benefit. In this case, the organisation must ensure that optimising for independent team productivity is not in conflict with the company goals. This bottleneck can easily be resolved by moving away from separate technology stacks.

5. Plan time to optimise, learn and teach

The organisation must plan learning and experimentation time for all engineers, in order to optimise the daily workflow for 'throughput and stability'.

This doesn't include time on feature work or innovation activities like hackathons or hackfests. The goal is to get the value stream from upstream to downstream and the feedback stream from downstream to upstream working as efficiently as possible. This is the time to remove impediments or technical debt and optimise daily work.

This learning time can either be run continuously, e.g. 10% or 20% of time can be dedicated to it, or as a batch, e.g. a one-week sprint with the whole organisation every couple of months.

Independent of optimisation and learning time, people must be encouraged to share knowledge. This can be in teaching sessions or by organising KATAs* or equivalent. Teaching or knowledge sharing sessions in multidisciplinary software teams aims to achieve T-shaped profiles in the team. T-shaped profiles means that you have a major area of expertise (e.g. Java development) and you have a minor area of expertise (e.g. front-end UI test automation).

If delivery teams don't have the time to optimise, learn or teach, consider having an organisation-wide coaching or consulting service team with engineers who can join the busy delivery team temporarily, helping them to optimise their daily work. These internal coaches also learn from this situation and can spread locally acquired knowledge to the global organisation.

Start your quality transformation

Find out more about our Quality Engineering services [on our website](#) or [get in touch](#) with us today.



* In the context of software development, a KATA is an exercise to improve developers' skills through practice and repetition. KATA originally comes from martial arts - it's a way to memorise and perfect movements.

References

- 1 K. Schwaber and J. Sutherland, The Scrum Guide (edition November 2020), 2020.
- 2 "Cycle time (software)," Wikipedia, 3 11 2023. [Online]. Available: [https://en.wikipedia.org/wiki/Cycle_time_\(software\)](https://en.wikipedia.org/wiki/Cycle_time_(software)). [Accessed 3 11 2023].
- 3 N. Forsgren, J. Humble and G. Kim, Accelerate: Building and Scaling High Performing Technology Organizations, IT Revolution, 2018.
- 4 Google, "State of DevOps Report," Google, [Online]. Available: <https://cloud.google.com/devops/state-of-devops/>. [Accessed 27 04 2023].
- 5 H. J. and F. D., Continuous Delivery (Reliable Software Releases Through Build, Test, and Deployment Automation), Pearson Education, 2010.
- 6 "Conventional Commits," 2024. [Online]. Available: <https://www.conventionalcommits.org/en/v1.0.0/>. [Accessed 01 01 2024].
- 7 "OpenAPI Initiative," 3 11 2023. [Online]. Available: <https://www.openapis.org/>. [Accessed 3 11 2023].
- 8 A. T. a. M. Borg, "Code Red: The Business Impact of Code Quality -- A Quantitative Study of 39 Proprietary Production Codebases," Accepted for International Conference on Technical Debt 2022, 2022.
- 9 D. Farley, Modern Software Engineering: Doing What Works To Build Better Software Faster, Pearson Education, 2022.
- 10 D. North, "Scaling Agile Delivery: Turning the lights on - Agile tour Vienna 2015," 2015. [Online]. Available: <https://www.youtube.com/watch?v=mWXTYNhz-sk>. [Accessed 24 04 2023].
- 11 D. North, "Scaling Agile Delivery Turing the lights on - Agile tour Vienna 2015," 2015. [Online]. Available: <https://www.youtube.com/watch?v=mWXTYNhz-sk>. [Accessed 25 04 2023].
- 12 C. M. Rebelo, "Agile's worst enemy is not waterfall - is bad agile," 14 12 2018. [Online]. Available: <https://www.linkedin.com/pulse/agiles-worst-enemy-waterfall-bad-agile-cristina-moura-rebelo>. [Accessed 29 04 2023].
- 13 "SAFe - Scaled Agile Framework," [Online]. Available: <https://scaledagileframework.com/>. [Accessed 29 04 2023].
- 14 J. H. P. D. J. W. Gene Kim, The DevOps Handbook, IT Revolution Press, 2016.
- 15 M. Y. a. P. F. Gary Gruver, A Practical Approach to Large-Scale Agile Development: How HP Transformed LaserJet FutureSmart firmware, Addison-Wesley Pearson Education, 2012.
- 16 J. Humble, "Why Scaling Agile Doesn't Work • Jez Humble • GOTO 2015," 2015. [Online]. Available: <https://www.youtube.com/watch?v=2zYxWEZogYg>. [Accessed 24 04 2023].
- 17 Google, "DORA | DevOps Quick Check," [Online]. Available: <https://dora.dev/quickcheck/>. [Accessed 27 04 2023].
- 18 ISTQB (International Software Testing Qualifications Board), Certified Tester Foundation Level (CTFL) Syllabus: Version 2018 v3.1.1, International Software Testing Qualifications Board, 2018.
- 19 H. Son, "What is the Role of QA in DevOps?," 26 04 2022. [Online]. Available: <https://www.testrail.com/blog/qa-devops-role/>. [Accessed 05 04 2023].
- 20 S. Mandot, "Continuous Testing in DevOps," 22 10 2020. [Online]. Available: <https://blog.gurock.com/continuous-testing-in-devops/>. [Accessed 31 01 2023].



resill!on

Quality: Engineered. Tested. Assured.