GETTING REAL VALUE FROM AGILE DEVOPS

PROJECT one

real change • real difference

There's a lot of noise around DevOps, Agile and other change program methods, which makes it hard for leadership teams to know what to believe and what is required to improve results. The Agile (and Lean) approaches have been around for years, and yet when we visit different companies and drill into how they deliver change it's still hugely inefficient.

Most CIOs and CTOs we talk to say they have adopted some/all of these methods, but there's often **no hard evidence** of a quantifiable improvement in **efficiency** or visible improvement in **time to market;** indeed, there often tends to be a pride in and a focus on the **method rather than the outcome.** But what CEO or business MD cares about the method of the change program, unless it has a direct correlation to an increase in revenue, profit or customer value?

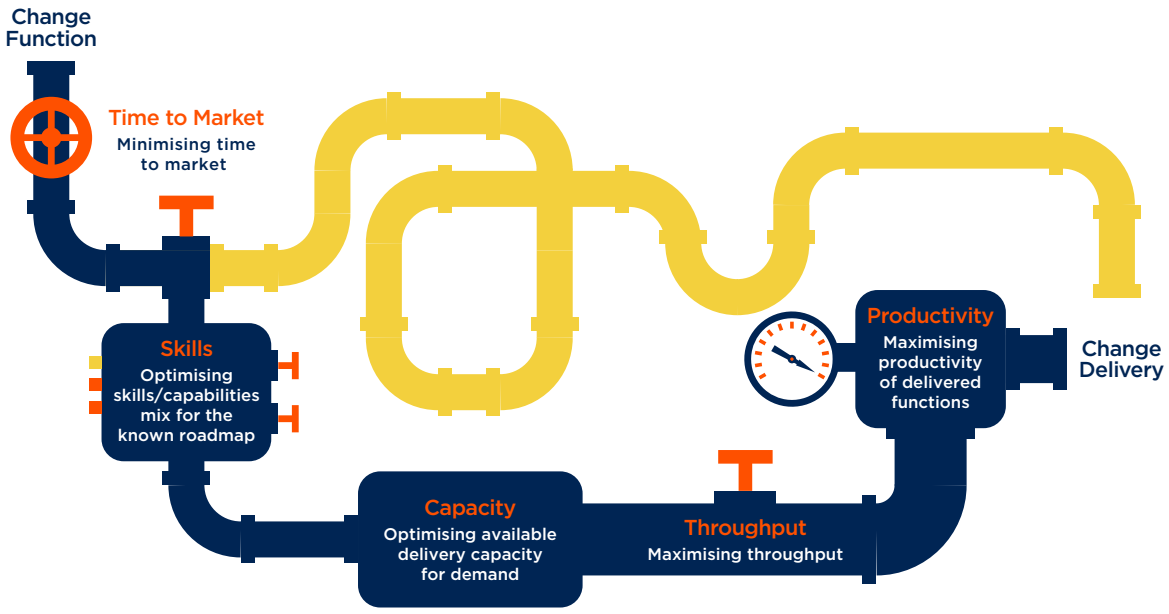
Individual projects can look healthy, but when there's a hundred of them all **delivering in parallel,** all dependent on shared functions, then it becomes a real challenge. Typically, you'll have several customer or business areas who all have a change agenda, and there will be a **finite pool of delivery resources.**

SO HOW DO WE APPROACH IMPROVING RESULTS IN A STRUCTURED WAY?

Follow our four step approach:

1. YOU GET WHAT YOU MEASURE – SO MEASURE DIFFERENTLY

Think of your change delivery function as a pipe, gushing out **value from delivered projects.** Using a pipeline analogy for the flow of change through the delivery function helps to focus on the five key high performance measures:



It goes without saying that a high-performing delivery function starts with the **right changes being shaped and prioritised** – to paraphrase the oft-used saying – request the wrong things, and the wrong things will be delivered.

2. KNOW YOUR CAPACITY CONSTRAINTS, AND ELIMINATE THEM

There are always more good ideas than there are resources to deliver them.

In other words, **demand always outstrips supply** – it's the nature of any healthy business – and if there's an oversupply, you can be sure a cost reduction exercise will be imminent! So, what is required is clear and simple prioritisation of demand vs available supply.

Of course, it is often the case that businesses cannot quantify their change delivery constraints, beyond an overall strategic investment cap. Sometimes, outsourcing contracts will also blur the lines of capacity constraints – but they will be there, and it will benefit you to keep digging until you establish what the constraining factors are.

TYPICAL CAPACITY CONSTRAINTS AND HOW TO ELIMINATE THEM

Typical supply constraints will include:

Developer capacity

- This is usually the best-understood constraining factor, because it tends to be a straightforward measure of the number of "warm bodies" available to code a particular system. On larger systems there will often be sub-teams looking after specific areas of functionality, or configuration, or the technical aspects of the system, and each needs to be quantified separately. In quantifying the constraints, you need to balance the need for a simple-to-understand homogeneity with the realities of differing levels of skill, knowledge, experience and capability
- Improving the capacity is theoretically simple – more bodies and/or more productivity from the existing team – but needs a medium- to long-term strategy to maintain the right skills and knowledge, and a sensitive approach to productivity improvement to maintain a healthy, motivated team

Test capacity

- If testing is predominantly a manual exercise run separately to development, this is usually an easily-understood and easily-manpulated constraining factor, because test resources tend to be more fungible and more easily sourced than development resources who will need a specific set of technology language skills. More testers = more testing
- However, in many organisations where testing is partially automated, clearly more automation can mean that more testing can be done in a shorter time period – but don't take this for granted. Typically, automation works best for regression testing – automating repeatable processes to check you haven't broken them – and it can be quite time-intensive to create new test scripts for your changes – so don't assume automation will immediately remove your test constraints – it's more likely to prevent them getting worse

Subject Matter Expert (SME) and Singe Point of Failure (SPOF) Capacity

- We often find that one of the main constraining factors in overall delivery of change is the over-reliance on a small number of experts. Note that this isn't just about technical SMEs, it can often be business expertise. Constraints like these are usually well-known (having existed for some time) but nebulous as they are hard to quantify and pin down. By their nature, SMEs tend to be pulled into a mix of business as usual, crisis and change activities, and it is hard to plan their time. The simplest approach to managing this is to identify the constraining SMEs, obtain acknowledgement from the relevant leads, and agree an allocation of time (e.g. a number of days per week) which are managed and tracked. Better still, doing a simple time/motion study can throw up a number of activities which they could pass on to other colleagues to optimise their time
- A compounding issue is that nobody feels particularly incentivised to solve the SPOF problem – the individual usually wants to retain the relative importance and power, and no programme sponsor sees this as their problem. However, it is critical to resolve for the medium- to long-term, not just to remove the bottleneck but to mitigate succession risks. Interestingly, it is often not as hard as anticipated!

Environments

- In a world of physical test environments, developers and testers are often constrained by a finite number of places to test – which means an artificial deadline for code changes to be promoted into a single build, and the inevitable sequential smoke/regression tests before the testers can get their hands on the changes
- Clearly, this constraint can be blown away if you can spin up environments on demand in the cloud, as you can have one per developer to allow each developer to test not only his/her changes, but their changes alongside everyone else's. This puts a large onus on code control and branch management (which has always been critical!), and can be made a lot easier through using build management software

It is then critical to be able to articulate these constraints in a simple-to-understand way, as typically these constraints will drive your demand and supply prioritisation decisions, and, often, project timelines.

If you can't articulate the decision choices on a single page, you won't get a decent decision!

3. OPTIMISE YOUR CRITICAL PATHS

Crazy as it seems, we often focus on critical path activities for projects and programmes, but we rarely focus specifically on optimising the activities that are on the critical path. How many times have we seen project teams waiting for a working test environment, or a working code build – or even - for an agreed design?

Typical areas for big gains:

Project Mobilisation

A week lost in project mobilisation is as critical as a week lost just before go-live

It's funny how elapsed time doesn't seem so precious when you're in the shaping and mobilisation phases of a change program – and yet the closer you get to project go-live, the more the urgency and executive attention increases! Elapsed time on the critical path is critical wherever it is. Many change programmes also suffer from a long gestation period before work starts in earnest – but from an ROI perspective, the clock is ticking!

We often see a chaotic and muddled process for mobilising projects, which is more akin to Brownian Motion than anything else. Whilst this is, by necessity, a phase of high uncertainty, critical path is almost never considered at this stage. Although prioritisation deserves an article of its own, prioritising between disparate initiatives is often a **"comparing apples and pears"** debate. Typically, Change Boards are there to govern this phase, but decisions can often take a long time, and in our experience, too few initiatives are killed off or completely reshaped early on.

Three key lessons:

1. Consider your critical path from the point of logging the idea – if nothing else, to focus people on the opportunity cost and need to take swift, smart decisions

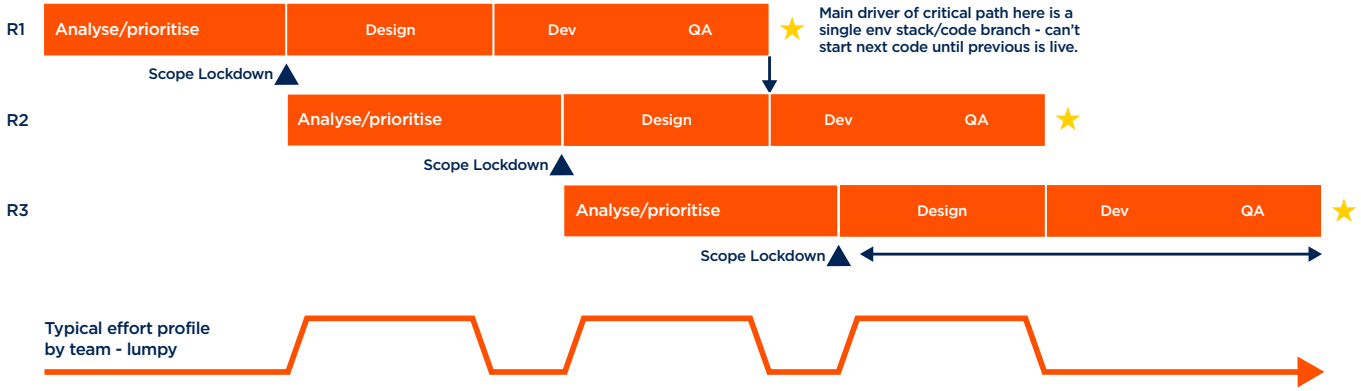
2. Follow an iterative process for shaping and mobilisation:
   - Consider design "spikes" – timeboxing and ringfencing to reach a pragmatic solution shape
   - Consider educating stakeholders in Lean Startup methodology as a way to focus on validation, customer input/feedback, and iterating from a minimum viable product

3. When prioritising, consider changing the solution rather than just the dates or the sequence. On the basis that a particular project can't be started for six months due to other priorities, do you still want the same thing, or will the market have moved by then? Could you make do with a tactical answer – or could you implement a cheap/quick pilot and use customer feedback to influence your strategic solution?

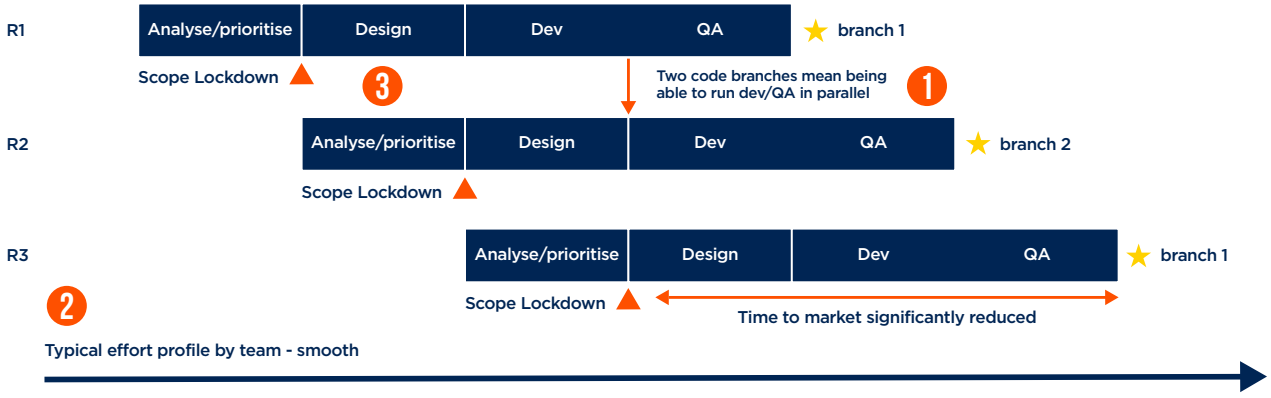
ROLLING RELEASES

Let's look at some typical inefficiencies:

Typical



Target

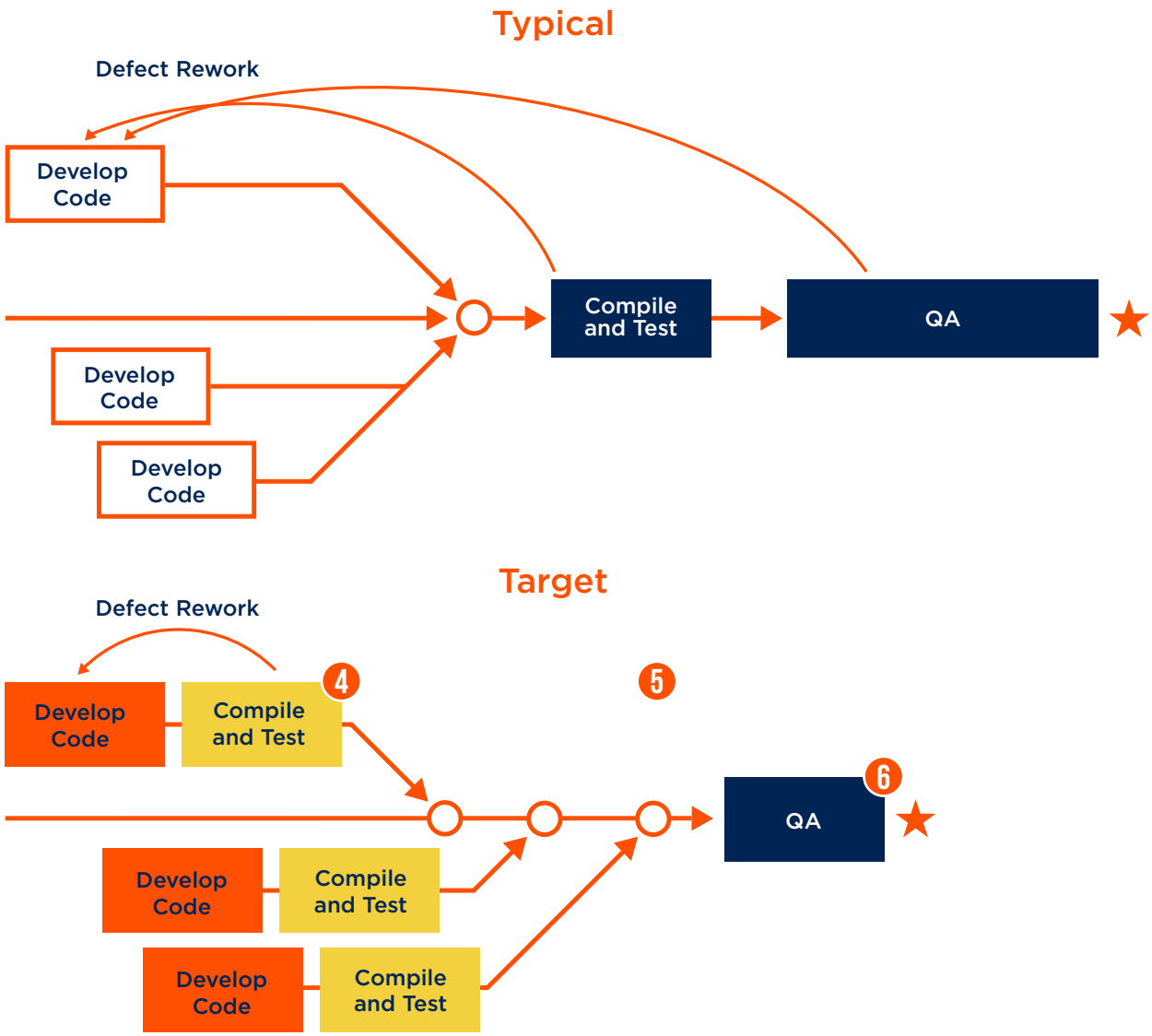


There are multiple opportunities to be had by synchronising the current release schedule with the schedule for the next (and the next) release:

1. By understanding what the main critical path driver is – in the example above, code branching limitations – and addressing it – you can typically double the frequency of releases by removing the blocking dependency
2. You can smooth each team's workload so they move from, say design of one release, straight into design of the next – meaning greater productivity
3. Through maintaining a backlog of analysed changes, you can delay scope lockdown until just before Design – meaning significantly shorter time to market for prioritised changes

DEVELOP AND TEST/QA

Let's drill into the Dev and QA phases:



Again, there are big opportunities to be had:

4. By providing developers with the continuous integration tools to be able to submit and test their code changes against the continually-working master codebase, means that there is minimal delay in finding out if the code doesn't work!
5. By maintaining a working build (master codebase) in a production-like environment, you eliminate the "shock" of code no longer working in a different environment
6. QA effort and duration can be minimised to purely integration and any end-user testing required

Whether you call this DevOps or something else is up to you – it's just good practice, and the outcome is an optimised critical path.

CHANGE / RELEASE MANAGEMENT PROCESS

- IT change and release management is a critical assurance step in controlling change into the production environment. However, in our experience, the process isn't always fit for the volume and speed of change. Often, we see 5- or 10-day lead times or SLAs on change, and Change Approval Boards trying to get through 200+ items, consequently unable to challenge any
- The mantra here is **automate,** and if you can't automate, **be pragmatic.** The same continuous integration toolsets which allow you to submit and test code within minutes of writing it, will also automate the process for releasing to production – meaning reduced "fat-finger" production issues, the ability to deploy changes independently of one another (so less waiting for a scheduled release date), and less "shock" to production from multiple changes being deployed at once

GOVERNANCE CAPACITY

- In rare cases we have found companies or business units where the lack of delegated decision making rights meant that the critical path constraining factor was actually getting **access to the MD** to take decisions! This is fairly unusual, but in organisations where there is a formal "drumbeat" of governance, decisions get delayed or (more likely) governance gets bypassed. There is no golden answer to governance, but how and when decisions are made (and by whom) needs pragmatic and creative consideration

There are plenty of other opportunity areas to go after – the key is to **focus on the critical paths.**

4. REDUCE YOUR DEPENDENCIES THROUGH AUTONOMY

Create autonomous delivery functions wherever you can

If there are changes to specific products, systems, or even types of change which can be parceled off to be delivered by a ringfenced team à la Agile, then why wouldn't you minimise the dependencies and allow this team the autonomy to manage their own backlog? In this way, you've reduced the constraints to a simple factor, which is the size of the scrum/sprint team and consequently the amount of changes they can work through in a set period.

If you're just starting out on the **Agile maturity journey**, it makes sense to start with change types or technologies which are well suited to being delivered by an autonomous delivery team. For example, eCommerce front ends, ongoing small enhancements to legacy systems of record, configuration changes to your CRM system – all perfect for an Agile approach of ringfencing a multi-disciplinary team and allowing it to self-manage the delivery of the backlog of change for an agreed area (with appropriate controls).

Back to the pipe analogy, this is about creating separate pipes which can deliver specific types of change independently (and therefore at the optimal flow rate) from the "mainstream". Of course, the autonomy of a ringfenced team implicitly means a delegation of decision making rights – can the executive trust the delivery teams?

Treat your shared functions as "services"

Ringfenced teams will inevitably be dependent on **cross-delivery shared functions** – whether for oversight/direction (e.g. Architecture teams), for consistency/reuse (e.g. middleware) or for service operations and management. Whilst you can minimise dependencies you can't eliminate them completely, so it's important to set out how these functions will manage their multiple customers. The answer is a service model – with a clearly defined set of services, a simple engagement/work initiation approach, and a pragmatic demand/supply.

For Example:

An architecture function may provide enterprise, process, system and data architecture services from a team of 6-8 people, with a remit of maintaining strategic direction of travel across 100+ projects. It's critical for the sustained operation of this team that their workload and priorities are continuously managed, by clear engagement (at what point, and what service will they provide), and with clear supply management (e.g. allocation of a nominal 1 day/week for 3 weeks to project X). And just as important, it's critical to run the function as a service so they do not become a constraint or critical path dependency on any individual project.

SUMMARY

It's actually quite simple. Start by defining what your success measures are. Increase your throughput by eliminating capacity constraints, speed up time to market by optimising your critical path activities, smooth the peaks so your "pipeline" is running at maximum flow all the time, create independent pipelines for specific types of change, and optimise the individual activities to reduce waste/increase productivity. Easy hey? Of course it's hard in practice, but if you can help your teams understand the basics via a simple model, then it becomes a lot more straight forward.

Throughout this article we've mentioned a number of "methods" – DevOps, Agile, Lean Startup – but the article is very purposefully not advocating any particular way of working – instead, focus should be on the outcome rather than the method. With a mindset of considering your delivery function through an engineering or manufacturing lens, it becomes easy to focus on the right areas.

Interested in finding out more? Please contact

Ollie Holden
Director Of Change Capability Improvement
Email: ollie.holden@projectone.com