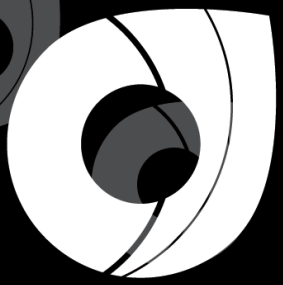
Corndel DevOps Engineering Programme

in association with Softwire

Module 8: Continuous Delivery



Corndel Digital.

DB2F

Introduction

This module builds on the the material covered in Module 7. You've learnt how to test your code using continuous integration pipelines, and how a strong passing test suite, combined with a CI pipeline, gives you confidence that your code is always ready to deploy.

In this module we'll look at the next step, and discover how we can build pipelines that automate code deployment. We'll introduce **Continuous Delivery** and how it enables us to automate code deployments. You'll learn about the positive impacts automated deployments can have on development teams, and organisations as a whole, as they enable a fresh, more agile view of software releases.

You'll learn how organisations set up multiple production-like environments and the advantages this brings, but also the challenges. We will cover how continuous delivery enables us to make the most of complex multi-environment setups, as well as review some of the common environment configurations used across the industry.

Dealing with failure is another key topic. You'll learn how to handle failed releases, why they happen and how you can leverage continuous delivery pipelines to reduce the likelihood of failure and mitigate the impacts.

Finally, we'll introduce **Continuous Deployment** and cover how it, despite the similar name, requires a fundamentally different culture to continuous delivery. We'll review the benefits that continuous delivery can bring, but also note some of the challenges you might face implementing it, both internal and regulatory.

Table of Contents

Core Material

Additional Material

Chapter 1

Continuous Delivery

What is Continuous Delivery?

Continuous delivery (CD) is about building pipelines that automate software deployment.

Without a CD pipeline, software deployment can be a complex, slow and manual process, usually requiring expert technical knowledge of the production environment to get right. That can be a problem, as the experts with this knowledge are few, and may have better things to do than the tedious, repetitive tasks involved in software deployment.

Slow, risky deployments have a knock-on effect on the rest of the business; developers become used to an extremely slow release cycle, which damages productivity. Product owners become limited in their ability to respond to changing market conditions and the actions of competitors.

Continuous delivery automates deployment. A good CD pipeline is quick, and easy to use. Instead of an Ops expert managing deployments via SSH, CD enables a product owner to deploy software at the click of a button.

Continuous delivery pipelines can be built as standalone tools, but are most often built on top of CI pipelines: if your code passes all the tests, the CI/CD platform builds/compiles/bundles it, and its dependencies, into an easily deployable **build artefact** which is stored somewhere for easy access during future deployments. At the click of a button, the CD pipeline can deploy the artefact into a target environment.

A CI pipeline tells you if your code is OK. Does it pass all the tests? Does it adhere to your code style rules? It gives you something at the other end: a processed version of your source code that is ready for deployment. But a CI pipeline does not deploy software. Continuous deployment automates putting that build artefact into production.

Why do we want it?

Continuous delivery pipelines automate deploying your code. This approach has many benefits. Releases become:

- **Quicker**. By cutting out the human operator, deployments can proceed as fast as the underlying infrastructure allows, with no time wasted performing tedious manual operations. Quick deployments typically lead to more frequent deployments, which has a knock-on benefit to developer productivity.
- **Easier**. Deployment can be done through the click of a button, whereas more manual methods might require a deployer with full source code access,

a knowledge of the production environment and appropriate SSH keys! Automated deployments can be kicked-off by non-technical team members, which can be a great help.

- **More Efficient**. Continuous delivery pipelines free team members to pick up more creative, user-focused tasks where they can deliver greater value.
- **Safer**. By minimising human involvement, releases become more predictable. It's harder to make one-off configuration mistakes, and you can guarantee that all releases are performed in the same way.
- **Happier!** Deploying code manually is repetitive and tedious work. Few people enjoy it, and even fewer can do it again and again without error. Teams that use automated deployments are able to focus on more interesting challenges, leaving the routine work to their automations.

Automated deployments reduce the barrier to releasing code. They let teams test out ideas, run experiments, and respond to changing circumstances in ways that are not possible, or practical, in slower-moving environments. This fosters creativity, positivity and product ownership as teams feel that the system enables, rather than restricts, what they are able to do.

Continuous delivery works well in environments where development teams need control over if and when something gets deployed. It's best suited to business environments that value quick deployments and a lightweight deployment process over extensive manual testing and sign-off procedures.

What makes it hard?

Implementing continuous delivery can be a challenge in some organisations.

Product owners must rely on automated processes for testing and validation, and not fall back to extensive manual testing cycles and sign-off procedures. Continuous delivery does involve human approval for each release (unlike *continuous deployment*), but that approval step should be minimal, and typically involves checking the tests results look OK, and making sure the latest feature works as intended.

Continuous delivery requires organisation-wide trust that a comprehensive automated validation process is *at least* as reliable as a human testing & sign-off process. It also requires development teams to invest time in fulfilling this promise, primarily through building strong test suites (see Module 3).

How do we do it?

Teams must be confident in the quality of their testing. You need to trust that every code version that passes your automated tests is a viable and safe release candidate. This is not easy, and requires significant effort both from software developers (who write test cases) and DevOps engineers (who create test infrastructure).

Continuous delivery increases the importance of high-quality test code and good test coverage, as discussed earlier in the course. You need strong test suites at every level (unit tests, integration tests, end-to-end tests), and should also automate NFR testing (for example, performance tests). You also need to run these tests frequently and often, ideally as part of a CI pipeline.

Even the best testing process, manual or automated, will make mistakes. Continuous delivery encourages rapid iteration, and works best when teams work together to get to the root of any failures/bugs, and improve the process for future iterations.

DevOps engineers are central to this effort. They set up automated testing infrastructure, and the CI/CD pipeline as a whole. DevOps engineers are also responsible for monitoring code after the release. Releases will fail, and not always in immediately-obvious ways. It's up to DevOps professionals to set up appropriate monitoring, logging and alerting systems so that teams are alerted promptly when something goes wrong, and provided with useful diagnostic information at the outset. We will come back to the topics of logging, monitoring and alerting later in the course.

Chapter 2

Environments

DevOps engineers want to confirm that their code and build artefacts work before deploying to production. However, it's often difficult to be sure of this on a development machine, which won't look exactly like a production environment. You don't want to end up saying "well, it works on my machine!".

The solution is to have multiple **environments** where you can deploy applications. An environment is a broad term meaning "your software, and anything else it requires to run". That includes infrastructure, databases, configuration values, secrets and access to third-party services.

Most software projects will have at least one environment that is a near-exact clone of the production environment. The only differences are that it won't typically handle live traffic, nor will it hold real production data. This kind of *staging* (also known as *pre-production* or *integration*) environment is a great place to test your build artefacts before releasing them to your users. If it works there, you can be reasonably sure it will work in production. Staging environments are commonly used for end-to-end automated testing, manual testing (UAT), and testing NFRs such as security and performance.

Production-like environments are sometimes referred to as "lower" environments, as they sit beneath the production environment in importance. Build artefacts are promoted through these lower environments before reaching production, either through build promotion or rebuilds from source code.

Some projects will also include less production-like environments that can be used for testing and development. Such environments are usually easy to create and destroy, and will stub or mock some services to achieve this.

Environments and Continuous Deployment

Maintaining multiple environments is very difficult. Configuration drift is common, and near unavoidable if you deploy applications to long-lived virtual machines and/or make manual configuration adjustments as part of the deployment process. Unfortunately, these issues only grow as you spin up new environments. As we'll discuss in a moment, there are advantages to having a lot of environments!

Fortunately, continuous deployment strategies can help. CD works with infrastructure as code and configuration management tools to deploy updates systematically across multiple environments in a reliable, source-controlled fashion.

Continuous deployment also helps manage migrating application versions (and

infrastructure) between environments. Pre-defined build promotion pathways can help clarify and enforce a sensible path to production.

Commonly, CI/CD pipelines are configured to automatically deploy build artefacts into a lower environment, making it easy for a human approver to inspect the working software in a safe, but production-like setting. A good CD setup will provide a clean interface through which users can promote that code version to a higher environment.

Build Artefacts

But what is a build artefact? A build artefact is created from your source code; it's a production-ready version that is easy to run without setting up a development environment. In practice, what the build artefact looks like depends on your programming language and infrastructure. For languages that compile to machine code, a build artefact is an executable binary file compiled for a target platform. For a JVM language, it might be a so-called "fat" JAR file (the application and all dependencies, compiled to Java bytecode). Python is an interpreted language and has no compilation stage. Instead, a build artefact for a Python library might be a `whl` file or simply a tarball of your source code. Build artefacts are not always executable programs - docker images can be treated as language-agnostic build artefacts, for example. You'll try out this approach in this Module's workshop and exercise.

Selecting an appropriate build artefact format is important. This is how you will distribute your app, and there are a few questions you should think through:

- **Who are my users?** Is it just my production VM, or will this be public?
- **Target platforms?** Who can run it? Artefacts might target a particular host platform, or require pre-installed software (e.g. Python 3.7, docker or JRE 8)
- **Are container images OK?** Docker images are a great build artefact for distributing apps targeting container runtimes. Are there users who won't be using Docker?
- **Does size matter?** Most build artefacts can be optimised to reduce output size, but there is sometimes a cost (e.g. losing debug information or code documentation)
- **Security?** What happens if someone tries to extract source code from your build artefact? Is it possible? Does it matter? Have I baked-in any security credentials (hint: you shouldn't!)

Build artefacts might be strictly internal, stored on private infrastructure and deployed to your own production hardware. Other projects may upload build artefacts to a public repository for others to use (e.g. PyPi). Some of these questions don't just affect your choice of build artefact, but also your choice of programming language in the first place!

Typical Patterns

Software projects should use as many, or as few, environments as they need to ensure an efficient software development and deployment pipeline. Naturally, the number and nature of environments will change from project to project, but there are a few common patterns that crop up again and again. These are listed below.

Integration and Production

This is the simplest multi-environment setup. The second environment (integration) is used for checking different services working together, conducting manual testing (e.g. UAT), and provides a place to test out development changes that can't be simulated locally.

The pattern can get a little messy, because the integration environment is used for many things. Testing or development requirements can lead to configuration drift between integration and production, testing interests can block development work and vice versa.

Dev, Staging and Production

Similar to the setup above, this architecture adds a new layer to separate development concerns from manual testing and release validation. The dev environment is a safe space where developers can test ideas and check solutions work, and the staging environment is used for manual testing and other release processes.

This system is not without its issues. Mostly notably, a single shared development environment can be an issue for large development teams pursuing multiple independent workstreams. Nevertheless, this separation is an improvement over a two-environment setup.

Dev, UAT and Production

This setup is near-identical to the previous one. The Staging server is renamed UAT - there is a greater emphasis on manual testing - but the architecture is otherwise unchanged. Build artefacts are promoted from dev to UAT and finally production.

Blue-Green Deployments

Blue-green deployments use two production environments: each takes turn handling live traffic. New releases are first deployed to the inactive environment, and live traffic is gradually re-directed from the second environment. If there are no issues, the re-direction continues until the live environments have switched, and the process repeats on the next update.

Blue-green deployments allow incremental rollouts and support extremely rapid rollback

- traffic can simply be sent back to the original (unchanged) environment. This can be more complicated to setup than static environments, but the benefits make it worthwhile!

Organisations using blue-green deployments may also deploy "lower" testing and development environments.

Branch Environments

In this setup, development environments are not permanent. Instead, the CD platform automatically creates a new environment for each branch. The branch environment exists for as a long as needed, and is deleted once the branch is merged and deleted. This approach gives every developer a sandbox to test their work.

Transient environments of this kind usually make extensive use of containers to easily launch, and destroy disposable infrastructure (e.g. application servers, databases), and usually take advantage of massively-scalable cloud infrastructure to deploy new environments on demand. It requires a sophisticated deployment pipeline, but is well worth the effort for organisations capable of doing so.

Environment Challenges

Managing environments can quickly become complicated. Copying the production environment means doubling the amount of infrastructure you need to manage. That includes provisioning and configuring servers, applying updates, managing secrets and users. It also means creating appropriate datasets to populate the databases on each environment.

We've already discussed configuration management and immutable infrastructure tools (such as chef, puppet and docker). The value of these tools increases with the number of environments - you could get by managing one environment without any automation (although we would not recommend it), but it's very difficult to handle more than a couple manually, and some of the most powerful environment management strategies listed above are impossible without powerful automation tools.

Shared Dev Environments

It's common to have a dedicated, persistent development environment. The rationale is clear enough: developers can't simulate all aspects of a production environment on their own computer; a remote environment owned and managed by the dev team can overcome this. However, this pattern runs into several notable difficulties:

- **Configuration Drift**. Manual adjustments of code and configuration are an essential part of development. On a persistent environment this can quickly cause significant drift from the production environment.
- **Breakages**. Development tasks routinely cause breakages in the code under development. That's OK on a personal environment (e.g. a laptop), but can cause issues on a shared environment. One developer's routine breakages might block the work of another dev.
- **Data**. Development tasks may require specific data on the dev environment. It's common for two developers to each require a database to be in a mutually exclusive state. Workstreams of this kind cannot be developed in parallel on a shared environment.

Instead of a shared development environment, you should consider developer-specific or, preferably, task-specific environments.

Persistent developer-specific environments may sound appealing, but in fact do not resolve the issues above. A single developer working on multiple tasks in parallel will experience the same blockers, and any persistent development environment will experience configuration drift.

Best practices involve automatically creating a new environment per development task. This gives the best of both worlds: developers can work on production-like infrastructure, and multiple tasks can be developed in parallel (either by the same developer, or distributed across a team). This kind of setup requires strong DevOps tooling to automatically deploy, manage and clean up environments. It also places some heavier requirements on your underlying infrastructure - you may end up with a lot of environments! Nevertheless, the benefits are often worth it.

Third-Party Dependencies

Production applications often use third-party services to handle some functionality. For example, many applications offload authentication and authorisation tasks to dedicated services, communicate with a log server or use one of the countless web APIs that expose assorted services.

It's good practice to avoid communication between non-production environments and external services where possible. Such traffic can incur unnecessary costs (for paid services) and pollute production data (for data storage services). If your use-case requires such communication - perhaps you are running full system tests - it's best to have non-production environments use different accounts or credentials to avoid any interaction with production.

Put another way: you should not share production configuration with lower environments.

"Configuration Hell"

Complex software has configuration options. Configuration options provide applications with data that may change between deployments (e.g. database connection information, API tokens). In contrast, application code should not know (or care) where it is deployed.

Because configuration is deployment-specific, it should not be checked into source control alongside your application code. It is not a re-usable part of an application.

It's generally good practice to store configuration options via a programming-language and system-agnostic method:

1. A configuration file (e.g. `.yml`, `.toml`)
2. Environment variables

Configuration hell is a common term for what happens when bad configuration management negatively impacts software usability and makes deployments tricky or brittle. There are many flavours of configuration hell, there are a few common causes:

- Config is drawn from many locations
- Config options are not granular
- Config options overlap
- Config options have unclear interactions
- Config handling does not scale well to new options and/or environments

In contrast, good configuration options:

- Are granular
- Are independent

You can read more about good configuration practices at the 12-factor app.

Data

Application data can be one of the trickiest, and most time-consuming, aspects of environment setup to get right. Most environments hold data, either in a traditional relational database (e.g. a postgreSQL server), or other storage tool. Applications running in the environment use this data to provide useful services. Some environments don't store any data - they are *stateless* - but this is rare. Most DevOps engineers will need to think about handling application data cleanly.

Production environments hold 'real' data, created by the users of your service. Production data is often sensitive, holding both personal contact information (e.g. email addresses) and financial details (e.g. bank account numbers). Handling personally identifiable data and/or financial data is often regulated, and the last thing you want is a data breach, or to send real emails to real

Data Anonymisation

We often need copies of production data for testing, development and analytics purposes. However, it is extremely bad practice, and often illegal, to disseminate production data outside of a secure production environment. Such datasets could include private financial and contact details, or other sensitive business assets.

Instead of distributing production data, DevOps engineers might automate an anonymisation process. To do this you need to decide:

- What data do I need to anonymise?
- How should I anonymise this data?

The idea is that you can remove sensitive or personal information from a dataset, rendering it safe for wider distribution (e.g. into a development environment).

What exactly you need to anonymise will depend on your organisation, but might include, for example, customer names and addresses. You can anonymise data by replacing it with dummy values (e.g. set all customer names to "John Doe"), or by mutating the data in such a way that it is unrecoverable.

Take care, however! An anonymisation process that sets all names to "John Doe" will not help test a feature that analyses customer names for known patterns. Different features may require their own custom test data, and it's often preferable to manufacture these rather than use production data.

In the modern world of big data and machine learning, even seemingly anonymised production data may be analysed to extract sensitive information. You should always take care when removing production data from its secure environment. If in doubt, it's probably safer to generate synthetic datasets instead.

customers as part of a test suite. For these reasons, production data should remain in the production.

Instead of using production data, lower environments can use 'manufactured' test data. It's not real data, and can be used safely for development and testing purposes.

So, how do we generate data for our other environments? First, work out what data you need on each environment. Frequently, development environments need very little data and a small, fabricated dataset is acceptable. This can be created manually (or automated), and automatically injected into a data store whenever the environment is first set up.

Some use-cases, such as performance testing, require more production-like data. Such data need to have the same "*size*" and "*shape*" as the production dataset. This is usually done through an anonymisation process. Starting with a backup of production data, you can systematically obfuscate any fields that could contain sensitive information. This can be laborious, and needs to be done very carefully. Using pure synthetic data is an alternative, but it can be challenging to reliably replicate the characteristics of a real production dataset.

Promoting Builds vs Rebuilding

We've introduced continuous delivery, its benefits and how it can be used alongside a multi-environment setup to deliver software quickly and safely. Now let's focus on how CI/CD pipelines and multiple environments interact with your source code.

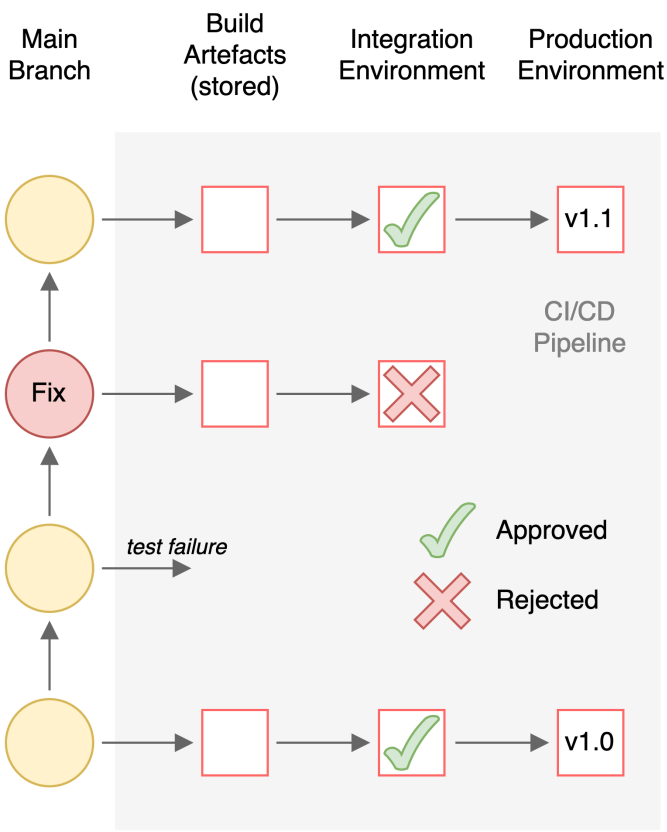
There are a few ways to integrate CI/CD pipelines and multiple environments into a source-controlled workflow. To illustrate the topic, let's consider a system with two persistent environments - integration and production. The system has a CI/CD pipeline that automatically builds and deploys source code to the integration environment, which is used for end-to-end testing and manual review. Human approval is then required to deploy code to production. There are two approaches to this kind of set-up:

1. Build Promotion

Build promotion works with trunk-based development. The idea is that each commit is built only once. That build artefact is stored for later use and can be deployed to as many environments as needed. Here's how it works in practice:

1. Your version control system has a single long-lived branch (`main`).
2. Every time new code is committed to `main`, the CI/CD pipeline builds an artefact and stores it.

Build Promotion



3. Every new build artefact is automatically deployed to the integration environment, replacing the previous build.
4. A person can look at integration, and sign off a production release. If they do this, the exact same build artefact is deployed into the production environment.

There is a lot to like about build promotion. Deployments and source code are neatly decoupled by the CI/CD pipeline, which lets developers implement workflows that suit their needs, rather than structuring their source code to reflect infrastructure. DevOps engineers are free to select any release management solution that fits their needs. The architecture can scale to handle multiple environments, and potentially expensive build steps are never re-run.

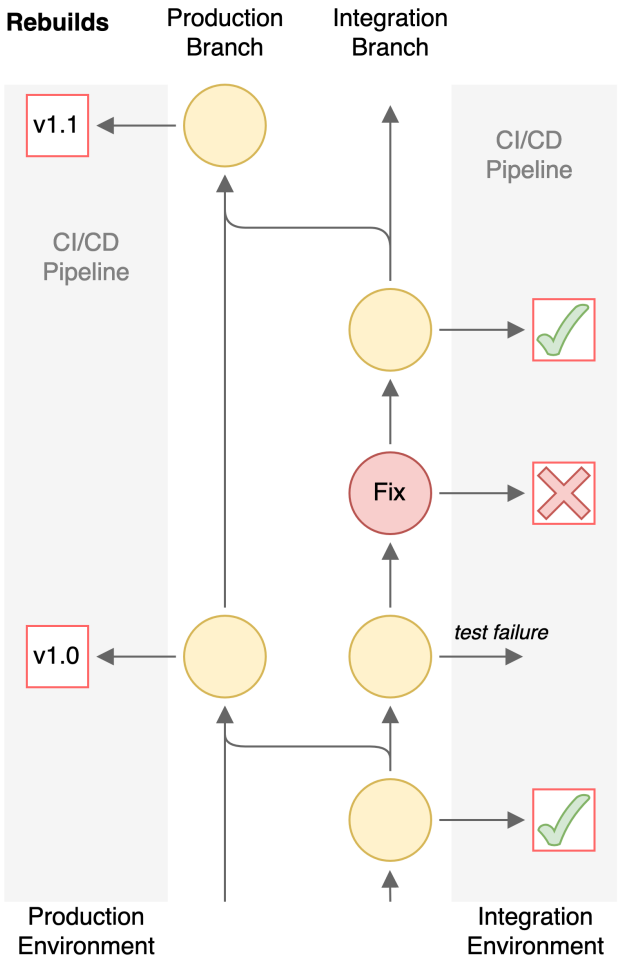
Build promotion works best when the deployment process is quick, and the development team are capable of putting trunk-based development into practice (as discussed in Module 2). In short, it's the preferred option for efficient agile teams that fully embrace automated deployment tooling.

2. Re-building

Sometimes, teams use an alternative pattern known as **rebuilding**. In this case, build artefacts are generated as and when a specific source code version is deployed to an environment. Build artefacts are usually not stored for long, and a software release is defined by the code version (e.g. the git commit hash), rather than a specific build artefact.

Re-building usually leverages multiple long-lived code branches, one to represent each environment. CI/CD pipelines build and deploy code from each branch to the respective environment.

Re-building re-purposes source code management tools (e.g. pull requests, branches) to handle release management. This makes it simpler to set up than build promotion (which requires additional tooling),



but adds complexity to the version control system. It forces developers to work with a branching pattern set by the infrastructure, not their own workflow, and does not scale well when adding more environments.

Teams need to take care when re-building artefacts and using source code branches to manage releases. It can be easy to make mistakes:

- Developers must commit their code to the appropriate branch, otherwise it will be released to the wrong environment.
- DevOps engineers must ensure that building the same code again and again will produce identical build artefacts. Depending on your tech stack, this isn't always guaranteed.

Chapter 3

Dealing With Failure

In every release process, something will go wrong. Mistakes will happen. This doesn't reflect a fundamentally-broken process; as discussed earlier in the course, all systems have flaws and we work to remove and mitigate those flaws. Nevertheless, it's important that we build systems that can tolerate mistakes and recover from them. In particular, we need systems that can recover from failure quickly.

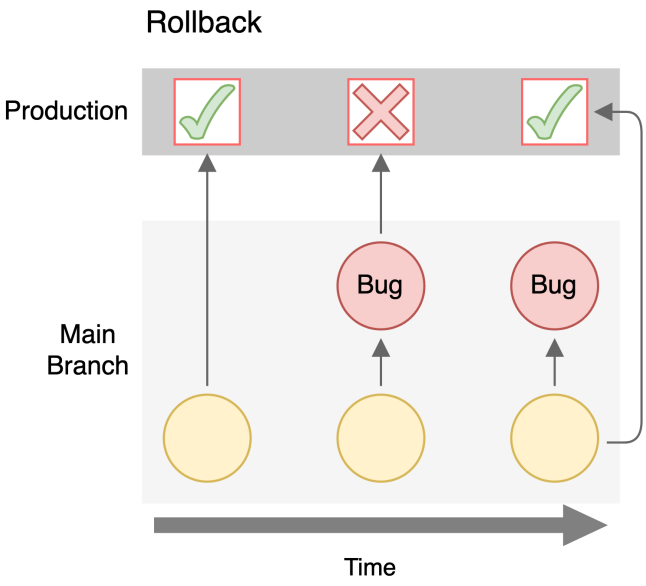
When something does go wrong with a release, it's important to know what to do. In general there are two options: revert to the previous code version (**rollback**) or move ahead and fix the underlying issue with a new release (**roll-forward**).

DevOps engineers should build deployment pipelines that facilitate both approaches, as you never know when they will be needed. The benefits of continual delivery apply to recovery as well as releasing new software. Teams that can deploy features to production within thirty seconds can also deploy a fix to production in thirty seconds! On the other hand, slow manual release processes can cause unacceptable delays to system recovery; teams are left with an uncomfortable choice between bypassing the process or waiting it out.

This discussion is about what happens when you release code into the production environment, and find that it does not work as intended. It does not discuss how to handle what to do if your deployment pipeline fails entirely! That's a separate challenge, more related to debugging than release management.

Rollbacks

What happens if you release your code and then find a bug? Even with the best test suite and most thorough QA process some bugs will only end up manifesting in production. If a bug is *very* minor you might choose to ignore it for the time being, raise a bug card and fix it in a future release. For a lot of bugs, that simply isn't an option and you will want to revert the change. One strategy for this is called a **rollback**.



The ideal rollback scenario is to detect a bug and immediately revert to a previous version of the code. With some deployment strategies this is straightforward: incremental roll outs can be suspended and reversed; blue-green deployments swapped back. If you have mutable servers you may have to redeploy the previous, working version of the live environment.

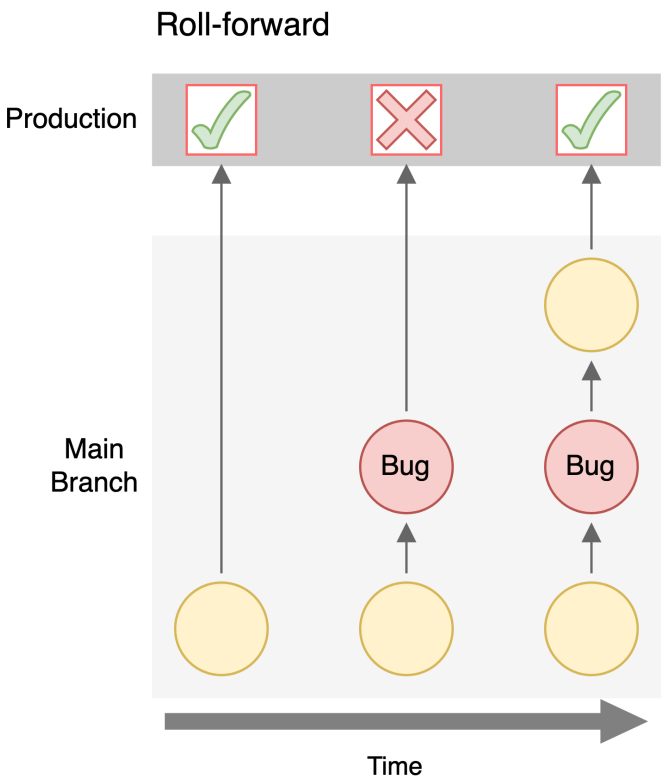
It is generally good practice to be able to rollback any release you make if required; you will find that some organisations require a rollback plan as part of every release.

Rollback is a convenient tool, and works well for its simplicity: "I don't know went wrong, but I know the previous version worked, so I will revert to the previous version". This simplicity lends itself to automation, and we'll discuss automated rollbacks in the next section.

Rollbacks are not without their drawbacks, however. Rollbacks can result in poor UX, as users part-way through a new journey reach an error page when the journey no longer exists. Rollbacks are also tricky if the release involves stateful changes (e.g. creating new data or data storage options). You cannot simply delete all new data as part of a rollback, nor can your previous release handle data only supported by the latest code. Finally, rollbacks neither diagnose nor fix the underlying issue - these issues are left as technical debt.

Roll-Forward

Roll-forward (also known as **fix-forward**) is an alternative to rollback. Instead of reverting to an old release when a production bug is detected, the development team diagnose the issue, expand their test code to cover it and release a *new* version that fixes the issue identified. The new release is deployed through the typical deployment pipeline.



Rolling forward neatly sidesteps the common issues found with rollbacks. No features are lost during a roll forward, and there's no need to deal with tricky state rollback issues.

Rolling forward is not strictly better, however! To deliver a new release, teams must both *debug* and *fix* the issue, steps which could take minutes for simple bugs, or many days for more complex issues. You'll also need new tests to protect against regressions. Finally, a new release needs to go through your organisations standard release process, which may be slow and manual in some organisations. None of these steps are necessary for a rollback.

For these reasons, roll-forward is typically slower than rollback (lowering your MTTR), but is a more flexible approach and delivers a better user experience. Users will not be jarred out of part-complete journeys, nor do you risk data loss or incompatibility. Rolling forward also addresses the underlying issue head-on rather than creating technical debt.

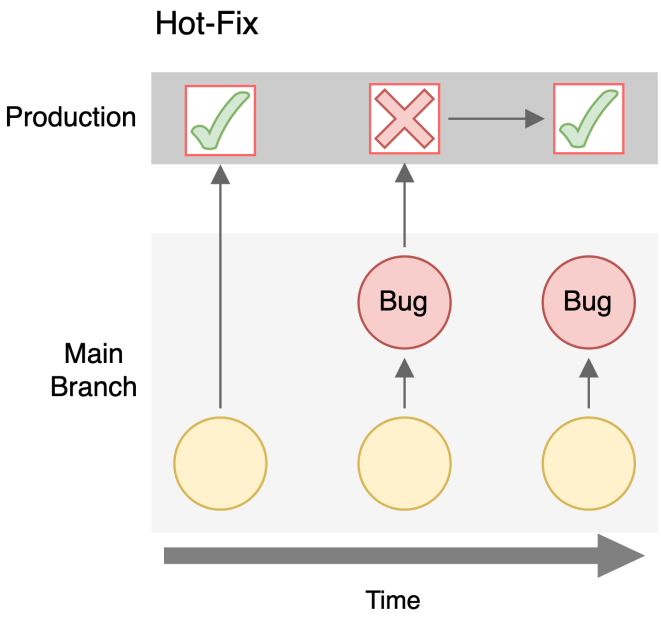
If a production bug needs fixing (but is not catastrophic), and the development team have a strong CI/CD pipeline that can get the code into production quickly, rolling forward is usually a good option.

Hot Fixes

Hot fixes are fixes applied directly to a live system. They bypass the usual validation and deployment processes. Hot fixes are generally used as a tool of last resort: if you can't rollback, and rolling forward would be too slow, make a hot fix.

Hotfixes are common in environments with extremely slow, manual deployment processes. By cutting through the normal approval process, hotfixes are inherently risky. They also create technical debt: fixes must be retrospectively applied to the development code branches *after* they have reached production, and may need significant re-working to be properly tested and adhere to code quality standards.

Technical debt is a real issue, and can significantly reduce a development team's velocity if allowed to accumulate. You should really only apply hot fixes as a last resort. If you find yourself making hotfixes frequently, take a look at your tooling and pipelines. Are there ways you can make the system work better?



Chapter 4

From Delivery To Deployment

So far, this module has discussed continuous delivery pipelines. Such pipelines automatically test, build and deliver deployable software. Often they will automatically deploy software into development or testing environments, but they don't automatically deploy new code into production. When practicing continual delivery, production deployments require human approval.

Continuous deployment takes continuous delivery a step further by removing human oversight from the production deployment process. It relies entirely on automated systems to test, deploy and monitor software deployments. This is not a complex transition from a technical point of view, but has wide-reaching implications for businesses that choose to put it into practice.

Continuous deployment is often considered the gold standard of DevOps practices. It gives the development team full control over software releases and brings to the table a range of development and operations benefits:

- **Faster, more frequent deployments**. Removing the human bottleneck from software deployments enables teams to move from one or two deployments a day to hundreds or thousands. Code changes are deployed to production within minutes.
- **Development velocity**. Development can build, deploy and test new features extremely quickly.
- **Faster recovery**. If a release fails, you can leverage a continuous delivery pipeline to get a fix into production quickly, without skipping any routine validation steps. Hot fixes are no longer needed.
- **Lower overheads**. Checking and approving every single release is a drain on team resources, particularly in Agile teams where frequent deployments are the norm. Continuous delivery frees up team resources for other activities.
- **Automated code changes**. Dependency checkers, and similar tools, can leverage a fully automated pipeline to update and deploy code without human intervention. This is commonly used to automatically apply security fixes and to keep dependencies up-to-date, even in the middle of the night!

Continuous deployment has the same technical requirements as continuous delivery: teams need excellent automated testing, system monitoring and a culture of continual improvement. Continuous deployment simply removes the human "safety net" from releases, thereby placing greater emphasis on the automated steps.

Continuous deployment can be a huge change for some organisations, and may sound particularly alien in organisations that typically rely on human testing and sign-off to approve large, infrequent releases. In such cases, continuous deployment is best viewed as the end goal of a long DevOps transition that begins with more manageable steps (e.g. agile workflows and automated testing).

Continuous Delivery vs Continuous Deployment

Despite their similar names, continuous delivery and continuous deployment are very different in practice!

As a DevOps engineer, you are responsible for designing and implementing software deployment pipelines that meet business and developer needs. It's important that you understand the various factors that favour, or prohibit, continuous deployment.

Continuous deployment is the gold standard for most DevOps professionals. However, it is not something you can simply implement; for most businesses, continuous deployment is not primarily a technical challenge to overcome - it requires a fundamental shift in business practice and culture. It's best to think of continuous deployment as the end-product of a steady transition to DevOps culture and values from more traditional Waterfall-based methodologies, rather than a one-off pipeline enhancement.

That transition typically starts with building automated tests and a CI pipeline into an existing manual testing and deployment process. Steadily, automated tests reduce the load on the testing team and your organisation begins to develop trust in the process. You might then build a continuous delivery pipeline to automate some steps of the traditional Ops role. Eventually, once the testing suite is hardened and your organisation is comfortable with leveraging automated pipelines, you can introduce continual deployment, perhaps into a blue-green production environment to provide additional recovery guarantees to concerned stakeholders. This account is fictional, but gives an example of how a DevOps transition to continuous deployment might play out in reality.

CD or CD?

Continuous delivery and continuous deployment are both, unhelpfully, abbreviated to CD! Despite similar names and an identical abbreviation, the two workflows are quite different in practice.

If your team has a CI/CD pipeline, make sure you know which flavour of CD you are talking about!

In some circumstances, continuous deployment is not possible and you should focus instead on delivering great continuous integration & delivery pipelines. This usually happens when human sign-off is required by law. We'll discuss how this sort of regulatory constraint can impact DevOps priorities shortly.

Autonomous Systems

Automated systems are generally "one-click" operations. A human triggers a process (e.g. runs a shell script), and the automated system handles the tedious detail of executing the task. Automated systems know *what* to do, but require external prompts to tell them *when* to do it.

Autonomous systems take automation to the next level. An autonomous system knows *what* to do, and also *when* to do it. Autonomous systems require no human interaction, operate much faster and scale better than human-triggered automated systems. Continuous deployment pipelines are autonomous systems.

Continuous deployment pipelines aren't just used to deploy new features. They are integral to larger autonomous systems that can handle other aspects of the software deployment process.

Automated rollbacks

Organisations using continuous deployment rely on automated monitoring and alerting systems to identify failed releases and take appropriate action. Often, the quickest and simplest solution to rollback to the previous working version.

Rollbacks can be automated using monitoring and a continuous deployment pipeline. If a monitoring system triggers an alert following a release (e.g. an app exceeds an allowable error rate), the system can trigger a rollback. Because the deployment pipeline is autonomous, the whole process can be done without human intervention. That means rollbacks can be performed quickly, reproducibly, and at all hours.

Autonomous monitoring and repair approaches are often only possible in environments using continual deployment. By removing the human bottleneck, rollbacks become quicker and more reliable. This further reduces the risk of deployments. Also, rollbacks are tedious and repetitive to do manually - automating the process is a great help to team morale!

Remember, not every bug can be fixed with a rollback, and all rollbacks create some technical debt as developers still need to fix the underlying issue. Nevertheless, automated rollbacks can make continuous deployments far safer.

Regulatory Constraints

The software you produce, and the ways in which you produce it, may need to meet certain regulatory requirements. Most sectors don't need to worry about this, but certain industries (e.g. banking) are heavily regulated. If you work in a highly-regulated industry, it's important to be aware of the regulatory requirements that exist. You should know how to deliver deployment pipelines that not only meet those requirements, but also help demonstrate compliance to auditors.

This course is no legal primer, but we can talk about the kinds of requirements that exist, and introduce some approaches DevOps-based teams can employ to demonstrate compliance.

DevOps Audit Defense Toolkit

The DevOps Audit Defense Toolkit summarises common techniques that Agile teams can use to demonstrate regulatory compliance. If you work in a heavily regulated industry, take some time to read the document in full. Even if you don't, we recommend having a look - it's a useful narrative that all DevOps engineers can benefit from.

The toolkit highlights some general points relevant to all DevOps deployment pipelines:

- How can I prove that change X was reviewed by person Y?
- Have I taken reasonable precautions to prevent unauthorised releases?
- Can I readily access a complete deployment history?

As a DevOps engineer, you should be aware of these potential issues and limitations. You are likely expected to build, or at least maintain, the path from source code to deployed software where regulatory requirements should be evidenced.

Chapter 5

Additional Material

External Resources

Continuous delivery and deployment are popular topics that have wide-ranging implications for business practices and team workflows. There are plenty of resources out there that can give you a flavour of how these process are used. Here are a few recommendations:

1. The Phoenix Project (Book)
2. Google SRE Books

You can also find many blogs, newsletters and videos on the topic.

CD platforms

Continuous delivery and deployment are generally run using the same tools and platforms that power CI pipelines. In fact, the two are usually implemented together (a CI/CD pipeline). You can find a list of popular options in Module 7.

Specific Regulations

Below are a few common examples of legislation that places regulatory requirements on software and the way it is produced. These just give a flavour! You should be aware of specific legislation relevant to your business environment.

Regulatory requirements will vary between countries, industries and over time.

FINRA

FINRA rule 1220 requires that software development for US investment banking purposes is overseen by a person who has passed the Series 99 exam. You need to be able to prove this to the auditors. In practical terms, this means that releases that include features that support the functions covered by rule 1220 need to be signed off by a S99 Certified Operations Professional.

From a DevOps point of view, this means that your deployment pipeline should enforce this sign-off requirement, and must include the traceability necessary to prove this.

SOX

The Sarbanes-Oxley Act (Section 404) is another piece of US regulation impacting DevOps. This one extends beyond banking, to all publicly traded companies in the US. Companies are required to take steps to ensure that the systems and processes that contribute to their financial reports are accurate and effective. This impacts the design, testing and deployment of software that deals with financial data, including sales, etc.

The law does not say exactly what practices must be followed, but rather it requires that companies have practices that work. Because this is very complicated, most companies follow established frameworks, such as COBIT and COSO, which are recognised by auditors as ensuring compliance with the law. Auditors will often simply assess compliance with one of these frameworks, rather than the law. Agile teams can fall afoul of these frameworks, since they were written based on Waterfall-like processes.

Typical controls include segregation of duties, which requires that processes be designed so that multiple persons are required to make a change that could impact financial figures. In waterfall development this condition is met naturally: the developer writes the code, a test team tests it, then a Ops team deploys and manages the code in production.

When practicing DevOps culture, we don't have 3 separate teams for these functions, so other ways must be found to mitigate the risk of a developer (intentionally or deliberately) changing a production system such that it produces inaccurate figures.