Octopus Deploy

How to map your Deployment Pipeline

Contents

Introduction

The deployment pipeline is the key ingredient for Continuous Delivery. You can increase deployment frequency and reduce change-related risk by mapping and improving your deployment pipeline.

We also have a white paper on The importance of Continuous Delivery[1] that describes the link between Continuous Integration, Continuous Delivery, and DevOps. It includes research-backed statistical links between specific practices and high-performing teams.

DevOps isn't just for tech giants; it also works well for small teams. DevOps is also tech-neutral, suitable for any tech stack you use. Thousands of organizations use DevOps to deliver modern distributed systems. Others apply DevOps to their mainframe and firmware development efforts.

A key challenge for organizations adopting DevOps is increasing the safety and rate of deployments. DevOps needs a generative culture alongside the technical capabilities of Continuous Delivery.

The table below shows that high-performers:

- Deploy more often
- Have shorter change lead times
- Experience fewer change-related failures
- Resolve faults faster

	Deployment frequency	Change lead time	Mean time to resolve	Change failure rate
Low	Monthly or less often	One week to 6 months	< 1 week	< 15%
Mid	1-5 days	< 1 week	< 1 day	< 15%
High	On demand / any time	< 1 hour	< 1 hour	< 5%

Source: The State of DevOps Report. Puppet 2021[2]

1 https://octopus.com/resource-center
2 https://puppet.com/resources/report/2021-state-of-devops-report

The challenge of moving to DevOps can be daunting, but in this white paper you'll learn:

- How to apply Lean thinking to *start from where you are*
- About mapping and improving your deployment pipeline
- Planning your adoption of the key technical practices

Where to start

Unless working on something completely new, you likely need to bring Continuous Delivery to an existing system. Don't let the task ahead put you off! People have used all Continuous Delivery practices, even on large legacy systems.

You need to assess where you're starting from, but don't get paralyzed by the amount of work. Each small step will provide time and space for more improvements. You should find you'll move faster as you introduce tools and practices that free your time from other activities.

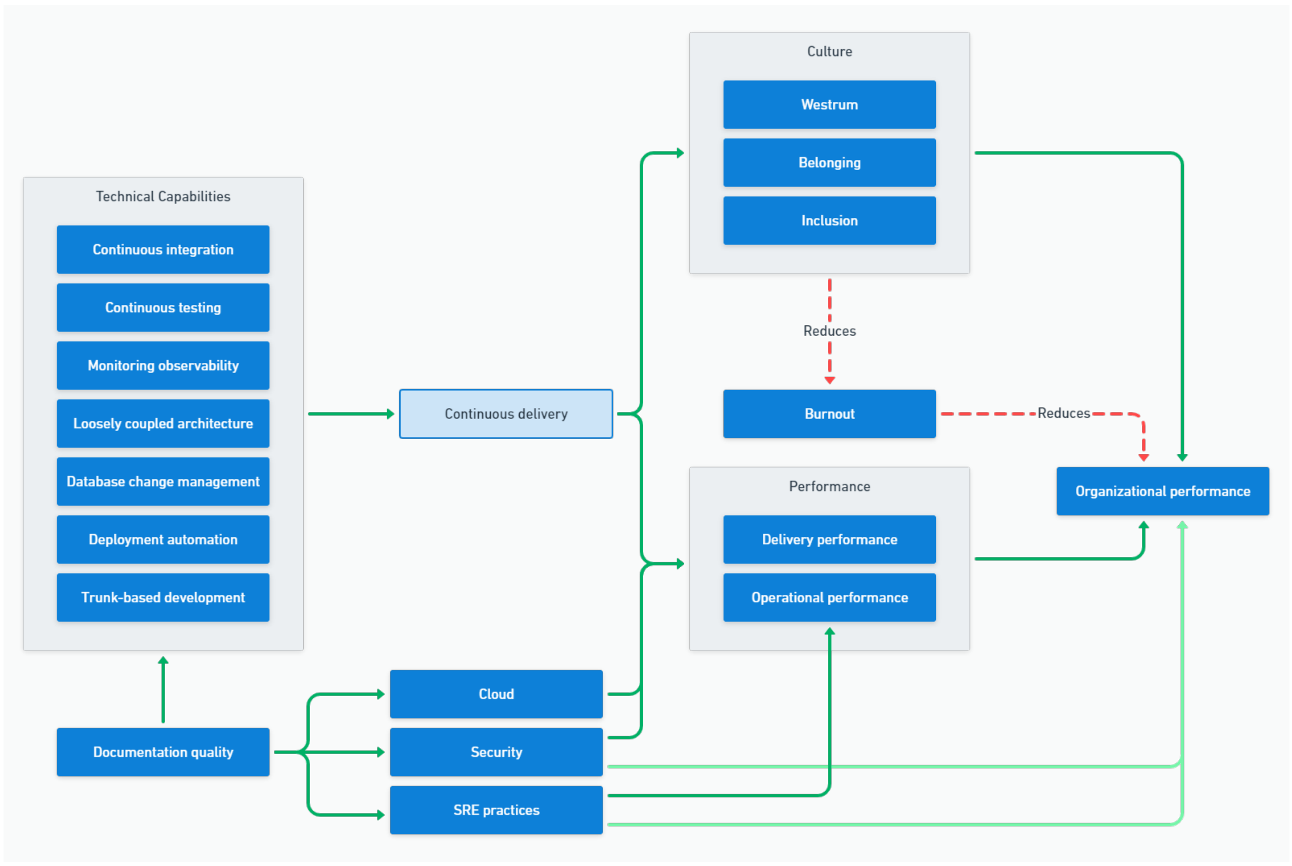
You may have challenges ahead, but you can do this!

To start, identify a small and independently deployable part of your system.

Component selection

From the statistical model of DevOps, Continuous Delivery has 7 specific technical capabilities:

- Continuous integration
- Continuous testing
- Monitoring observability
- Loosely coupled architecture
- Database change management
- Deployment automation
- Trunk-based development



The DevOps Structural Equation Model

Of these, loosely coupled architecture and continuous testing are the most difficult to add in retrospect. You may face challenges adopting Continuous Delivery for applications or components with poor isolation from the rest of the system, or those without automated tests.

If these two capabilities are absent from your system, you should try to create a *walking skeleton*, a minimal version of the system that meets a single need. You can then build a deployment pipeline to support this walking skeleton and learn how to apply the techniques more broadly.

You may have a system that already has a loosely coupled architecture and automated tests. Where you feel comfortable, you could skip the walking skeleton and create the deployment pipeline around that. When starting with an existing application, select something small and separately deployable first.

No matter the route, your goal is to gain experience applying Continuous Delivery practices and shaping how you'll work in the future. You'll also be able to prove the benefits of Continuous Delivery using your first deployment pipeline.

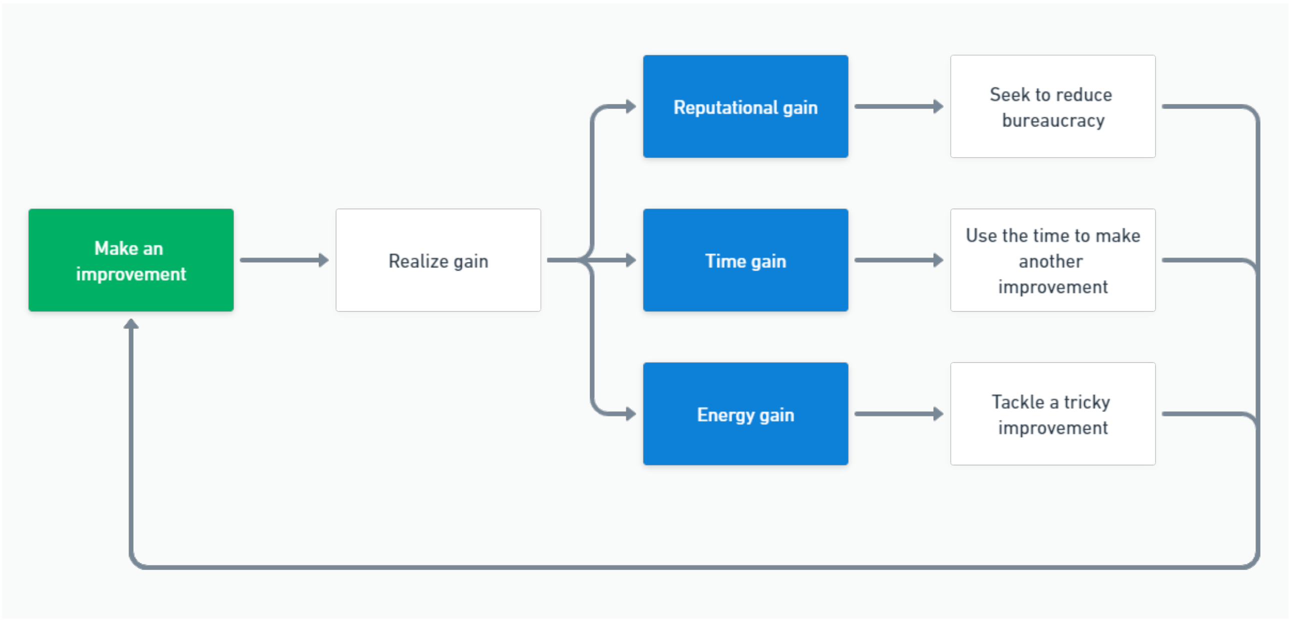
Reinvest your gains

Each time you improve your deployment pipeline, you gain something. The gain may be time, reputation, or motivation. You can use each of these gains differently to make more improvements. Your gains should accumulate over time, even if some of your attempts don't move the win line.

When you save time (by automating a process that took an hour a week, for example), you can use that time to make more improvements. This is the most direct reinvestment opportunity. Keep a running total of time saved and put it all back into other improvements.

If you introduce an improvement that has your team fired up, it can be a good time to tackle a more tricky improvement. You might have a difficult step to automate, and the extra motivation will help the team make it happen.

When you make a reputational gain, take the chance to remove adjacent bureaucratic steps in your deployment pipeline. Most bureaucratic steps in your deployment pipeline exist due to past problems. If you solve the root cause of past deployment pains, you can ask to remove the redundant double-check steps.


Make an improvement
Realize gain
Reputational gain
Time gain
Energy gain
Seek to reduce bureaucracy
Use the time to make another improvement
Tackle a tricky improvement

Use different types of gain to reinvest in Continuous Delivery

When you first start to bank gains, you should aim to reinvest 100% back into deployment pipeline improvements. As you get further into your Continuous Delivery journey, you can redirect a bigger share of gains into other work, like feature development. Never let your improvement budget reach zero, though!

Iterate

Improving and reinvesting gains only works if you map and measure your end-to-end process. That means the time to get a code change from a developer's machine into production.

The first version isn't a Continuous Delivery pipeline, just a plain old deployment pipeline. By making a series of small changes, you can eventually meet the goal of *high-quality, regular software releases to production*. The number and difficulty of these steps will vary, but keep in mind Arthur Ashe's famous quote:

“Start where you are. Use what you have. Do what you can.”

Start with improvements that offer the largest gains for the smallest investment, and need the fewest people to agree.

Summary

Start from where you are and make an effort to understand the existing deployment pipeline, even if you're going to create a new component. The existing deployment process indicates people's expectations for your deployment pipeline. This doesn't mean you have to include the same steps, but you might have to convince people to accept new ways of working.

Start measuring your key metrics as early as possible. You need these to measure your own improvement efforts and convince others of Continuous Delivery's benefits.

Octopus has a white paper dedicated to measuring Continuous Delivery, which can help you decide what to measure.

Introducing a deployment pipeline

Before building out your deployment pipeline, assess your context and accept practical short-term limitations. Choose one small improvement that will give you a large return. This lets you increase your ability to do bigger things later.

Keep in mind the aims of the deployment pipeline as you plan and build it:

- **Feedback** - Your pipeline should amplify feedback and solve problems as early as possible
- **Visibility** - The whole process should be visible to everyone involved
- **Self-service** - Teams should be able to deploy any version to any environment at any time

As well as building a deployment pipeline, you're also growing your team's reputation. You can prove many benefits at low risk by implementing stages earlier in the deployment lifecycle. For example, you could start with frequent, high-quality releases to pre-live environments.

Although well-established capabilities exist, you can be pragmatic in approaching their introduction. You should introduce change at a pace the organization can manage. If you try to change things too fast, you might meet resistance from people who don't understand your plans' impact on their team or role.

We cover the ideal cultural factors, pipeline stages, and tool support next.

Cultural factors

If your organization has a strong culture, you and others will understand its values and how they expect you to work. You should be able to classify a strong culture using the Westrum model.

In a weak culture, where values and working methods are unclear, you tend to find divisions have subcultures with conflicting values and ways of working. Weak cultures are harder to classify, but you should be able to determine the prevailing local culture your team experiences.

The ideal cultural state for Continuous Delivery is high-trust and information-rich. Westrum calls this a *generative* culture.

Westrum believes a *generative* culture has the following values:

- Psychological safety
- Cooperation
- Shared risk
- Innovation

We explain the three categories of the Westrum typology below.

Pathological

Pathological organizations form around power structures. Groups work alone and define involvement in work to minimize responsibility when something goes wrong.

Pathological organizations:

- Discourage collaboration between groups
- Stifle new ideas
- Quickly assign blame to a scapegoat

Bureaucratic

A bureaucratic organization is rule-oriented, with separate groups having limited collaboration and innovation. The root cause of failures is usually determined to be an accountable person.

Generative

A generative organization focuses on performance. Generative organizations encourage collaboration and share risks. They see failure as systemic rather than individual and introduce improvements after investigation.

The ideal culture

Ideally, you should adopt and grow a generative culture. Organizations achieve high levels of performance when:

- Responsibilities and goals are clear
- There are common tools, languages, and methods
- Teams share best practices

Some observable properties of the ideal culture are:

- **Psychological safety** - People feel safe sharing bad news so you can make things better
- **Cross-functional teams** - Representation for every area involved in software delivery
- **Shared responsibility** - Things like security, quality, and reliability are everyone's job
- **Innovation** - People are free to explore new ideas

In the absence of any of these properties, you can predict the kind of challenges you may face when introducing Continuous Delivery.

Common tools

Continuous Delivery and DevOps are not just about tools, but automation *is* a key element in reducing repetitive work, or *toil*. There are many tools with a valuable and well-established place in the deployment pipeline. You may already have some of these in place:

- Version Control System
- Build Management System
- Artifact Repository
- Deployment Automation
- Environment Management
- Monitoring and Alerting System

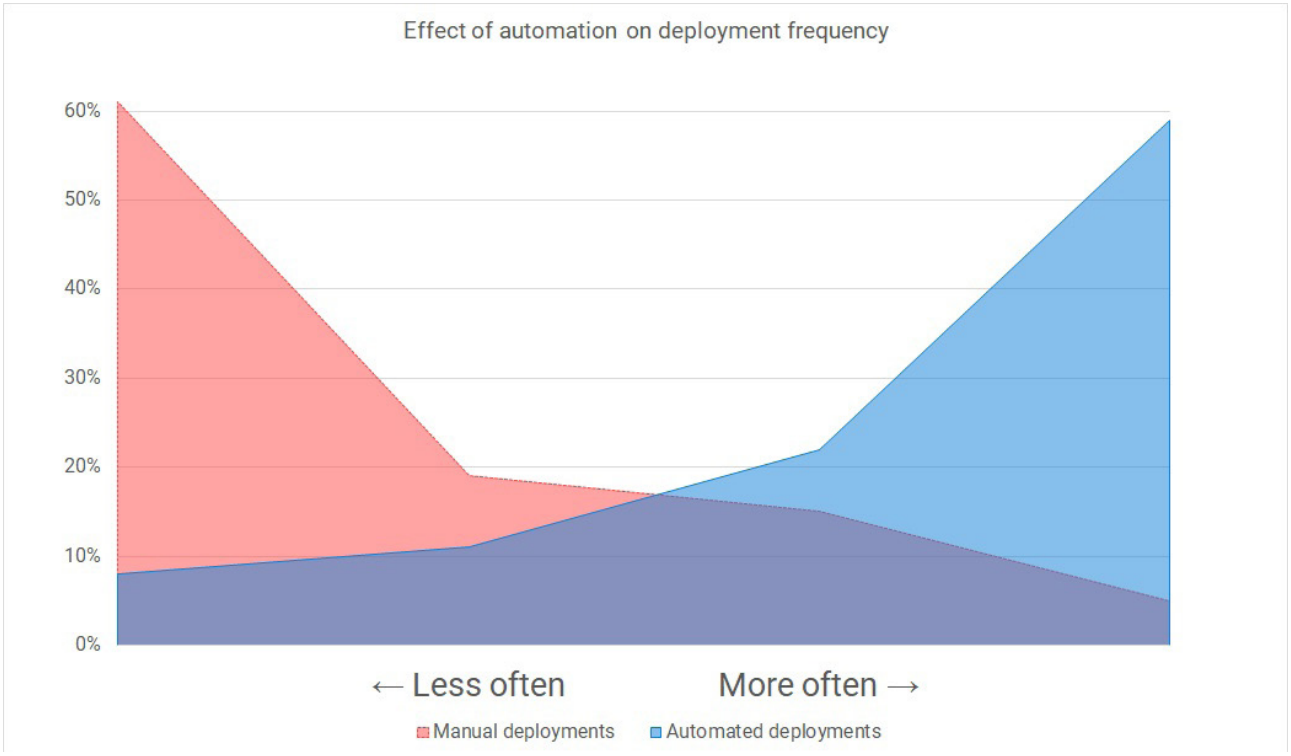
If you don't yet have these tools, you can plan to adopt them alongside the basic deployment pipeline.

Almost all software development teams have a version control system and build-management system. Fewer have all the tools within a complete deployment pipeline.

Manual work is expensive and varies too much. Use Continuous Delivery tools to automate all work that:

- Needs high accuracy
- Happens often
- Doesn't need human interaction

Without this automation, it becomes impossible to increase the rate of deployments.



Automation increases deployment frequency

Our research shows:

- Without automation, teams are most likely to deploy once a month
- With automation, most teams deploy software at least once a day

Deployments	**Manual deployments**	**Automated deployments**
One or more times a day	5%	**59%**
Several times a week	15%	**22%**
Once a week	**19%**	11%
Once a month	**61%**	8%

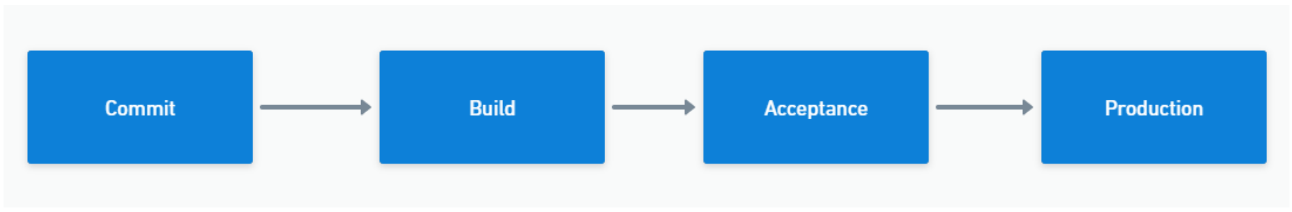
Source: The benefits of deployment automation[3]

3 https://download.octopusdeploy.com/files/whitepaper-automated-deployment-octopus-deploy.pdf

Basic deployment pipeline

The minimal deployment pipeline has four stages:

- Commit
- Build
- Acceptance
- Production



A simple four-stage deployment pipeline

You should introduce stages in this order if they don't already exist. A phase shouldn't run if the previous stage fails.

In addition to the deployment pipeline, you should be continuously monitoring your application. Having a monitoring and alerting tool in place will give you the confidence to deploy often.

We describe each stage below, including the goal, tools, timescale, and method.

Commit

Goal:

- Reduce the size and complexity of integrating code
- Avoid outdated branches
- Keep the cost of merging changes as low as possible

Tools:

- Version control

Timescale: 10 minutes

Method

The commit stage uses the Continuous Integration technique. Every few hours (at least once a day), each developer integrates their changes into the main branch in version control.

Doing this often usually results in plain sailing. But, if there is a problem and the developer can't fix the issue in 10 minutes, they should revert their changes.

Before you introduce this technique, you need to merge your existing branches to get an up-to-date main branch. Developers should then work against this main branch using Continuous Integration.

Although the process is simple to describe, it can be a big change for developers. Plus, it'll be even harder if you have old habits that shame developers for breaking a build. You should encourage developers to commit often, so it must be safe for everyone to make a few mistakes along the way.

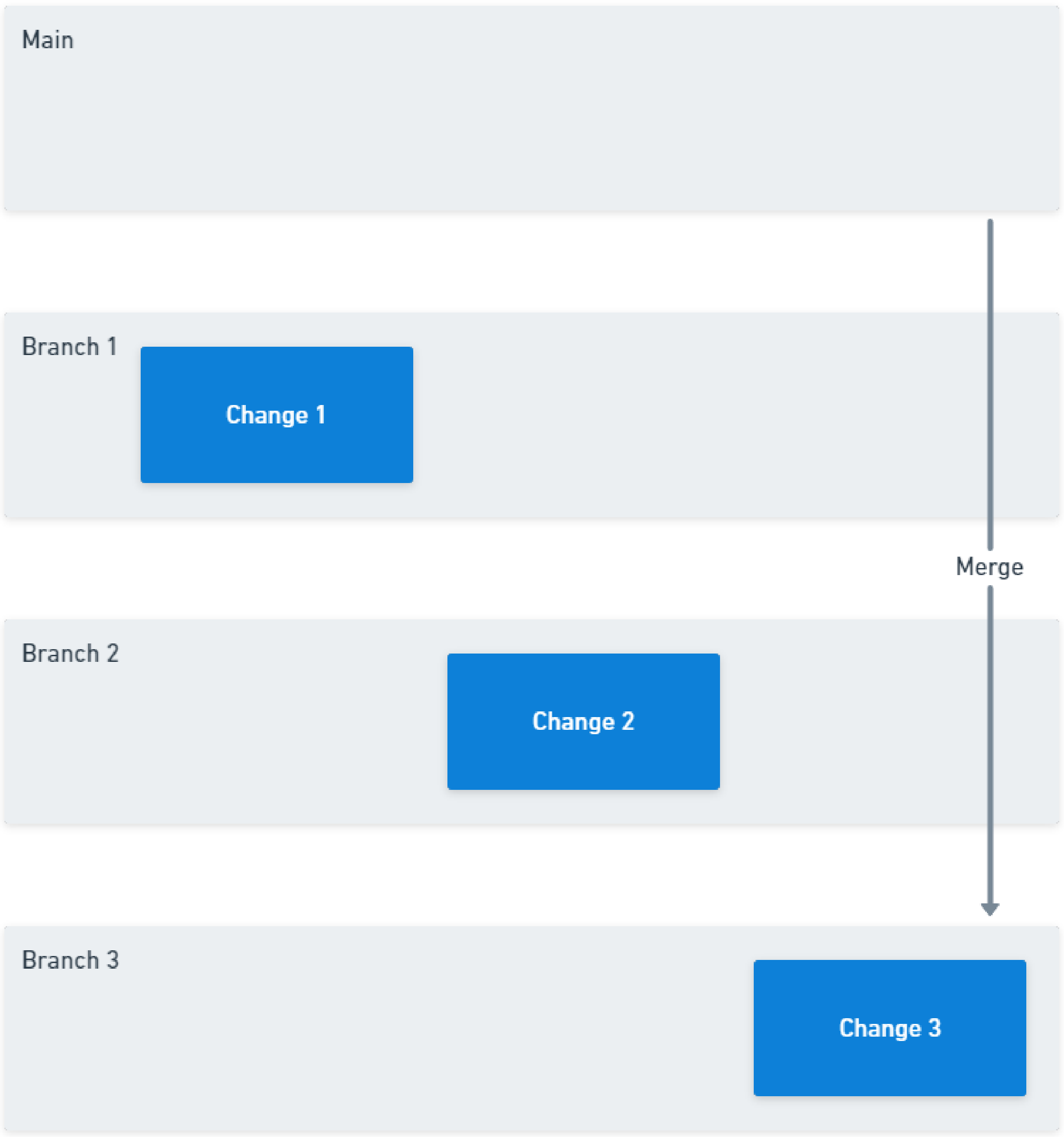
The process for Continuous Integration is:

1. Pull changes from the main branch to the local copy
2. Rebuild the local copy with the merged changes
3. Run tests locally
4. Commit to the main branch if the tests pass

Running tests locally on the integrated application version prevents deployment pipelines from getting blocked by minor faults developers can fix quickly. The more you perform this step, the smaller the integration task.

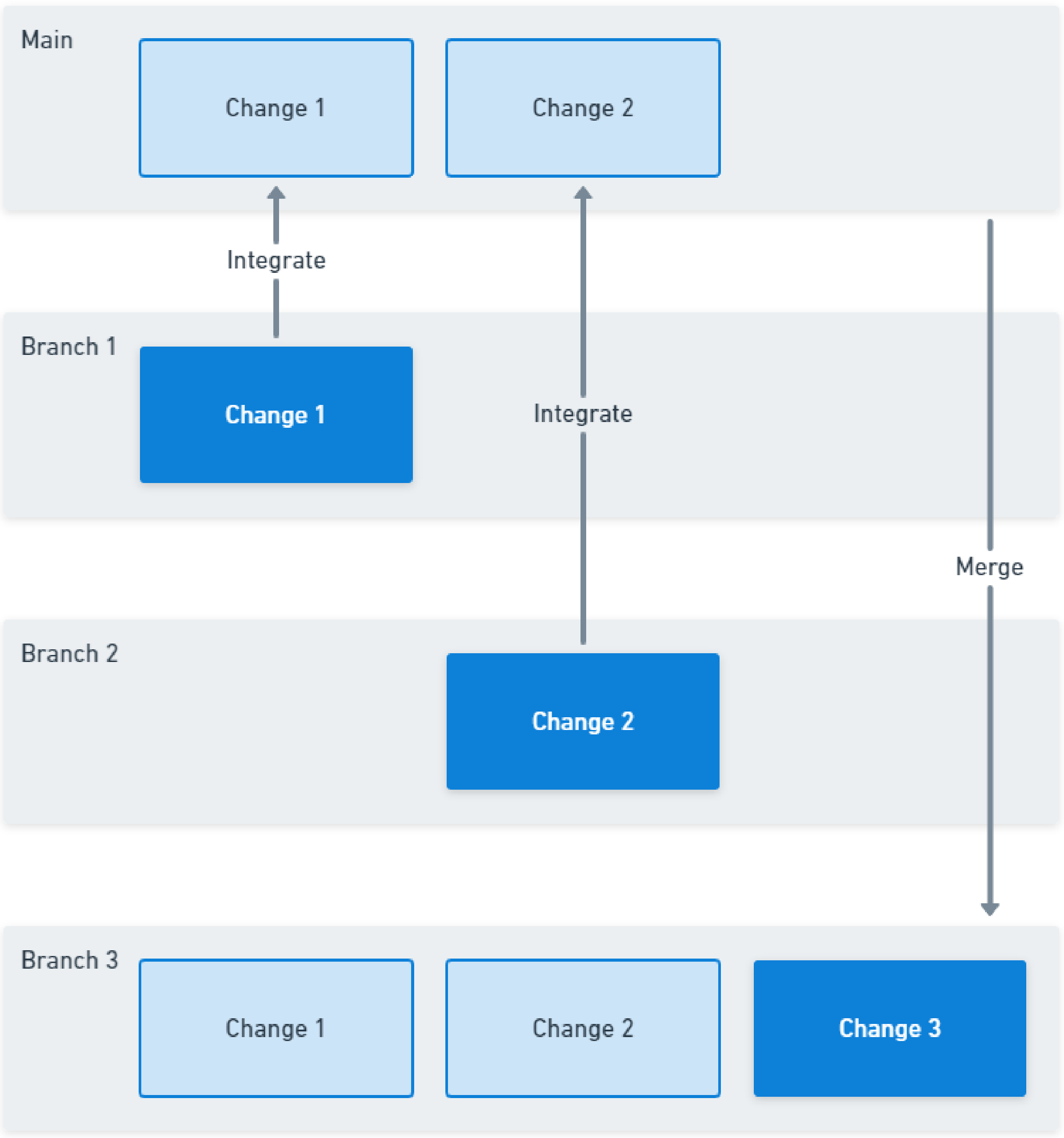
You only perform Continuous Integration when the whole team merges their code into the main branch daily.

Merging changes from the mainline into branches often doesn't solve the problem, as most changes aren't in the main line. You can only minimize integration risk by keeping the main line updated.



Long lived branches are naturally out of date

High-performing teams have developers who commit their changes to the main branch several times a day.



Trunk-based development means less change drift

Build

Goal:

- Provide almost immediate feedback to developers about issues with the software
- Create a canonical package to re-use for deployments to all environments

Tools:

- Version control
- Build management
- Artifact repository

Timescale: 5 minutes (automated)

Method

With each commit to the main branch, the build management system should automatically build and test the new version of the software. Once complete, the build management system should send a package with the new version to the artifact repository.

It's important all deployments use the same package for this version of the code. You shouldn't need to rebuild the package for deployments or rollbacks.

Most build management systems allow the build, test, and package upload as steps within a workflow. The packing approach will be specific to your application. It could be as simple as zipping some files, or may involve creating an installer or docker image. Most build management systems have step templates that simplify this process for common package types.

The artifact repository may be part of your deployment automation system, like the package repository in Octopus Deploy. It might also be a separate package repository or registry.

Developers should only get involved in this step if something fails. Everything should be automatic, including alerting developers to a fault.

Acceptance

Goal:

- Validate the software in a live-like environment
- Run acceptance tests to catch faults

Tools:

- Build management system
- Artifact repository
- Deployment automation
- Environment management

Timescale: Within 1 hour of the original commit

Method

The acceptance stage should happen automatically if the software version passes build stage requirements. The build management system usually triggers the following steps:

1. Create the environment (environment management)
2. Deploy the package (artifact repository and deployment automation)
3. Run the acceptance tests (build management system or deployment automation)

Acceptance tests should include functional business tests and quality attribute tests, like security and performance. You can introduce separate stages to handle complex scenarios. For example, you could add component and system performance testing in different stages.

Manual exploratory or user experience testing should only happen *after* the acceptance stage. This way, people only spend time testing high-quality software versions.

Production

Goal: Get feedback early by safely, reliably, and frequently deploying new versions to production.

Tools:

- Artifact repository
- Deployment automation
- Environment management

Timescale: Under 30 minutes

Method

You should use the same automated deployment process for all environments. This means you test the deployment automation as often as you test the application code.

Ideally, you should automate both environment and deployment, though many teams first deploy to an existing production environment. You can recover from faults easier and faster if you automate the environment, just like deployments.

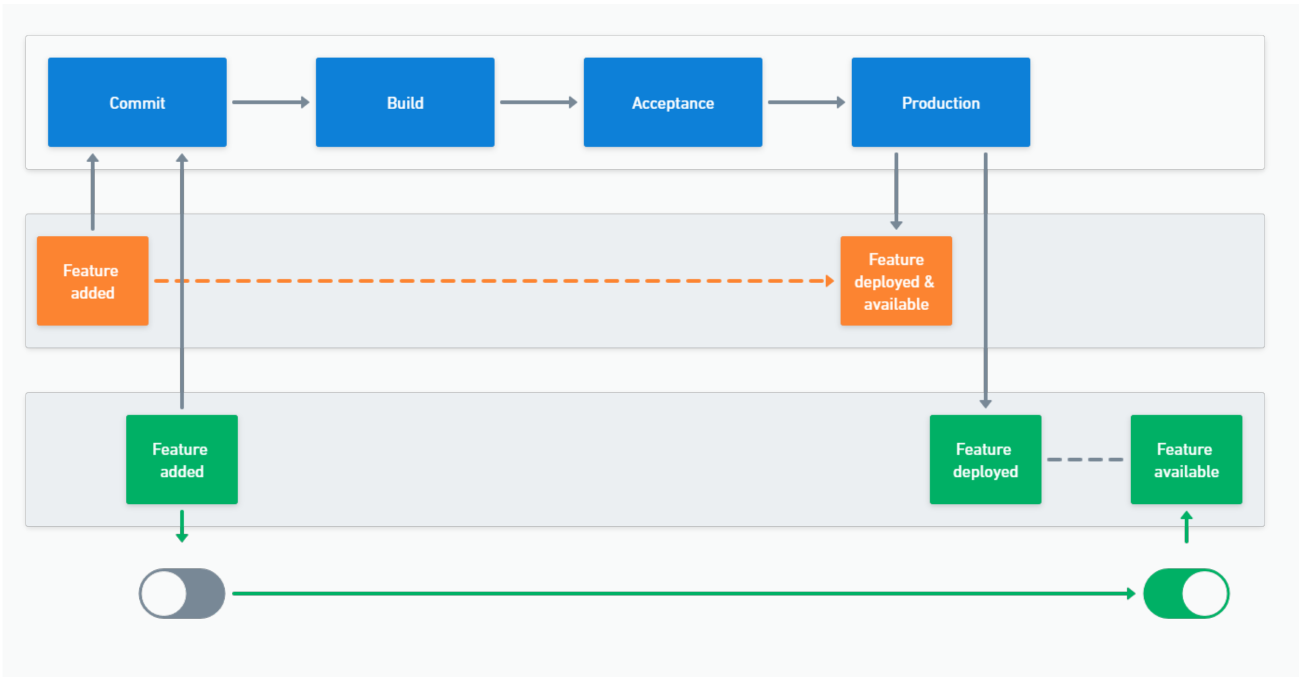
Your deployment automation tools should provide everything needed to track production changes and capture approvals in the audit log. The history in version control and your deployment automation tools solves many of your compliance and regulatory requirements.

Continuous Delivery requires that software is always ready to deploy. You might deploy every change as soon as it's ready, or regularly deploy the latest available version.

One situation to avoid is delaying deployments as part of your marketing strategy. It should be possible to control the release of features independent of deployments.

If your organization wants to schedule feature *releases*, consider separating deployments and releases. With techniques like feature toggles, you can deploy all versions you build but still choose when to enable a feature. This also allows you to make a feature live for a group of users before making it widely available.

Feature toggles also allow you to temporarily suspend a feature if there is a fault, or to reduce load during an unexpected peak.



Use feature toggles, not deployments, to control feature availability

Monitoring and alerting

Goal: Know about problems before anyone else and fix them fast

Tools: Monitoring and alerting system

Timescale: Continual and real-time

Method

Monitoring and alerting happens continually outside of your deployment pipeline. Having automated tools to detect production issues early is vital in giving confidence to make changes more often. Many organizations start by monitoring infrastructure resource metrics, but you should aim to set up business metrics and use those to detect problems.

Improving your deployment pipeline

Once you have a working deployment pipeline, there are techniques to improve the quality and frequency of deployments.

Find and elevate constraints

The goal of Continuous Delivery is frequent, high-quality releases. A *constraint* is anything that prevents you achieving this goal.

Eli Goldratt described how there can only be one constraint, which sets the pace for the whole system. For example, even if pull requests take 4 hours to approve, they are not the constraint if a management deployment approval step takes 2 days.

You can classify constraints as:

- Tools - For example, not using automation for routine and repetitive tasks, or not buying a product to support a stage in the deployment pipeline
- People - Such as a lack of skilled people, roles sat outside teams, or mental models that prevent improvements
- Policy - For example, committee-based approvals or decision making authority outside the team

Identify the constraint and use it to set the pace for the whole deployment pipeline. This means reducing throughput at all previous stages to match the pace of the constrained stage.

Then, you can focus improvement efforts on the constraint by:

- Automating all or part of the stage
- Aligning people to the work
- Changing policy to improve flow

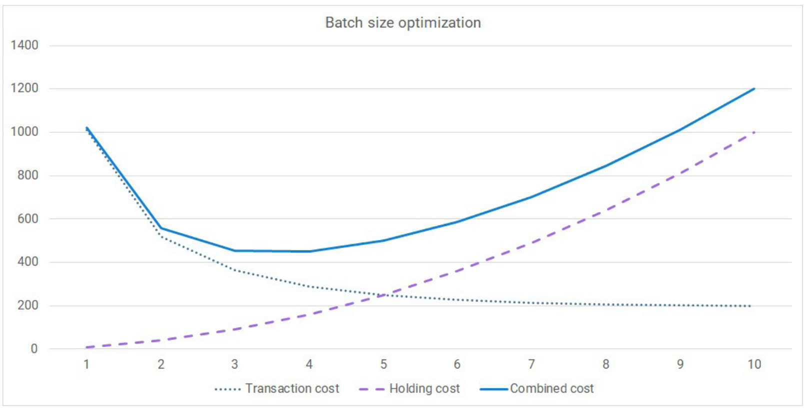
Avoid hiding constraints in batches

It's important to avoid using batches as a buffer to hide a constraint. Small batches drive Continuous performance[4]. Your deployment pipeline needs to allow any single change to flow through to production.

Collecting a series of changes into batches for approval means all changes are at risk if just one doesn't get approved. You must keep your software deployable at all times, so you need to reduce the chance and duration of any blocking changes.

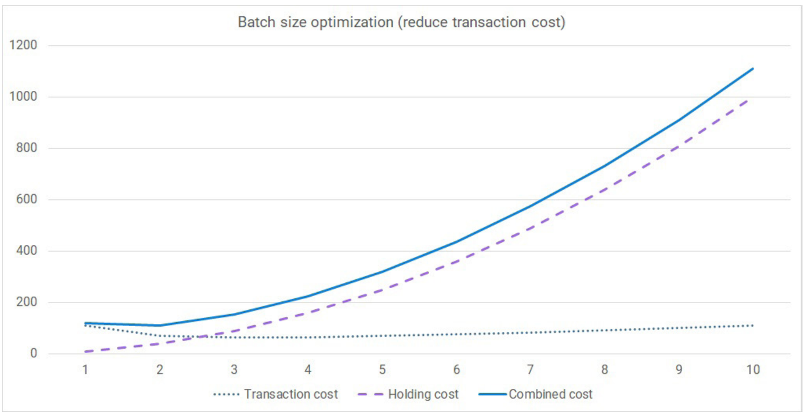
In the past, teams would optimize batch sizes by calculating the combined transaction and holding cost:

- Transaction cost - Refers to the overhead of processing a batch, like time spent building, testing, and deploying software
- Holding cost - Refers to the money invested in the batch and the risk of needing more investment to complete, like bug fixing and re-testing



The transaction cost usually encourages organizations to work in large batches. If a test cycle involves 5 testers for 2 weeks, it's expensive to perform the tests for each small change.

The chart below shows a typical transaction vs. holding trade-off. Using the combined cost, a batch size of 3 or 4 is optimal.



Rather than accepting the trade-off, Continuous Delivery lowers the transaction cost to where processing one change is the optimal batch size.

If you allow batches to get larger, the investment increases and the risk gets higher and more complex. There is no method to prevent holding costs increasing, so you should focus on reducing the transaction cost.

4 https://cloud.google.com/architecture/devops/devops-process-working-in-small-batches

Common deployment pipeline constraints

Here are some common constraints to help you consider where to look for improvements in your deployment pipeline.

Quality constraints:

- Missing functional tests
- Missing quality attribute tests
- Manual deployments
- Hand-crafted environments

Frequency constraints:

- Delays waiting for code review or pull request approval
- Manual stages
- Waiting for management approval to deploy
- Too many branches or long-lived branches

For challenging policy constraints, there's evidence to support team autonomy throughout the deployment pipeline.

> **"We found that external approvals were negatively correlated with lead-time, deployment frequency, and restore time, and had no correlation with change failure rate."** - Forsgren, Humble, Kim. Accelerate (2018).

External approvals are a common policy constraint that delays the deployment of software. The cost of external approvals is time wasted, as they do not improve software quality or failure rate.

Technical capabilities for improvement

There are 7 technical capabilities that drive high-performance in Continuous Delivery. We discuss these in detail in our first white paper *The importance of Continuous Delivery*.

1. Trunk-based development
2. Continuous integration
3. Continuous testing
4. Deployment automation
5. Database change management

6. Monitoring and observability
7. Loosely coupled architecture

You should aim to adopt all these practices and increase your skill in using the techniques. Teams more experienced with the technical capabilities outperform teams who were new to them.

Keep an eye on the DevOps structural equation model[5] to stay up-to-date with these capabilities and their impact on software delivery performance.

Summary

Your deployment pipeline is the mechanism for inspecting and improving the frequency and quality of your software delivery. You can use the Theory of Constraints to find bottlenecks and focus efforts where they will most benefit the whole system.

You should introduce the technical capabilities that drive software delivery performance, and aim to increase your skills in those areas. It may take some time to be proficient in these practices, so prepare to hold your course when you first experience a dip in performance.

When rolling Continuous Delivery out to other parts of the organization, focus on teams who are open to the idea. Build a track record that will draw interest from other teams. Avoid hold-outs initially, as pushing a practice onto a team is rarely successful. Teams should adopt Continuous Delivery where there is demand. Don't force anyone to use it.

You can help teams build their deployment pipeline and everyone can share what they learn. And, ideally, you should make the knowledge global for your whole organization.

5 https://www.devops-research.com/research.html

Automation

Automation is not the goal of Continuous Delivery, but it's fundamental to achieving high-quality, high-frequency software deployments.

Many of your processes need human interaction, like innovating feature ideas, designing software, and exploratory and usability testing. Your motivation for automation is to free people from repetitive tasks to concentrate on higher-value work.

You'll find several natural automation candidates in your deployment pipeline:

- Builds (not just compilation)
- Tests *and* test data management
- Deployments
- Infrastructure provisioning
- Operations tasks

Continuous Delivery benefits from specialized tools to help with each stage of the deployment pipeline. Specialized tools for builds, deployments, and operations provide automation along with robust security and audit trails. This will help you build trust within your organization.

Summary

When you create a deployment pipeline, you create a single authoritative mechanism to get a change into production. With a deployment pipeline, you shouldn't need to skip processes when releasing an urgent change. The deployment pipeline should already provide quick resolution for failures.

Your deployment pipeline should stop an unfit-for-release change as early as possible, whether the failure relates to functionality, security, or quality attributes. Although the tests can't prove a version is good, it can prove a version is bad.

Approvals should be part of your deployment pipeline and should happen either automatically, or within the team.

It should be possible to complete the entire deployment quickly. For example, many times a day.

You should be looking for ways to improve the quality of the software and frequency of deployments. Using the Theory of Constraints can help identify areas to improve.

You may find our white paper *Measuring Continuous Delivery* helpful to your continuous improvement activities.

Further reading

To find out more about Continuous Delivery, the following titles by Jez Humble and Dave Farley provide a canonical reference:

- Continuous Delivery. 2011. Humble, Farley.
- Continuous Delivery Pipelines. 2021. Farley.

For DevOps, there are two essential reads:

- Accelerate. 2018. Forsgren, Humble, Kim.
- The DevOps Handbook (Second Edition). 2021. Kim, Humble, Debois, Willis.

If you enjoy the business novel format, the following books are a great way to picture the concepts:

- The Goal. 1984. Goldratt.
- The Phoenix Project. 2013. Kim. Behr.
- The Unicorn Project. 2019. Kim.

Octopus Deploy

Octopus Deploy Pty. Ltd.
Level 4, 199 Grey St
South Brisbane, QLD 4101, Australia

Email: sales@octopus.com

Phone: +1 512-823-0256.