POCKET-SIZED INSIGHTS FOR SOFTWARE TEAMS

TEAM GUIDE TO SOFTWARE RELEASABILITY

Manuel Pais & Chris O'Dell

4

conflux

Team Guide to Software Releasability

Manuel Pais and Chris O'Dell

This book is for sale at http://leanpub.com/softwarereleasability

This version was published on 2021-06-24

ISBN 978-1-912058-61-7

conflux

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1. *Team Guides for Software*

Pocket-sized insights for software teams

The *Team Guides for Software* series takes a *team-first approach* to software systems with the aim of **empowering whole teams** to build and operate software systems more effectively. The *Team Guides for Software* books are written and curated by experienced software practitioners and **emphasise the need for collaboration and learning, with the team at the centre**.

The books focus on specific techniques and approaches that have been proven to work well for teams building software systems, and contain **several case studies** from people in the field, bringing to life the concepts in real situations. You can **use the chapter headings like a roadmap** or backlog of things to adopt and improve as a team, and each chapter begins with an overview of the key points, making it **straightforward to adopt the techniques and practices**.

Titles in the *Team Guides for Software* series include:

1. *Software Operability* by Matthew Skelton, Alex Moore, and Rob Thatcher

2. *Metrics for Business Decisions* by Mattia Battiston and Chris Young
3. *Software Testability* by Ash Winter and Rob Meaney
4. *Software Releasability* by Manuel Pais and Chris O'Dell

Find out more about the *Team Guides for Software* series by visiting: http://teamguidesforsoftware.com/

2. Foreword

(This chapter is not available in this edition of the book.)

3. Introduction

Would your organization survive an estimated loss of over $100 million over a single weekend?

That was the initial estimated direct cost after an accidental shut-down of British Airways IT systems (Butler2017) in May 2017 that brought down most of their IT systems and left 75,000 passengers stranded worldwide for a couple of days. British Airways' share value also plummeted by over $170 million.

Besides the discussion on what led to the disaster (initial blame fell on a power surge (Hern2017) and how it could have been avoided, we are interested in the fact that it took British Airways over 2 days to bring their systems back up and running.

Did they not store artifacts released into production in a secure and readily available repository? Did they not regularly provision (manually or automatically) a minified replica of their production environments?

We can only speculate but the key here is that every organization should be able to positively answer these and other questions, if they want to be resilient and recover quickly from disaster. Coincidentally, the same practices that help with recoverability also support faster and safer delivery.

In today's world, being able to quickly release changes to our software (including infrastructure updates), either incrementally or from scratch, is mandatory for survival.

3.1 What is software releasability?

Software releasability is the capability to release changes to our IT systems with minimal delays, 24x7. To achieve software releasability, we must design and evolve the delivery system to continuously improve reliability and reduce time to feedback for everyone involved in delivery - from developers to testers, operations, security and business owners.

A resilient delivery system allows us to recover swiftly from mistakes and even disasters in our production systems, safe in the knowledge that our pipelines will be up and running, feedback will be provided in a timely fashion, and, crucially, any production system can be fully rebuilt from scratch (notwithstanding production data issues).

Increasing software releasability encompasses an adequate pipeline design - including its evolution over time, a reasoned choice of toolset, version controlling all the intervening pieces in a pipeline, and caring about the operability, scalability, security, testability and usability of our delivery system.

3.2 What constitutes a delivery system?

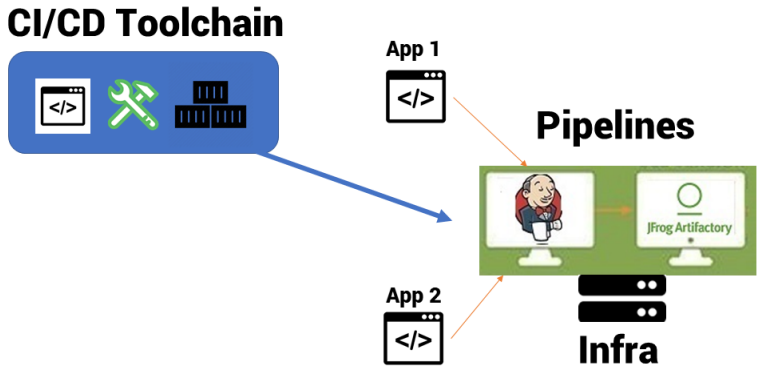
A delivery system encompasses all the tooling, configuration and practices required to get a change from idea into our customer's hands, by progressing all the required artifacts through a delivery pipeline. Such a system needs to evolve along with both the software technology and the organization's processes.

In particular, the delivery system includes at least these components:

- Continuous Integration (CI) tool

- Pipeline orchestration tool, also known as a Continuous Delivery (CD) tool
- CI + CD infrastructure
- Orchestration plugins and 3rd party tools
- Pipeline definitions
- Source code repositories

Throughout this book we will be referencing and giving examples of the above components.



An example delivery system with CI/CD/3rd party tools in container images and respective configuration code under version control (top left box with blue background), source repositories including pipeline definitions for two apps and, finally, the running toolchain (Jenkins and Artifactory inside containers) over the CI/CD infrastructure.

3.3 What does resilient delivery feel like?

The best way to describe what resilient delivery means is to highlight how it feels.

Everyone involved in product development today, from developers to testers or product owners, is under pressure to deliver results

as soon as possible. Fast feedback is paramount to validate results. Therefore a resilient delivery means that upgrading to the latest major version of a pipeline tool, adding a new plugin or changing the pipeline configuration does not stop delivery or delay feedback for hours. **Changes to the delivery system itself are deployed transparently and quickly roll backed in cases of failure** (applying patterns such as immutable infrastructure (Stella2015) and blue-green deployments (Fowler2010), which we cover in chapter 5). There is no need for maintenance windows or downtime in our delivery system or after hours updates.

A resilient delivery system **gracefully handles peak load of build and pipeline runs, while maintaining an efficient resource usage**. This has all to do with scalability, and is especially relevant in organizations with a large number of teams which make capacity planning for CI/CD challenging. There are several ways to go about this, from distributing the delivery system itself, to auto-scaling with cloud resources (see chapter 4). The point is to prevent queued builds and pipelines that force teams to wait for feedback due to unexpected lack of capacity (Christian Deger calls this the "deployment pipeline elasticity" (Deger2018a)).

Issues with CI/CD infrastructure or tools during regular operation are detected and handled swiftly in a resilient delivery system, with adequate (aggregated) logging, monitoring and alert mechanisms in place. Impact on development teams, while not 100% avoidable, is at least greatly minimized. For instance, a build does not fail due to lack of disk space because resource usage is being monitored and alerts get triggered when a defined threshold gets hit. Anomalies are dealt with before they become blocking issues for the teams. Chapter 6 covers these aspects.

Resiliency means that **disaster recovery is possible (almost) at a click of a button, at least in terms of getting systems ready to deploy again**. With nothing but source repositories for setup and configuration of our CI/CD toolchain and infrastructure - as well as the applications repositories themselves - we can recreate the entire

delivery system from scratch on a regular basis, not only after a disaster. Supporting practices here include (applications) pipeline-as-code and CI/CD infrastructure as code, and others described in chapter 2.

Perhaps the greatest benefit of a predictable and reliable delivery system is to remove the unnecessary stressful situation of not knowing if/when we can recover from disaster. It's not hard to imagine the high levels of stress and management pressure on the engineers working to bring BA's systems back to normal operations. Avoiding teams and individuals from suffering burnout is more valuable than anything else.

3.4 Warning signs of software delivery debt

Strong software releasability capabilities are key to rapid and reliable delivery of modern software systems. However, it's easy to underestimate its importance until a failure of serious consequences takes place. Like testability or operability, releasability is a "silent enabler" for delivering and running software effectively:

"Low drama flow doesn't look like progress to most people" - John Cutler (Cutler2017)

The analogy here is that a low drama software release looks effortless, just changes flowing smoothly through the delivery pipeline. But without putting in the necessary work on an on-going basis, releases become painful, undesired but necessary procedures - like going to the dentist (for many of us, at least).

An ad-hoc, "as needed" approach to releasing software can work temporarily in a small startup or engineering department where everyone in the team is more or less abreast of how everything works. But changes in the software, technology and tools quickly pile up and any team, no matter how brilliant, will eventually find it

impossible to keep the entire delivery process in their heads. Issues start to mount because of the increasing number of moving parts in our delivery system, consistency and reproducibility takes a hit and more of developers' time gets spent fixing them rather than working on the software itself.

Some warning signs that the delivery approach needs re-thinking include:

- time from committing changes until they are deployed to production has increased significantly
- dependency conflicts (for instance different teams depend on different versions of the same component in the same environment) becoming more frequent
- drifting release processes between teams (a new engineer in the team has to re-learn how to release)
- most commits are at end-of-day to avoid wait time during the day, leading to the pipeline being broken in the morning more often than not
- issues in the CI/CD toolchain or infrastructure detected by developers first, and taking a long time to diagnose
- having to rebuild artifacts for a redeploy because an artifact management repository and policy are not in place
- failing manual steps that only one engineer knows how to fix
- no simple method for determining if a change has been released to production or is still "in the pipeline"

These signs of an ad-hoc approach to software releasability while rare at first, tend to manifest and bundle together quickly to a point where delivery becomes extremely slow, causing revenue and/or reputation loss for the organization.

Lack of software releasability is akin to drinking a Kwak beer, it flows fine until reaching the tipping point in the middle. If not careful, things will end up messy.

3.5 Why invest in software releasability?

Software releasability has costs, no doubt. Infrastructure (especially to support scalability), people (we cover different organization models for CI/CD in chapter 8) and tooling all have costs.

But investing the necessary CI/CD capabilities means we save the time and effort (usually taken out of product development) previously spent on things like fixing the pipeline, restoring dependencies, searching for artifacts, or rolling forward deployment fixes.

On the customer side, new releases become an eventless routine, instead of a scheduling nightmare with consecutive delays. Downtime during release is minimized meaning higher availability and less disruption to the business and/or end users. Also, roll backs in case of blocking issues become a straightforward process.

3.6 Relationship to Continuous Delivery

This book wouldn't exist without the ground breaking work in the "Continuous Delivery" book by Dave Farley and Jez Humble (HumbleFarley2010), a compendium of good practices around building, testing, deploying, and managing software. In fact, software releasability is one of the outcomes of fully implementing continuous delivery practices and principles (you can find a list of the book's chapters and practices that can serve as a checklist for your continuous delivery adoption at Skelton2016).

You could look at the practices and patterns described in this book as an extension of the "Continuous Delivery" (CD) book. We've seen them work well across different clients and we hope this book provides a contextualized approach for teams adopting continuous delivery.

The deployment pipeline is a key technique introduced by Farley and Humble. Since the book came out in 2010, pipelines and the tooling around them have evolved significantly as more and more organizations adopted CD. This book is fundamentally about the sustainability part in Farley and Humble's definition of Continuous Delivery:

> "The **ability to get changes of all types**, **into production**, or into the hands of users, safely and quickly **in a sustainable way**"

We believe sustainability of the delivery system is a key enabler for speed and safety in the deployment pipeline.

Our aim with this book is in part to collect the patterns and anti-patterns we've collected from helping customers move to CD and experiencing their benefits and their pain, respectively. A core pattern here is to focus first on goals, principles and practices and later on automation and tooling.

3.7 What this book is (not) about

This book focuses on good practices around releasing software that emerged from both our consulting work with multiple clients and industry experience. Although it touches on multiple aspects of software delivery, it does not specifically address the following:

- How to write good build scripts (or Makefiles for that matter)
- Software operability and testability (if you're interested on those topics please see the dedicated books in this book series)
- Defining the contents of a software release (this is highly contextual, our only advice is to keep them as small as you can)
- Deciding whether a given software release is a "go" or "no go" (again highly contextual, we only recommend asking yourselves "how confident are we with the testing we've done?")
- How to build C#, Java or applications in any other stack (there are plenty of resources available out there, we would not be adding anything new)
- Change management processes (although we might have a suggestion or two if your delivery is being held back by slow change approval boards)
- Deployment strategies (we recommend reading chapters 6 and 10 of the "Continuous Delivery" book and look at examples of how other organizations implemented said strategies)

3.8 How to use this book

Each chapter is readable independently, containing the necessary level of detail to be understood and actionable on its own, without requiring any of the other chapters in the book to be read first (although certainly reading the full book will provide a more

comprehensive understanding of the concepts and practices and their inter-relations).

Chapter 1 explains why we need to consider our pipelines and the delivery system as a product, rather than just another (set of) tools. Focusing on operability (as detailed in SkeltonThatcher2018) is fundamental for safe and rapid delivery today.

Chapters 2 to 6 detail the benefits and techniques for ensuring the delivery system is: recoverable, operable, scalable, testable, usable, and measurable.

Chapter 7 explains why mapping the value stream activities in the pipeline is crucial for reducing time to deliver and promote global optimization of the delivery flow, rather than local.

Chapter 8 covers different team structures supporting software releasability and the pros and cons of each.

3.9 Feedback and suggestions

We'd welcome feedback and suggestions for changes: please contact us at publications@confluxdigital.net, via @ReleasabilityY on Twitter, or on the Leanpub discussion at https://leanpub.com/SoftwareReleasability/f

Chris O'Dell & Manuel Pais

4. Treat your pipeline as a product for resiliency and fast feedback loops

Key Points

- **Your pipeline is an extension of production** because it is the only route to reliably releasing your code.
- Development teams are your pipeline customers (among others). They need the **delivery system to behave like a customer-facing system: reliable, responsive and non-intrusive**.
- **Treating the pipeline as a product requires on-going maintenance and evolution**. Invest in build and release capabilities and allocate adequate effort to run a reliable delivery system.
- **Measure your delivery performance so you can let the data drive (global) optimizations**, instead of best guess (local) optimizations.
- **Apply production level monitoring and logging** to delivery to minimize issues and downtime.
- **Scale both your delivery infrastructure and practices**. Pipeline needs will change as you grow.

Your delivery pipeline encompasses everything that must be done in order to release an application change. This includes code changes, infrastructure changes, and configuration updates. A pipeline is the orchestration of activities, whereby some of them require environments for testing, QA, up to production.

4.1 Make your pipeline the single route to production

The tools used in most delivery pipelines are treated poorly - poorly maintained, patched, backed up, etc - and yet, if these tools were to break down, a company's ability to release software would be brought to a halt. Tools are also not regularly updated, missing out on useful new features. Often because no-one feels responsible for them. It is shocking how many forward thinking, diligent teams, where emphasis is put on code quality, can neglect the suite of tools they so heavily rely upon.

As the pipeline is your route to production, it becomes a part of production as well and all its parts should be treated as such. Pipeline configuration should be in source control, secrets should be secured, and backups taken regularly. Another activity that takes place in production is monitoring and logging. They should also be present in your pipeline tooling.

4.2 Your pipeline is now a product: invest in it

High performing teams deploy changes to production multiple times per day (ForsgrenHumbleKim2017). They rely on the delivery pipeline to build, secure, test and validate those changes before deployment. The pipeline is the system through which work flows into the customer's hands.

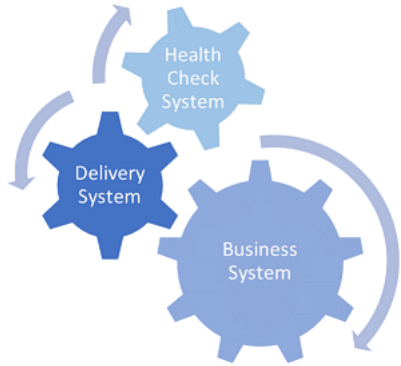
In other words, the software delivery system has now become an essential product for internal use in any organization producing modern software. Because the delivery system is often a composition of 3rd party tools (vendor or open source or a mix) with some "glue code" (scripts), configurations and customizations, it is tempting to look at it as just another set of tools, not as a full-fledged system that provides a critical service to the organization (Christian Deger recommends thinking of it as "CD-as-a-Service" (Deger2018b), an approach taken by ING Bank (Romano2017) for example).

Instead, we should be thinking of modern IT as running at least three types of interconnected systems (inspired by a conversation with Mirco Hering (Hering2018)):

- customer-facing business systems (we include here internal customers as well)
- operations-facing runtime health check systems (including monitoring, alerting and repairing)
- development-facing software delivery systems

When we say "operations-facing" or "development-facing" we're not referring to particular roles but to anyone performing operational or development tasks.

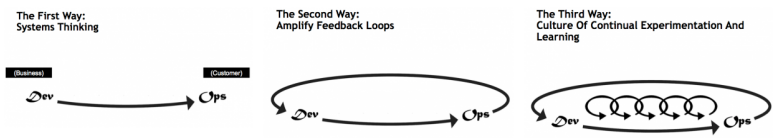
All of them need to be managed as critical systems for the organization, not just the customer-facing apps or services. Without reliable runtime health checks, we won't be aware that a customer-facing system is facing issues until customers are complaining. Without fast and safe delivery, we can't put out fixes predictably and with confidence.



Interdependent systems that make up today's critical IT landscape

Having these systems in place is a fundamental enabler of the first two of "The Three Ways of DevOps", as outlined by Gene Kim (Kim2012):

- **Systems thinking (first way)**: the delivery system connects dev to ops by visualizing status of changes and highlighting bottlenecks between commit and deploy to production.
- **Amplify feedback loops (second way)**: the runtime health check system provides critical feedback on the performance of the commercial systems back to development.



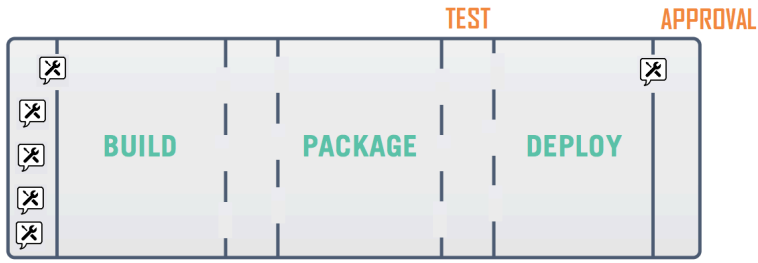
The three ways of DevOps

(*Note*: the third way is about establishing a culture of continual experimentation and learning. While systems can help, organizational culture (Westrum2004) and psychological safety (Delizonna2017) are the key enablers here)

4.3 Avoid simply retro-fitting CD into a CI server

In many organizations, teams installed Jenkins (or its cousin Hudson) in order to build their applications in a central server, in a clean environment that was not subject to the "works on my machine" syndrome. In some cases, this backfired into the "it only builds on the CI server" syndrome, but overall CI servers helped make the build process more predictable and traceable. While that's only one of multiple steps needed to achieve continuous integration (Fowler2006), it helped many teams move a bit faster.

With the advent of Continuous Delivery those teams typically extended their use of the CI server from running a set of jobs to a structured pipeline that orchestrates a set of activities required to validate and deploy a change to production. However, it fell short of actually adopting continuous delivery. Often developers took initiative to adopt new plugins or even tools that supported the pipeline. Rushed fixes prevailed because there was never effort allocated explicitly to the evolution of the delivery system.

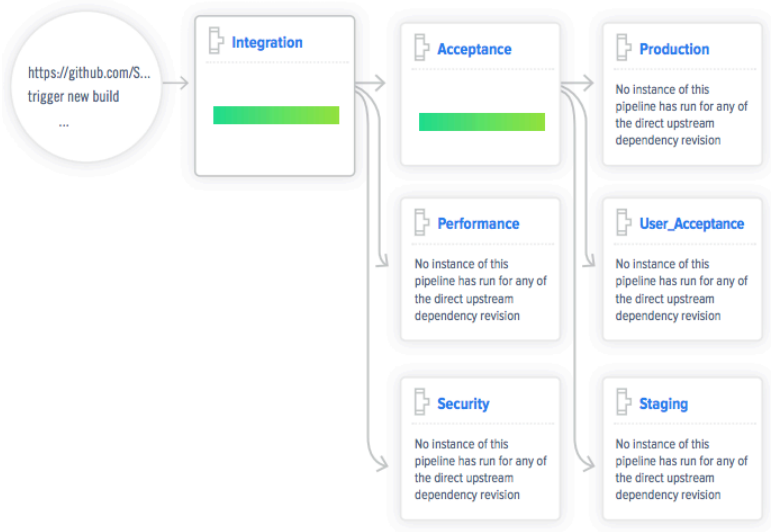


Anti-patterns: build requests queueing up (left), unclear borders between activities, manual steps (testing, approvals) not part of the deployment pipeline

All this gets compounded by the fact that our pipelines include more and more activities (infrastructure updates, shift-left security testing, and so on). Execution time is already long, teams cannot afford adding wait time or downtime due to half-baked solutions

in the delivery system.

The problem is not thinking about requirements or pipeline design. It can be ok to use simply Jenkins with plugins if that fits the requirements. It's important to regularly reassess the requirements, ensuring that we use the best tool for the job and avoiding stretching tools beyond their purpose. Nevertheless, at some point it becomes important to consider adopting tools that natively embed CD concepts as these will tend to make it easier to adopt the CD practices.



Pattern: full delivery workflow mapped in the pipeline, clear separation between activities, (short and wide) pipeline design allows risk-based options for faster delivery

In the next few sections, we will cover some important yet often overlooked approaches for an effective delivery system that promotes fast and safe delivery of changes.

4.4 Measure delivery to visualize flow and identify bottlenecks

It's natural to focus our attention on the build-test-deploy activities in the delivery pipeline. The whole team can immediately see that the latest build took 13 minutes to complete or that deployment to staging failed.

These are recurring, easy to observe results around automated activities in the pipeline. But how long did it take for someone to look into that failed deployment? What if it failed because a port must be opened in the staging and that can only be done by the IT security team sometime within the next couple of days?

We don't usually track the answers to these other questions which are not directly provided by the pipeline.

But if we want to deliver changes faster and reduce lead time we need to have a good picture of all the activities that take place and, especially, the wait times getting a change into customer's hands. Let's walk through an example.

The team is working on this new feature that requires a new port for communication. This feature might require 20 to 30 pipeline runs until ready for production. If the team is able to reduce the build time from 13 to 10 minutes then we'd be reducing lead time by up to 3 minutes x 30 runs = 90 minutes.

Once the feature progresses through the pipeline, we stumble against that deployed failure to staging because of the missing open port. This is a one time fix (by the IT security team) but will take between a couple of hours and a few days, depending on the security team's availability. Removing the dependency on the staging environment - for example, by dynamically provisioning ephemeral acceptance environments - would have reduced the lead time for this feature by at least 2h (33% more than the build time improvement).

Often there's also no direct correlation between cost of the improvement and the benefit it brings. Simple changes can have a much higher impact in lead time than complicated technical changes. However, there's a tendency to prefer those technical improvements that do not require approval or buy in from peers or managers. But we need to leave our comfort zone to make any significant progress.

And we need to measure to understand the highest bottlenecks in our delivery.

Chapter 7 details the practices for measuring delivery by mapping all the activities involved in the pipeline.

Another aspect to consider are trends and deviations. Looking at discrete pipeline metrics (latest build or test execution time, for example) without keeping in mind historical trends can lead to acceptance of an increasingly longer lead time. For example, if a suite of automated tests takes 1h to run in our pipeline today, and we increase it by 5 minutes (perhaps because we're adding more tests for a new feature) we will probably find it reasonable. And if this happens again next week also. And next week. After only a month, we've now increased the test run time by one third! We need trends to put metrics in perspective and highlight deviations over time.

Chapter 6 talks about identifying and tracking metrics for continuous delivery, as well as thresholds and alerting on deviations.

4.5 Design the delivery system to evolve with your needs

When setting up the delivery system there are a multitude of choices along the way. And your requirements, like with any customer-facing system, will change over time.

We recommend first of all focusing on (and documenting) the practices that the team wants to promote. Take for example the practice that all artifacts built should be stored for a set period of time:

- 1 day for all artifacts that did not get deployed to staging
- 1 week for all artifacts deployed to staging
- forever for all artifacts deployed to a live environment

A tool with artifact management capabilities will come in handy. Some initial functional requirements for this tool could include separate repositories per application, artifact promotion (for e.g. after deployment to staging), and of course defining the retention policies.

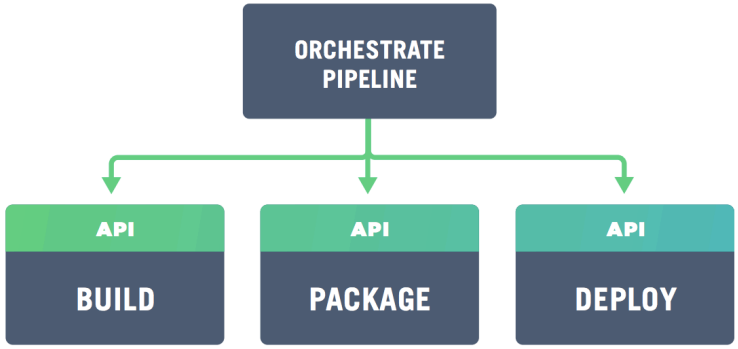
Now imagine you've containerized your application and you want your pipeline artifacts to be container images. Ideally, you would have included support for multiple binary formats, including container images, in the initial requirements. But chances are that this was not an anticipated move at the time.

After some research, the team concludes they need to migrate to another tool that supports the original requirements plus the new multi-format requirement. This should not be a headache in a delivery system that is maintainable and easy to change.

Choosing tools that support pipeline evolution means considering their (inter)operability capabilities. To begin with:

- does it expose features via an API (or at least an easy to install command-line interface)?
- does the API cover the entire feature set, not just a subset?
- does it provide access to logs?
- does it have good error handling and expose it in the logs?

In particular, the integration of disparate tools in our pipeline becomes much more maintainable when API-driven. If both the new and old artifact management tools support the operational requirements above then swapping them should be a matter of changing from the old to the new API calls (migrating existing artifacts is possibly more complicated, but the APIs should help if there's no easy export/import mechanism).



An example pipeline where all the orchestrated tools can be driven via an API.

The requirements for logging access and error handling support log aggregation and collecting metrics, as you would in a production system. We can dig deeper when tooling issues arise and we can look for trends related to build and other activities timings, failure rates and so on. In short, there is a wealth of tools in this space so it is important to pick ones which aid your process and help recover from failure, not hamper evolution by being a closed system.

Any delivery system - even if based on cloud managed services - needs some "glue code" and configuration to orchestrate different

activities/tools at some point. The less code you need to maintain the better. But especially with self-managed delivery systems (even if tool agents and environments run in the cloud) the amount of pipeline code keeps increasing. We need to watch out for the same code smells (duplication, coupling, magic numbers, to name a few) and apply the same good practices (refactoring, single responsibility principle, design patterns) and code analysis (especially security wise) to ensure we keep that code clean and easy to extend. Avantika Mathur referred, jokingly, to a "Script-ocalypse" as the proliferation of poorly written scripts that become more and more costly to maintain (Mathur2018).

4.6 Apply monitoring and logging to minimize issues and downtime

Monitoring is the act of collecting real time data about your system and possibly alerting people when values go above, or below, a threshold. This includes, among others, low level monitoring of resources such as CPU load, remaining disk space and memory. You would never run a production system without these basic checks. As your pipeline is now a product and an extension of production, these checks should also be run on it.

We talk about monitoring in chapter 3 (monitoring the health of the delivery system) as well as in chapter 6 (monitoring CI and CD metrics, thresholds and deviations).

Much like monitoring, no production system should be run without logging - diagnosing and debugging issues depends on it. In the case of a pipeline, the logs will be coming from the automation tools themselves. As a pipeline is generally composed of multiple tools running on multiple instances, log aggregation is required to make sense of what's going on when the tooling breaks.

Log aggregation can help both the investigation of issues with the tools themselves (for example a new API version that returns a slightly different response to a call, breaking our delivery) or with the actual pipeline activities by providing more in-depth information on obscure failures.

Chapter 3 exemplifies how to include log aggregation in your delivery system and how to detect issues early.

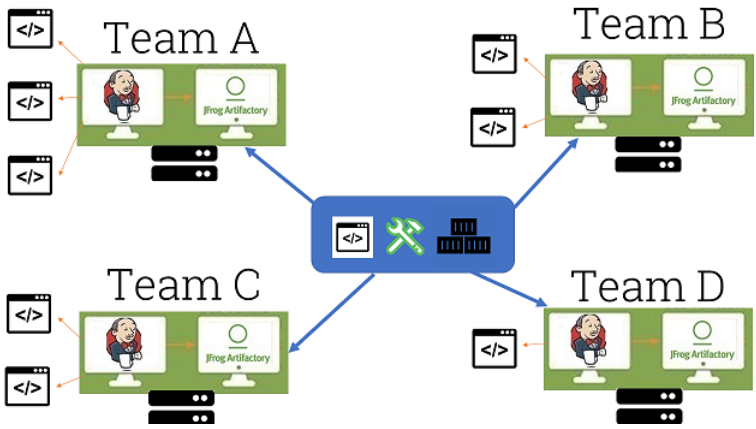
Resiliency in software delivery requires continuous monitoring and alerting to detect failures early, just like business systems do. Issues that escape monitoring are good candidates for new tests or alert conditions on the delivery system.

4.7 Scale the infrastructure to avoid pipelines queuing up

As with customer-facing systems, when the usage grows the system must also scale, which generally requires a redesign of the architecture. The pipeline for a single team or a small number of teams does not look the same as a pipeline for a hundred or more teams. When scaling a production system it is important to identify your blockers so that efforts on scaling are targeted at the right place. The same applies for your pipeline (for more insights see "theory of constraints", as introduced in GoldrattCox2014).

The most straightforward aspect that will need to scale with growing usage is the infrastructure. Ideally, if we want to prevent build and pipeline queues (as well as reduce execution time via parallelization), we need an elastic CI/CD infrastructure that can (auto-)scale up when necessary, and scale down. When using cloud infrastructure, that becomes straightforward as elasticity is a core selling point for cloud providers.

However, most organizations we've consulted for still prefer to run their delivery system on-premise. In this case, running pipeline agents in ephemeral containers (that live only during a given pipeline execution) can achieve a more efficient resource usage when compared to virtual machines or actual physical machines being used as agents. Still, this approach can only take you so far.



A delivery system where each team is responsible for their own delivery infrastructure and practices, but still can benefit from a centralized infrastructure and toolchain setup, as well as default pipeline definitions.

Monitoring low level resource usage (CPU, disk and memory) in your CI/CD infrastructure, and alerting when certain thresholds are reached, are the first steps to trigger a re-evaluation of the system capacity to cope with current usage. Again, trends are important here. 90% CPU usage might be an anomaly if it never passed 80% for the last 3 months, or it could be a sign of danger if usage has been steadily increasing in the last 3 months.

Chapter 4 details some techniques for scaling the CI/CD infrastructure, namely infrastructure as code, agent farms, or ephemeral agents in containers.

4.8 Scale the practices and pipelines to support growing usage

A less obvious concern when scaling CI/CD to a growing number of teams and users is the capacity to quickly onboard new teams and quickly spread good practices to all the teams already using the system.

Pipeline-as-code, in similar fashion to infrastructure-as-code, allows defining all the steps, parameters, sources, dependencies and any other aspects of a given pipeline. This means that the pipeline can be defined before it's actually created. Modern CD tools can search for pipeline definitions in a given repository and instantiate them, and update them every time a change to the definition is committed.

Besides reducing the chance for human error when configuring pipelines via the CD tool's GUI, pipeline-as-code offers other far reaching benefits:

- creating a full pipeline from zero for a new application in a matter of minutes, by leveraging pipeline definitions with similar technological stack and delivery requirements
- promote reuse by parameterizing and pushing code to execute the pipeline stages down to be shared by multiple pipelines
- clearly assign the responsibility for defining the pipeline to the team that delivers the application (more on team organization patterns and responsibilities in chapter 8)
- allowing teams to share good practices by merging pipeline code (this requires making the practices explicit and decoupled from specific tools as much as possible)

Chapter 4 provides concrete examples and guidance on how to adopt pipeline-as-code for scaling CI/CD across teams.

Focus on making practices easy to understand and easy to adopt, rather than enforce standards. Reasonable policies (such as not deploying artifacts to production with known vulnerabilities) can be mandatory without enforcing specific practices.

4.9 Care for pipeline testability and usability to encourage adoption

Testability and usability are as important for a delivery system as for a customer-facing system. Testability helps reduce downtime due to untested changes to the pipeline and tools. Usability makes the pipeline actions and data accessible to a wider range of users, namely business owners.

It is all too common to find the delivery system that the application teams rely on being directly modified (for example, adding new tools), without prior testing. This has a high probability of leading to failures and possibly downtime. There is no reason not to run a CI/CD development environment (a scaled down replica of the delivery system) for testing those changes in isolation beforehand (further made easier when the CI/CD infrastructure is already codified).

By leveraging pipeline-as-code and creating pipelines for simple "Hello World" applications that mimic the actual application pipelines we can perform a sort of "smoke tests" of changes to the delivery system. This should include deployment to a production environment, knowing that the presence of the example app is harmless.

We are not done after testing in isolation though. By deploying with a blue-green approach we can then monitor for instabilities in the pipeline, and rollback to the previous version if necessary.

Simultaneously reducing downtime and impact on the pipeline users.

4.10 Build security into and around the pipeline

Unfortunately, we can't simply monitor security and wait to take measures when some threshold is reached. Very much like testing, we can never say security work is complete or that our system is fully secure.

Security needs to be a continuous activity and the earlier we embedded it in the software lifecycle the better. That is of course one of the core premises of DevSecOps (Riley2017), which focuses on building security in the pipeline.

Some straightforward techniques that we can and should be using today to improve the security of the applications we're delivering and their dependencies:

- relying on a mature secrets management tool
- minimizing attack surface by keeping OS images as small as possible
- scanning for vulnerabilities at the last responsible moment (before deployment)
- restricting access to production ecosystem from the delivery system (pull new artifacts, don't push)

But we also need to consider the security of the delivery system itself. Are we hardening the underlying infrastructure? Are we isolating and using minimal OS images on the build agents? Do we regularly recreate the underlying infrastructure not only as proof of disaster recovery, but also to eliminate potentially unauthorized access? Are critical activities in the pipeline protected by role-based access?

While we might not be able to address everything at once, the key is to keep security in mind. Start small but make security a key concern in the design and evolution of the delivery system.

This is a wide subject that could easily be the subject of an entire other book by security experts. Nevertheless, in appendix A, we give you some pointers on how to get started with security inside and around the pipeline, including concrete examples for some of the techniques mentioned above.

4.11 Get started!

1. **Identify your software delivery system**. What are all the tools, plugins, pipeline definitions, code repositories, and infrastructure required to deliver software today?
2. **Document your current and future practices**. Take a step back to extract what are the key practices in your pipelines today, besides the traditional build-test-deploy. How are artifacts managed and retained? How is infrastructure provisioned? Do you create dynamic acceptance environments? Establish a baseline of what are critical practices today and start defining what are the desired practices to adopt.
3. **Adopt pipeline-as-code**. Most CD tools now support configuring (YAML, JSON) or coding (Groovy, Java) all the steps and dependencies for any given pipeline. This relatively low effort step can dramatically increase predictability, ease bootstrapping of new pipelines, and scale new practices across multiple teams.
4. **Start moving to CI/CD infrastructure-as-code**. If this infrastructure is still running on snowflake servers and/or agents, start by documenting it (most configuration management tools have utilities to reverse engineer a server) and then

take small steps to automate a replica of that. Don't go all in with any given tool, experiment a few and get a feeling for which best fits your requirements and skills.

5. **Start gathering core pipeline metrics**. You don't want to go crazy and start tracking all sorts of metrics. But there are core analytics you can quickly start measuring like cycle time, execution time for each activity, and, if possible, wait times (when you have manual steps in the pipeline). These will be building blocks for later analysis of trends and deviations.

4.12 Summary

The deployment pipeline and its tooling are an extension of your production system and require similar levels of product ownership, maintenance and product development. As with a customer facing product, there are tasks that you would not imagine skipping, such as using source control or monitoring server health, and this attention to detail needs to be extended to the pipeline too. The increased visibility and traceability will greatly improve the development and deployment experience and the focus on monitoring and logging will help keep the pipeline healthy and recover quickly from tooling issues.

5. Ensure delivery system is recoverable to endure disaster

(This chapter is not available in this edition of the book.)

6. Ensure delivery system is operable to minimize downtime

(This chapter is not available in this edition of the book.)

7. Ensure both practices and infrastructure can scale to meet usage growth

(This chapter is not available in this edition of the book.)

8. Care for pipeline testability and usability to encourage adoption

(This chapter is not available in this edition of the book.)

9. Measure delivery to visualize flow and identify bottlenecks

(This chapter is not available in this edition of the book.)

10. Treat your pipeline as a value stream to tackle largest bottlenecks first

(This chapter is not available in this edition of the book.)

11. Organize teams to promote build and release ownership

(This chapter is not available in this edition of the book.)

12. Appendix A: build security into and around the pipeline

(This chapter is not available in this edition of the book.)

13. Terminology

14. References and further reading

14.1 Introduction

Butler2017 - Sarah Butler, The Guardian, 'British Airways could face £100m compensation bill over IT meltdown', 2017. [Online]. Available: https://www.theguardian.com/business/2017/may/28/british-airways-faces-100m-compensation-bill-over-it-meltdown [Accessed: 2-July-2018]

Cutler2017 - J. Cutler, 'Crack The WIP (Work In Progress)', 2017. [Online]. Available: https://medium.com/@johnpcutler/crack-the-wip-work-in-progress-7b0c646a7cf8 [Accessed: 2-July-2018]

Deger2018a - C. Deger, 'Cloud Native Continuous Delivery', Continuous Lifecycle London, 2018. [Online]. Available: https://youtu.be/H0Ae38_-J-y8 [Accessed: 2-July-2018]

Fowler2010 - M. Fowler, 'BlueGreenDeployment', 2010. [Online]. Available: https://martinfowler.com/bliki/BlueGreenDeployment.html [Accessed: 2-July-2018]

Hering2018 - M. Hering, *DevOps for the Modern Enterprise*, 1 edition. IT Revolution Press, 2018.

Hern2017 - Alex Hern, The Guardian, 'British Airways IT failure: experts doubt 'power surge' claim', 2017. [Online]. Available: https://www.theguardian.com/business/2017/may/30/british-

airways-it-failure-experts-doubt-power-surge-claim [Accessed: 2-July-2018]

HumbleFarley2010 - J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1 edition. Upper Saddle River, NJ: Addison Wesley, 2010.

Riley2017 - C. Riley, 'Preventing Security from Slowing Down Software Delivery', 2017. [Online]. Available: https://www.twistlock.com/2017/09/13/pre security-slowing-software-delivery [Accessed: 7-July-2018]

Skelton2016 - M. Skelton, Skelton Thatcher Consulting, 'Continuous Delivery checklist template', 2016. [Online]. Available: http://cdchecklist.info [Accessed: 2-July-2018]

SkeltonThatcher2018 - M. Skelton and R. Thatcher, *Team Guide to Software Operability*, 1 edition. Conflux Digital, 2018.

Stella2015 - Josh Stella, O'Reilly, 'An introduction to immutable infrastructure', 2015. [Online]. Available: https://www.oreilly.com/ideas/an-introduction-to-immutable-infrastructure [Accessed: 2-July-2018]

14.2 Chapter 1 - Treat Your Pipeline as a Product

Deger2018b - C. Deger, 'Cloud Native Continuous Delivery', Continuous Lifecycle London, 2018. [Online]. Available: https://youtu.be/H0Ae38_-J-y8 [Accessed: 2-July-2018]

Delizonna2017 - L. Delizonna, 'High-Performing Teams Need Psychological Safety. Here's How to Create It', Harvard Business Review, 2017. [Online]. Available: https://hbr.org/2017/08/high-performing-teams-need-psychological-safety-heres-how-to-create-it [Accessed: 3-July-2018]

ForsgrenHumbleKim2017 - N. Forsgren, J. Humble, G. Kim, '2017

State of DevOps Report', 2017. [Online]. Available: https://puppet.com/resources/wh
of-devops-report [Accessed: 3-July-2018]

Fowler2006 - M. Fowler, 'Continuous Integration', 2006. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html [Accessed: 3-July-2018]

GoldrattCox2014 - E. Goldratt, J. Cox, *The Goal: A Process of Ongoing Improvement*, 30th anniversary edition. North River Press, 2014.

Kim2012 - G. Jim, 'The Three Ways: The Principles Underpinning DevOps', 2012. [Online]. Available: https://itrevolution.com/the-three-ways-principles-underpinning-devops [Accessed: 3-July-2018]

Mathur2018 - A. Mathur, 'Surviving the "Script-ocalypse" on the Road to Scaling Enterprise DevOps', 2018. [Online]. Available: https://www.youtube.com/watch?v=2rQMrt2H4zQ [Accessed: 4-July-2018]

Romano2017 - D. Romano, 'Global Continuous Delivery in a Financial Organization', DevOps Enterprise Summit London, 2017. [Online]. Available: https://www.youtube.com/watch?v=A8Qwu1bYIO8 [Accessed: 3-July-2018]

Westrum2004 - R. Westrum, 'A typology of organisational cultures', Qual Saf Health Care, 2004. [Online]. Available: https://qualitysafety.bmj.com/conte
2/ii22.full.pdf [Accessed: 3-July-2018]

15. About the authors

15.1 Chris O'Dell

Chris has been developing software with Microsoft technologies for nearly fourteen years. She currently works at Monzo helping to build the future of banking.

She has led teams delivering highly available Web APIs, distributed systems and cloud based services. She has also led teams developing internal build and deployment tooling using the unconventional mix of .Net codebases onto AWS infrastructure.

Chris promotes practices we know as Continuous Delivery, including TDD, version control, and Continuous Integration.

15.2 Manuel Pais

Manuel Pais is an independent DevOps and Delivery Consultant, focused on teams and flow.

With a diverse experience including development, build management, testing and QA, Manuel has helped large organizations in finance, legal, telecom and manufacturing adopt test automation and continuous delivery, as well as understand DevOps from both technical and human perspectives.

Manuel is co-author of the Team Guide to Software Releasability book and lead editor for the remaining books in the Team Guide series.

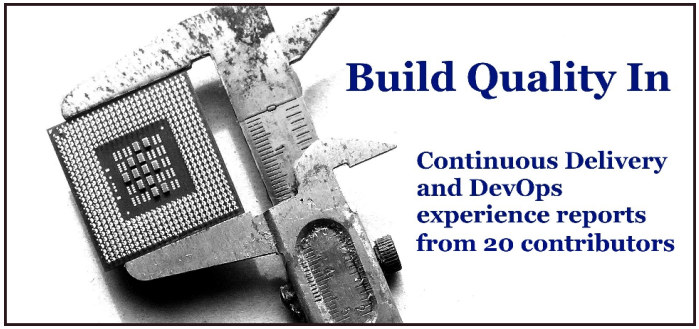
16. Conflux Books

Books for technologists by technologists

Our books help to accelerate and deepen your learning in the field of software systems. We focus on subjects that don't go out of date: fundamental software principles & practices, team interactions, and technology-independent skills.

Find out more about *Conflux Books* by visiting confluxbooks.com

conflux



Build Quality In - a book of Continuous Delivery and DevOps experience reports. Edited by Steve Smith and Matthew Skelton. Conflux Books, April 2015

Internal Tech Conferences by Victoria Morgan-Smith and Matthew Skelton. Conflux Books, April 2019

Better Whiteboard Sketches by Matthew Skelton. Conflux Books, August 2019

Team Guides for Software: *Software Operability, Metrics for Business Decisions, Software Testability, Software Releasability.* Skelton Thatcher Publications, 2016-2019