Octopus Deploy

The importance of Continuous Delivery

Contents

Introduction 2

Part 1: What is Continuous Delivery? 3

Principles of Continuous Delivery 5

- Build quality in 6
- Work in small batches 8
- Computers repeat, humans problem-solve 10
- Relentlessly improve 11
- Everyone is responsible 13
- Summary 14

Continuous Delivery practices 16

- Loosely coupled architecture 18
- Continuous Integration and trunk-based development 19
- Continuous testing 20
- Database change management 21
- Deployment automation 24
- Monitoring and observability 26

Summary 27

Part 2: The importance of continuous delivery 28

Continuous feedback 29

Tangible benefits 32

Failure demand 35

The people factor 36

Return on investment 38

Case study 39

- Benefits reported 39
- Obstacles 41

Summary 42

References 44

Further reading 45

Introduction

Continuous Delivery is a software engineering approach that uses interrelated principles and practices to ensure software is always in a deployable state. The idea is to reduce the cost, time, and risk of delivering changes so you can deploy more often and check you're heading in the right direction. By testing your product and feature ideas sooner, you can avoid spending money on building features unimportant to your customers and move your focus elsewhere.

By reading this white paper, you'll understand:

- The principles and practices of Continuous Delivery.
- How Continuous Delivery's technical capabilities are fundamental to a successful DevOps adoption.
- The benefits of achieving high performance, explaining why Continuous Delivery is so important.

This white paper has two parts:

- Part 1: What is Continuous Delivery?
- Part 2: The importance of Continuous Delivery.

Part 1 defines Continuous Delivery, its principles, and necessary technical capabilities. You need to adopt and master these principles to achieve the benefits described later in the white paper.

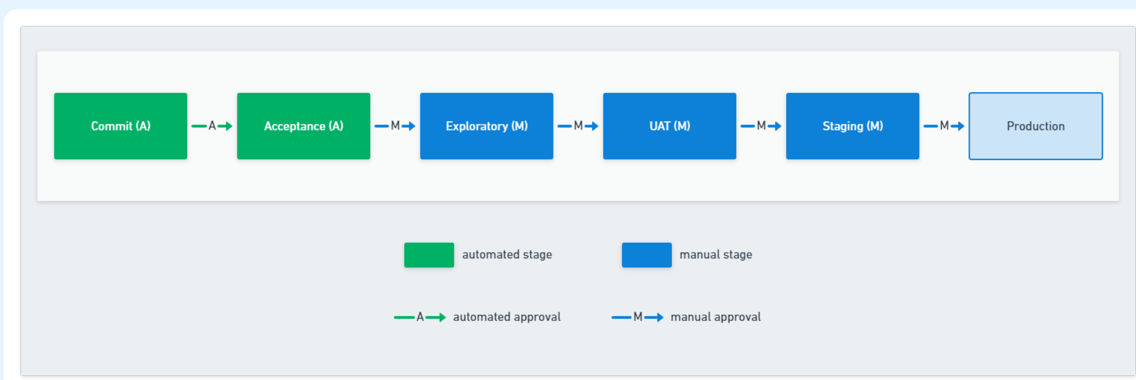
Part 2 highlights the benefits of Continuous Delivery. It explains how Continuous Delivery is statistically linked to high performance in software delivery, and how it helps organizations meet and exceed their goals.

If you're familiar with Continuous Delivery and DevOps, you can read Part 1 as a refresher or skip to Part 2.

Part 1: What is Continuous Delivery?

Continuous Delivery is five principles and seven technical practices that allow you to deploy changes safely, quickly, and sustainably. When you design your process around Continuous Delivery, you can deploy changes on-demand with high confidence.

To do this, you need to identify all activities needed to make your software *ready to go* and use them to build a *deployment pipeline*. A deployment pipeline is all the steps needed to get a code change from a developer's machine to the production environment. Your deployment pipeline may look like the example below, which shows a step sequence separated by approvals. These steps and approvals might be manual or automated, but it's essential to make them all visible when you map your pipeline.



Sample deployment pipeline

In the example, the developer commits their code change and, on a successful build, the pipeline automatically moves to the next phase - an automated acceptance test pack. If the acceptance tests pass, a tester can approve the change into the manual testing environment, where they do exploratory and usability tests. The change continues along the deployment pipeline, moving to the next stage each time it passes the last.

You may have one deployment pipeline for your whole application, or multiple pipelines handling different components, modules, or services.

> You might also hear the term *Continuous Deployment*, which is when *every change* deploys to production *instantly*. Continuous Delivery describes how you do this, but doesn't need you to deploy every software version immediately.

To understand Continuous Delivery in more depth, you need to know about the principles and practices involved. We cover these next, along with information on how Continuous Delivery is a crucial part of DevOps.

Principles of Continuous Delivery

You can use the five principles of Continuous Delivery to understand the reasons for the practices. They're based on well-established lessons learned in software and manufacturing. If you already know about Lean Software Development or the Toyota Production System, you'll recognize their influence here.

The principles are:

1. Build quality in
2. Work in small batches
3. Computers repeat, humans problem-solve
4. Relentlessly pursue continuous improvement
5. Everyone is responsible

You can find out more about each principle below. The example deployment pipeline used in the diagrams will update to show the affect each principle has, such as shorter lead times or earlier defect detection.

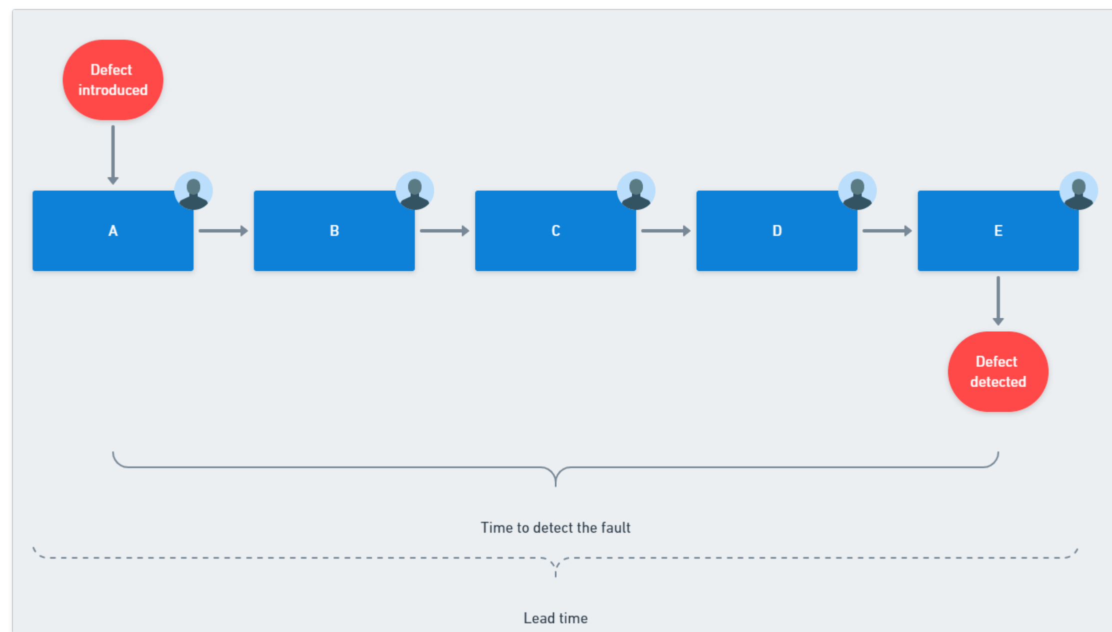


Diagram illustrating the delay between a defect and its detection

Build quality in

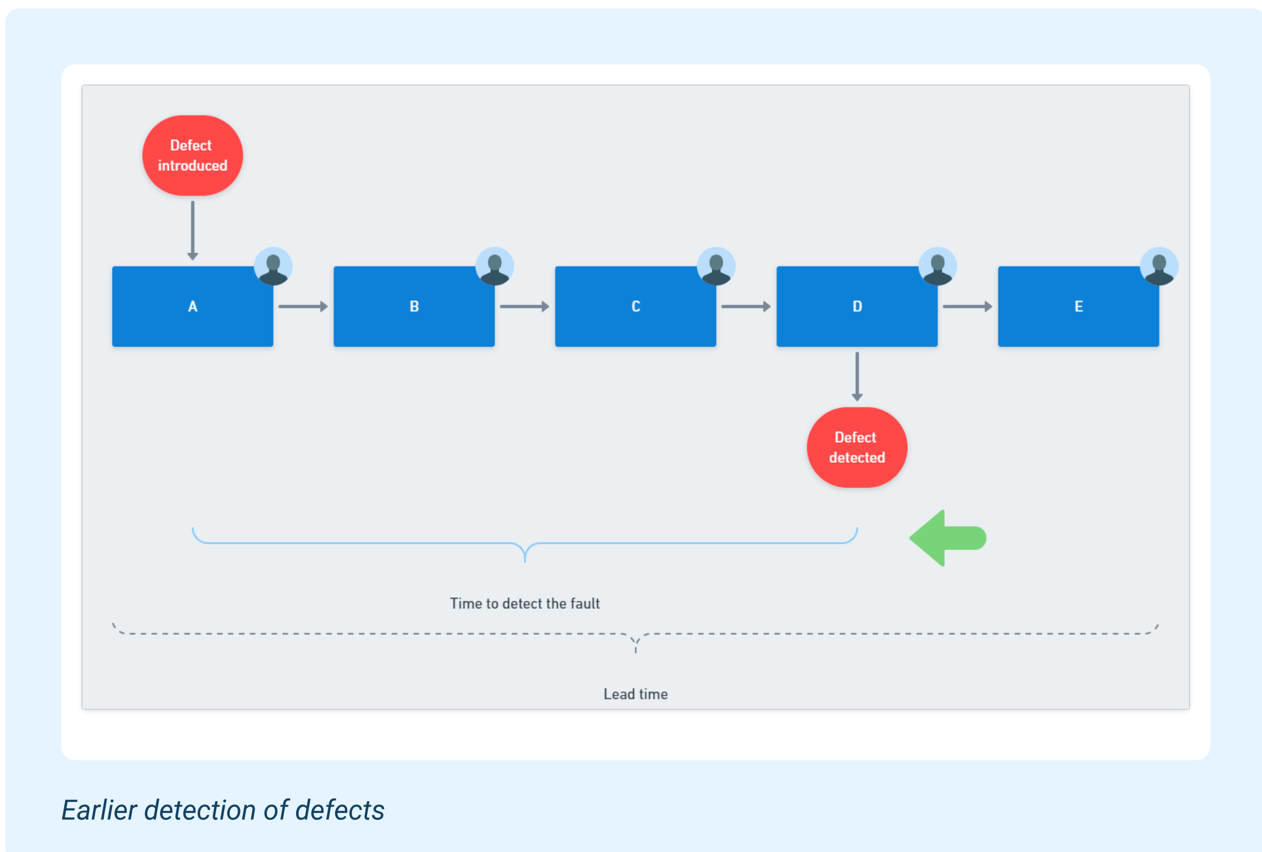
To *build quality in*, you must design your deployment pipeline to detect faults. When you find an issue, you should update your deployment pipeline to catch it earlier, if you can.

In the manufacturing industry, production lines had inspection processes that reviewed finished products and rejected those not meeting quality standards. The delay between completing a defective step and inspection at the end of the production line often resulted in many faulty products. Other workstations may have taken steps after the problem's introduction, meaning scrapped finished products and wasted raw materials.

The lean manufacturing movement realized you could solve issues faster and with less waste if production lines could immediately spot mistakes. Factories required each downstream process to raise a problem to the previous operation as soon as it detected defects. If necessary, a worker could stop the production line to allow everyone to crowd the issue and resolve it. No downstream work would happen on a faulty product, and factory workers would scrap fewer items.

Traditional software development methods worked the same as older production lines. The quality assurance process was the last thing to happen to the software before deployment to production. The software version tested would contain hundreds, or even thousands, of changes. By finding problems earlier in the deployment pipeline, there are fewer changes that could be responsible for the fault, and fixes can take place before new work starts.

Building quality in results in detecting problems earlier in the deployment pipeline.



Earlier detection of defects

Work in small batches

When you work in small batches, you reduce risk, get feedback faster, and find it easier to fix faults. If a small batch introduces a catastrophic change, you can discard the changes to return to a good state without losing much of your total investment.

In phased software development, teams passed large batches to the next stage. After weeks or months of work, developers would hand the software to a test team for validation. A problem found during testing might have been introduced months before detection. You could only deploy the software at the end of this long development and test cycle, after time spent fixing all the issues found. Organizations often try to manage the failures of large batches by introducing tighter controls, which slows the deployment pipeline and results in yet larger batches.

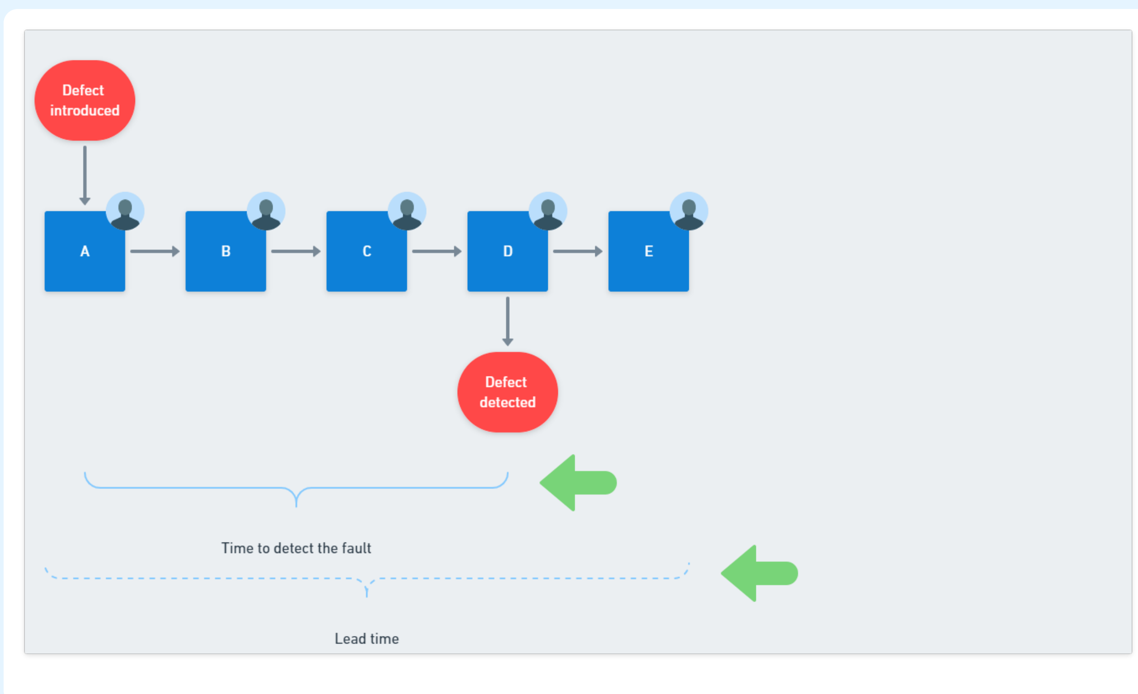
Each change you make to the software introduces risk. There may be a bug in the change, or the change might not be useful to users. When you batch changes, risk increases at a rate higher than the sum of changes. So two commits in a batch are *more* risky than the same two commits processed end-to-end one after the other. Even if you carefully test the batch many times, you only reduce functional risk - you don't reduce the risk of users or customers rejecting the change.

If you work to a calendar-based cycle, such as monthly deployments, the batch size will increase if:

- The team increases its delivery rate
- The team increases in size
- The date moves further out because of a problem

With large batches, each handover is more expensive. The natural response to these costly handovers is to do them less often to reduce expense, but this makes each hand-off cost even more. The solution is to *increase* the frequency as this will make the handovers less expensive. If you execute the whole deployment pipeline for every change, you flatten the risk to a single change. And if the change is bad, you can revert it to get back to a good state.

Working in small batches reduces time spent on each phase and results in faster feedback.



Small batches reduce lead times

Computers repeat, humans problem-solve

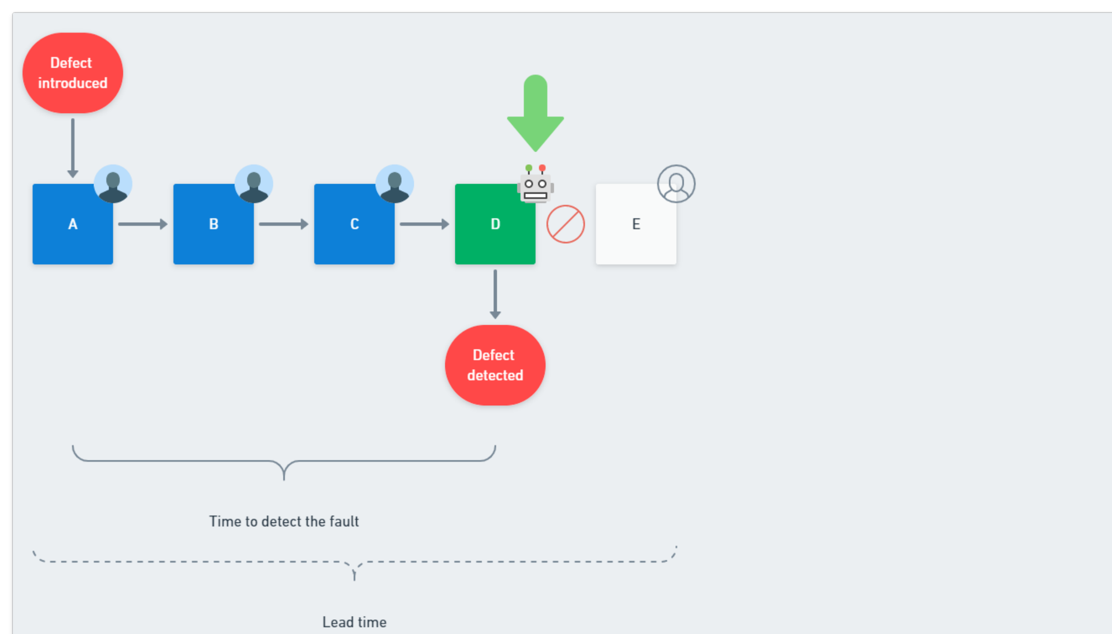
Humans and computers have different strengths. Computers can perform repetitive tasks consistently. They don't get bored or distracted and make mistakes. Humans are excellent problem-solvers who can generate innovative ideas. You should use this difference in your process design. Wherever you find a human performing a repetitive task, change it so a computer does the job and frees up the human to do more valuable work. For example, you should use computers to run standard regression test packs so testers have more time for exploratory and usability testing.

Where you can't fully automate a task, you can still find ways for computers to assist the work. For example, if you automatically generate a report on changed components, testers can focus their efforts on the right area of the software. Reducing the human effort of a manual step can improve human-decision accuracy, and free their time for more valuable activities.

Automation applies to more than testing. There are many other repeated tasks you can automate with high accuracy. For example, creating infrastructure for the application to run on.

As you seek to increase automation, look for opportunities to gather information for human approval and ways to set up fully automatic, machine-checked approvals.

Automating routine work plays to the strengths of machines and frees up time for people to do more valuable work.



Use human and computer strengths

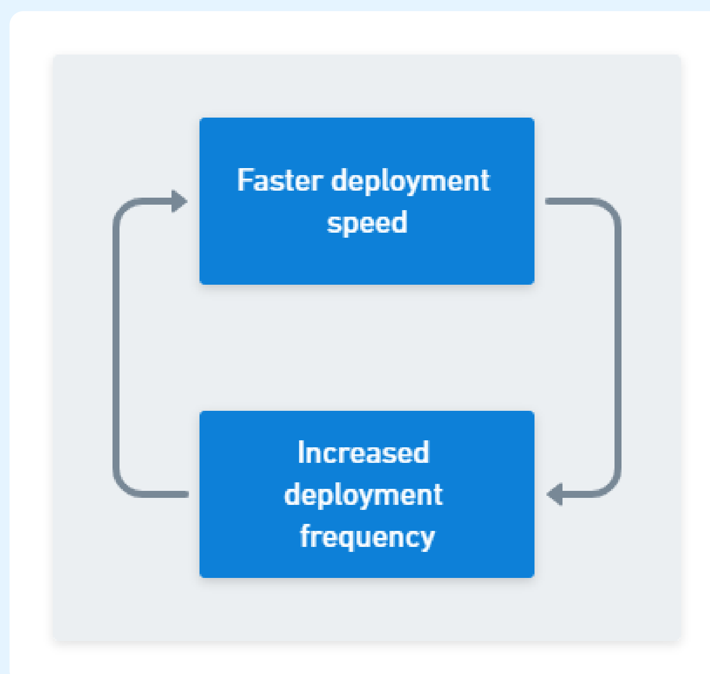
Relentlessly improve

Using the three principles of *quality*, *batch size*, and *automation*, you can find almost limitless areas for improvement. Continuous Delivery asks you to *relentlessly pursue continuous improvement*, which means working out how to:

- Reduce your batch size
- Shorten your lead time
- Detect problems sooner
- Ensure you divide tasks correctly between people and machines

When you introduce Continuous Delivery, you're likely to make big gains in a shorter time. Once you've reduced your lead time from months to days, improving it by a few more minutes might not seem worthwhile. However, you can amplify small improvements by increasing how often you execute the tasks. Small but regular gains are where your organization can find an advantage over your competition.

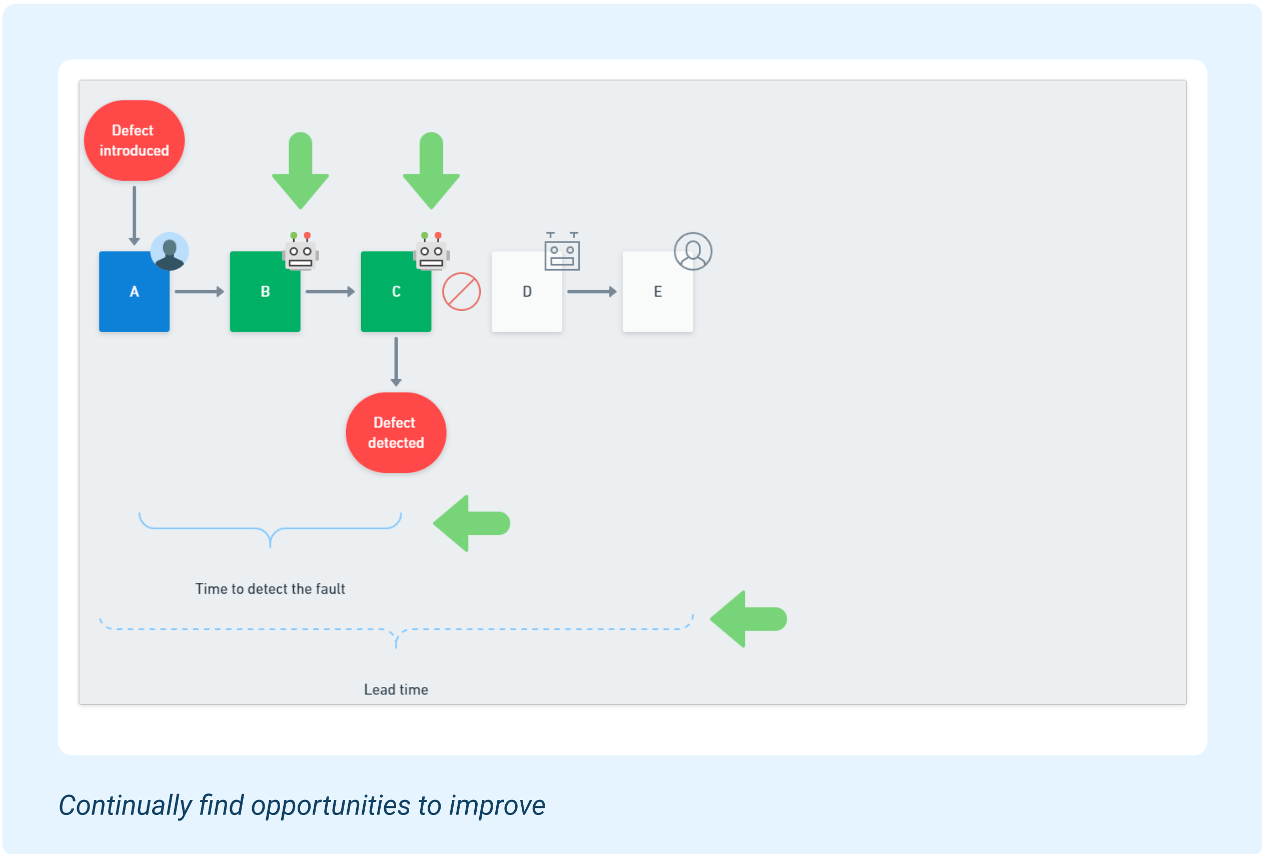
In a **survey conducted by Octopus Deploy in 2013**[1], deployment automation proved to hugely reduce deployment times and increase deployment frequency. Teams commonly moved from monthly deployments to daily deployments. While a fifteen-minute improvement is a modest gain on a monthly deployment, the payback increases to more than five hours per month when you deploy daily.



Relationship between deployment speed and frequency

You will find similar results elsewhere in your deployment pipeline, where seemingly small improvements become increasingly valuable as you deliver software more often.

The *relentless pursuit of improvement* drives the previous three principles and should result in fewer failures and shorter feedback times.



Continually find opportunities to improve

Everyone is responsible

Driving all previous principles is the most general cultural principle: *everyone is responsible*. There is no room in Continuous Delivery to compartmentalize problems. Where there's an issue, everyone involved in software delivery should share the responsibility for the resolution. Team members across all disciplines should work together to deliver software and achieve the organization's goals. Judge all improvements by their impact on the whole system.

To create a culture that aligns with this principle, you must embody the spirit of psychological safety to maximize learning. When something goes wrong, people need to feel comfortable speaking up quickly and honestly, which doesn't happen when a culture focuses on assigning blame. You also need to fix any reward structures that discourage collaboration. Suppose you reward developers for the number of features they deliver and testers for the bug numbers they find. In that case, each team member has an incentive to optimize for only their work, which will negatively impact software delivery and prevent your organization achieving its goals.

Rather than *forcing* everyone to take responsibility, management needs to nurture an environment of collaboration and ensure there are no incentives that discourage it.

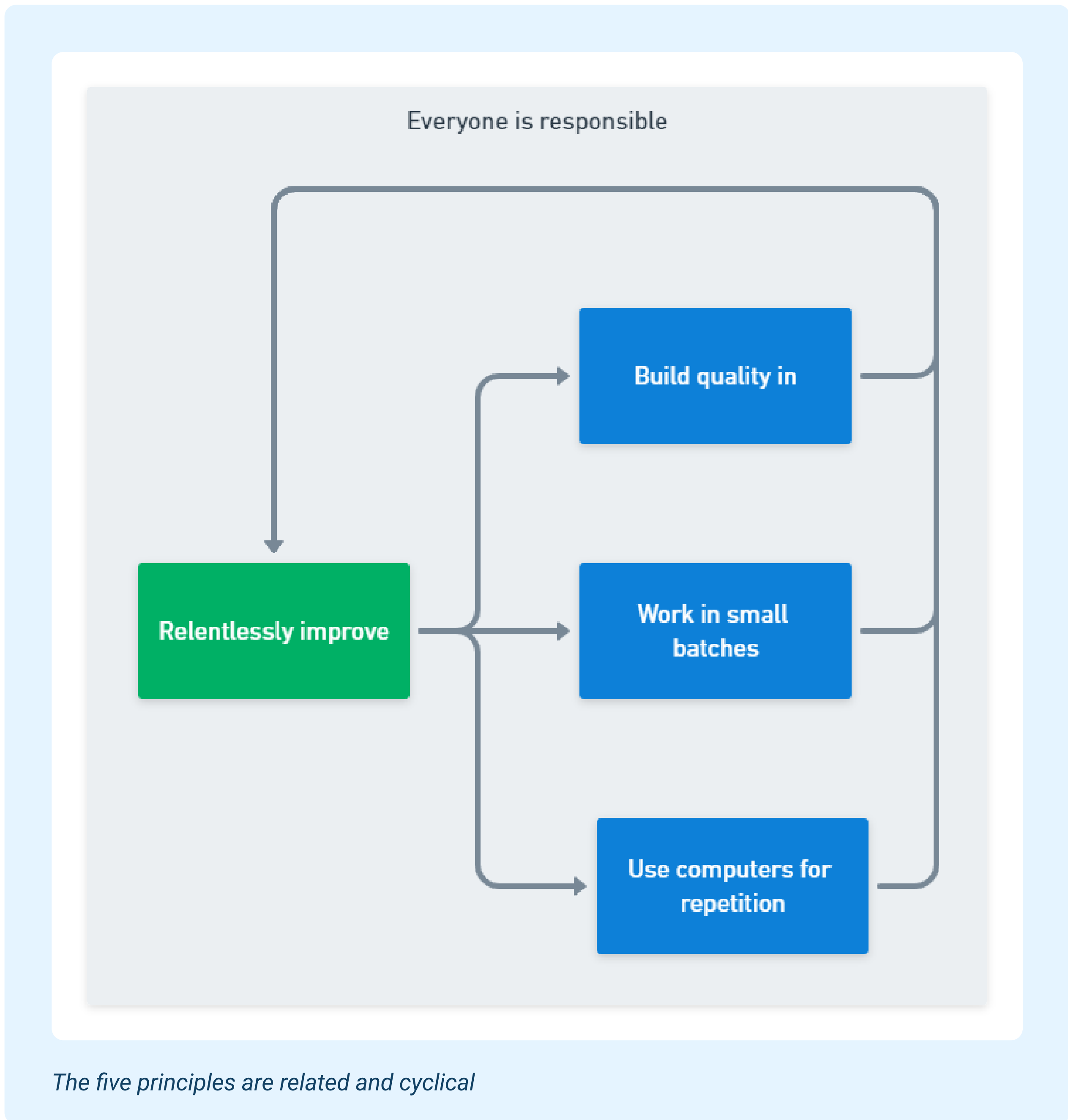
Summary

To recap, the five principles are:

1. Build quality in
2. Work in small batches
3. Computers repeat, humans problem-solve
4. Relentlessly pursue continuous improvement
5. Everyone is responsible

The principles are designed to help create a process and a culture that supports Continuous Delivery. The principles are tightly related to each other.

The fourth principle, *relentless improvement* underpins the first three of quality, small batches, and automation. This cycle amplifies the behaviors you need to constantly improve as part of your software delivery capability.



The five principles are related and cyclical

Many of your practices must become continuous to drive Continuous Delivery. You need to lower the cost of the deployment pipeline to ensure the cost to deploy a version of software is not a factor in your deployment decisions. You need technical skills and a strong culture that encourages collaboration and learning.

Now the principles are clear, you're ready to tackle the technical practices described in the next section.

Continuous Delivery practices

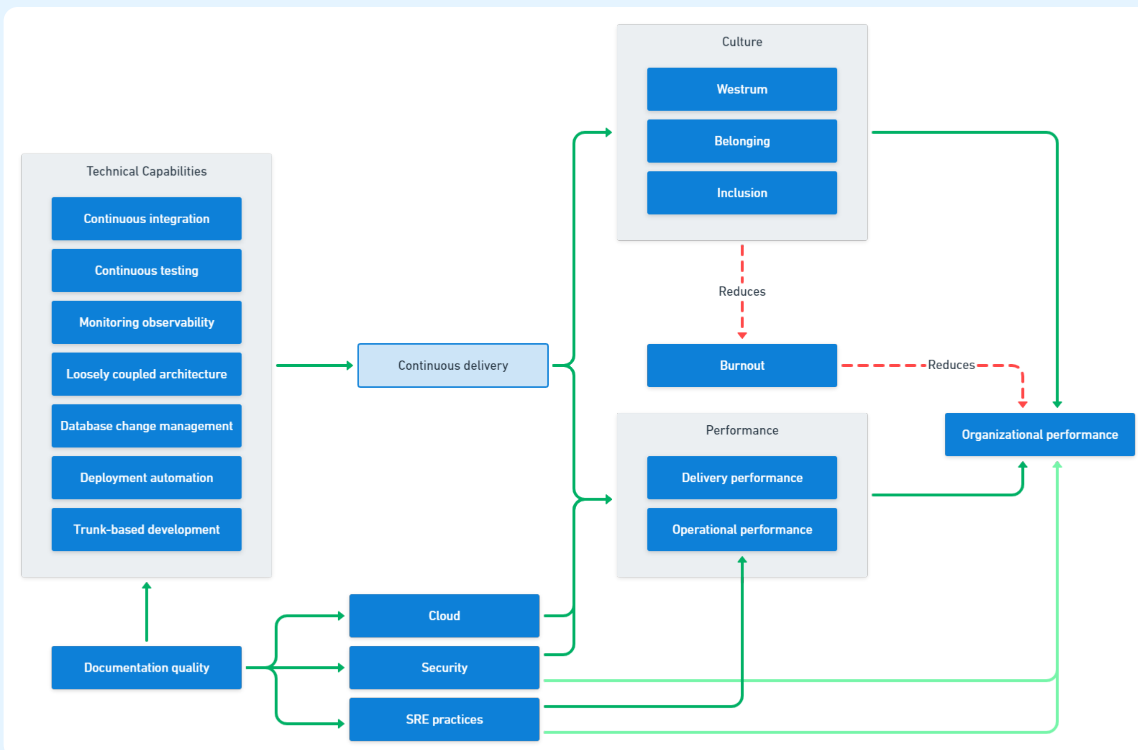
The State of DevOps Report (started 2011) studies adoption and how practices link to organizational performance. DORA (*DevOps Research and Assessment*) uses academic techniques to research and analyze capabilities driving high software delivery performance and positive organizational outcomes. The CD Foundation commissions surveys and reports to track Continuous Delivery adoption and impact.

The DORA research finds that Continuous Delivery depends on seven significant capabilities:

1. Continuous Integration (CI)
2. Continuous testing
3. Monitoring and observability
4. Loosely coupled architecture
5. Database change management
6. Deployment automation
7. Trunk-based development

The **structural equation model**[2] diagram below shows the predictive relationship between:

- Specific technical capabilities
- Continuous delivery
- Other drivers of technical and organizational success



The DORA structural equation model

> A *predictive relationship* uses a technique called 'inferential predictive analysis' for testing. This is a scientific approach used to test theories in real-world situations, rather than experimental models. DORA uses this technique to discover and test relationships shown with arrows in the structural equation model above.

We explore the technical capabilities in more detail below, in the order you might see within your deployment pipeline. You can find the benefits reported in the various studies in Part 2.

Loosely coupled architecture

2017's State of DevOps report found that architecture is Continuous Delivery's biggest predictive indicator. To be *loosely coupled architecture*, your application should be made of small, independent components with their own deployment pipeline. You can deliver faster because you only need to run the deployment pipeline when you change a component. The stages will take less time because there is less code to build and test in a component than there is in a whole system.

You also need to carefully manage dependencies between components and the teams that build them. This helps avoid an architecture that appears decoupled yet acts like a single large application. If you find a change triggers cascading updates through your components, dependencies are likely to be the problem. Equally, if a change to one component needs tests to cover other components, your architecture may be too tightly coupled.

To achieve the goals of a loosely coupled architecture, you need to align your team and architecture. There is little benefit to having many active components maintained by a single team or many teams tripping over each other. Alignment issues caused by an imbalance between people and architecture leads to a build up of undeployed changes, visible in your lead times.

> Conway's Law is an adage that says that a system's design will mirror the communication structure of the organization that created it. Both MIT and Harvard Business School found strong evidence to support this 'mirroring hypothesis'. This is good if you intentionally design your communication structure, as it will likely result in the correct architecture. Re-designing team structure to drive architecture is the 'inverse Conway manoeuvre'.

Continuous Integration and trunk-based development

Continuous Integration means merging changes into version control often. Usually every few hours but at least once a day. After running the build and test cycle locally, developers should push their working version into the trunk (or main line) in version control. For each of these commits, your deployment pipeline should automatically run the build process. Your build should:

- Compile the code
- Run a set of static analyzers
- Run a set of fast automated tests that will reveal any serious problems you must fix immediately

The result of your Continuous Integration process should be a validated package you use throughout your deployment pipeline. Your package should be canonical, which means using the same package every time you deploy the software version (rather than the same code re-built for each environment).

> Trunk-based development was always the intention of Continuous Integration, but the definition is now a separate capability based on the analysis collected in the State of DevOps Report.
>
> Trunk-based development is where you commit code directly to the main branch in version control. The analysis by DORA found benefits still apply providing you use no more than three branches, and they're merged into the main line in less than a day.

Continuous testing

Your deployment pipeline should allow testing for every software version. In your pipeline's earlier stages, automate tests to be reliable and fast so developers can get near-immediate feedback. Your developers should write and maintain these tests. Externally written tests don't predict performance in Continuous Delivery.

You should divide your tests into sets that allow fast and automated packs to run before longer-running or manual tests. Move individual tests to the earliest stage possible without making that stage take too long to run. With this design, you find out as early as possible when you have a bad version of your software and avoid wasting time running slower and manual tests. You may also be able to run some test packs in parallel to reduce your overall lead time.

Types of testing

As well as tests that check intended functionality, you must also test component security and performance. For example, you can run security scans and response time tests as part of the deployment pipeline. You can even include tests that check your infrastructure and configuration.

Where you have long-running characterizing tests that check application and resource use over time, they should run outside the deployment pipeline. These tests should form part of your monitoring and alerting strategy. For example, it's common to use real-user monitoring to track response times long-term, and your monitoring tools should alert you to sudden performance changes and degradation over time.

Realistic test environments

You should test software in a production-like environment as this makes the production deployment more likely to succeed. If your pre-production environments don't resemble the live environment, configuration differences may cause the production deployment to fail.

> A crucial part of continuous testing is test data management. If the data used for tests is unstable, you will get inundated with false failures. You should automate test data creation to ensure tests run against a consistent data set every time.

You should always try to find errors with the fastest test set that can detect the problem. If a slow integration test finds a bug, try to design a faster test that would detect the issue earlier.

Database change management

During a deployment, it's common for two versions of the application to run against the database:

- The live version
- The new version that you are putting live

Even with one application running against one database, there will be a short window where a newer database version runs alongside an older application version. In more complex setups, there could be a mix of application versions running against the same database. To support this, we need to put in place a *database change management* strategy.

There are many nuclear **database refactoring**[3] techniques that allow you to perform multi-step safe refactorings. There are also two general approaches to database change management:

- Create two databases
- Decouple database changes

Create two databases

With the two database strategy, you have a version of the database for each application version. During the deployment of a new version, you put the production database into a read-only state and take a backup. You then use the backup to populate a new database, which you can upgrade with all the changes.

The new application version connects to this updated database. So by switching traffic over to the new application version, you move all traffic away from the old application and database. Once all traffic flows to the new application and database, you can decommission the previous version.

To successfully use the database-per-version strategy, you must automate the processes and ensure each step is fast:

- Backup
- New database creation
- Upgrades
- Pre-flight checks
- Traffic management

Decouple database changes

You can decouple the dependency between the database and application using the expand and contract pattern. This allows you to add new items to the database as additions don't impact the running application. However, you can only delete an item if no running version of the application depends on it.

Breaking changes must be carefully handled to allow previous application versions to continue using the old field, and for newer versions to use the new field. For example, we might want to change a column name from 'surname' to 'lastname' to make it consistent with terminology used in the organization. The process to rename the column is below, with each step deployed to production before moving to the next:

1. Add a new 'lastname' column to the database and create a synchronization trigger to ensure 'lastname' updates whenever 'surname' changes
2. Deploy!
3. Write a migration to copy data from 'surname' to 'lastname'
4. Deploy!
5. Update the application to use 'lastname' and remove all references to 'surname'
6. Deploy!
7. Delete the synchronization trigger and 'surname' column, to prevent its use by mistake (instead of 'lastname')
8. Deploy!

You should include data migration as part of your deployment pipeline. If you need to plan a long-running migration, consider using a runtime data migration strategy or a long-running process to update the records.

A runtime data migration strategy moves data at the point of access. For example, when retrieving a customer record, the last name would update if the field is empty. This allows a gradual population of most data, but the old surname column must remain until all records update.

A long-running migration process could be a throttled update script. This would detect records with a last name that isn't the same as the surname and update a few at a time. This allows all records to update without impacting application responsiveness for users.

Deployment automation

Deployment automation allows frequent deployments that are both reliable and repeatable. Using the same automatic process to deploy the software to all environments, you can test the deployment process as often as the application code. You should also use deployment automation beyond software packages. For example, automating environment creation and infrastructure based on definitions stored in the same version control repository as the application.

When you use an automated deployment process, you can make sure the same steps happen in the right order each time you deploy. Crucially, you can guarantee you miss no steps.

This means you can allow self-service deployments to different environments, rather than depend on another team to push changes. There are **many observable features of good deployment automation**[4], such as:

- Repeatability
- Recoverability
- Visibility
- Auditability

There are also several **deployment patterns**[5] that can help reduce downtime. These include rolling deployments, blue/green deployments, and canary deployments.

You can decouple your deployments (putting a software version live) from your releases (making a feature available) using techniques such as feature flags.

> Feature flags, or feature toggles, allow you to make a feature available independently of deployments. You can use them to:
>
> - Remove a problematic feature without having to redeploy an older application version.
> - Make a feature available to subsets of your users, either as part of a beta program or for multivariate or A/B testing.
> - Gracefully degrade performance by temporarily removing a resource-intensive feature when there is high traffic or infrastructure issues.

In a **survey conducted in 2013**[1], organizations found automating their deployments with Octopus allowed faster and more frequent deployments. Organizations using deployment automation were five times more likely to complete deployments in under thirty minutes. And more than eleven times more organizations reported daily deployments after introducing automation.

Capability	Manual deployments	Automated deployments
Deployments under 30 minutes	15%	**86%**
Deployments over 2 hours	**45%**	4%
Daily deployments	5%	**59%**
Monthly deployments	**61%**	8%

To break this down, for a team member responsible for a manual deployment that takes eight hours:

- A monthly deployment takes 5% of their annual working hours
- Weekly deployments would take 18% of their time
- Daily deployments would be 92% of their role

Deployment automation is self-funding and essential for increasing deployment frequency.

Monitoring and observability

Monitoring first makes information from many sources visible in the same place, then allows you to categorize and learn from the data. You can use what you discover to design automated alerts that sound an alarm when something isn't right. You can check infrastructure properties, such as resource use, and also track specific application features and business metrics.

By including business metrics in your monitoring and alerting strategy, you can tell when a deployment or feature release affects your organization's goals. You can respond to business metrics the same way you respond to downtime. For example, a hotel might detect and react to a drop in average booking value, which might indicate a pricing error or rogue voucher code.

You can use business monitoring metrics alongside canary releases or multivariate testing to compare performance of different software versions.

We've listed some monitoring metrics you may not have considered below:

- The number of deployments to production
- Build times
- Test run times
- Use per feature
- Conversion rates
- Financial data
- Cloud costs
- Complaint numbers

If you bring all these metrics into a single location or tool, you can match them to specific software versions. You could quickly respond to changes, such as a version causing more complaints, or an unexpected rise in cloud costs. When you discover the relationship quickly, you can learn what works and what doesn't.

Summary

The technical capabilities of Continuous Delivery are not always easy to put in place. In some cases, they need big changes to how you work. However, you can start small and iterate your way to success. Unlike other software delivery approaches, large-scale research and statistical analysis backs the benefits of that hard work, and proves the predictive relationship between them.

Each practice contributes to short feedback cycles, so you can run your deployment pipeline more often and release software more frequently. This links back to the Continuous Delivery principles discussed earlier.

Now we have a clear definition of Continuous Delivery, we can look at its importance for teams and organizations.

Part 2: The importance of continuous delivery

Prior software delivery methods were based on experiential evidence, themselves based on practices working for particular teams. This was an improvement over processes and frameworks based on theoretical models about what *should* work. Continuous Delivery and DevOps take this further, with extensive research and theory-driven scientific analysis. This is the highest bar taken for assessing the effectiveness of a software delivery approach so far.

When you read the benefits in this part of the white paper, they're backed by research from the Continuous Delivery Foundation, DORA, and other research-backed surveys. Instead of single-case studies, this research represents data collected from thousands of organizations.

Published by	Report	Respondents
DORA	The State of DevOps Report	More than 32,000
CD Foundation	The State of CD Report	More than 19,000

In Part 1, we described the technical capabilities in detail. This is important because the research relates to a specific Continuous Delivery definition. The principles and practices have an amplification effect on each other. As you adopt more of them, you are more likely to get the benefits described below.

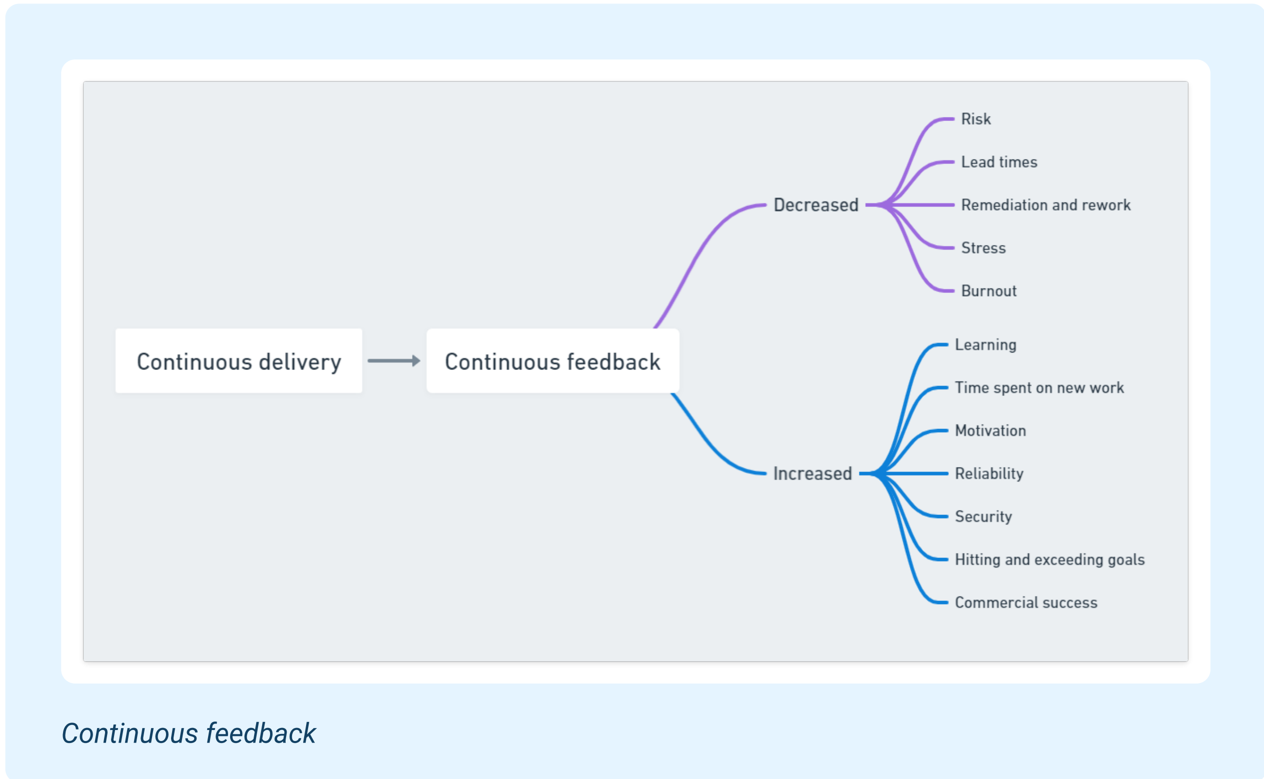
Continuous feedback

Continuous Delivery has many technical elements, but they all help shorten the feedback cycle. This lets you learn faster and change direction sooner, if you need to. This gives you a competitive advantage that can help you meet or exceed your organization's goals.

Where traditional software delivery methods try to manage risk by deploying less often, research shows this has the opposite effect. The deployment itself isn't the risk source, it's the changes introduced. In particular, the more changes you make before you deploy, the higher the risk. We share the statistics for this later on.

By increasing deployment frequency, each batch has fewer changes and less risk. This applies across all risk factors, like service outages, security, and quality. So you must bring these practices into the deployment pipeline and make them *continuous*.

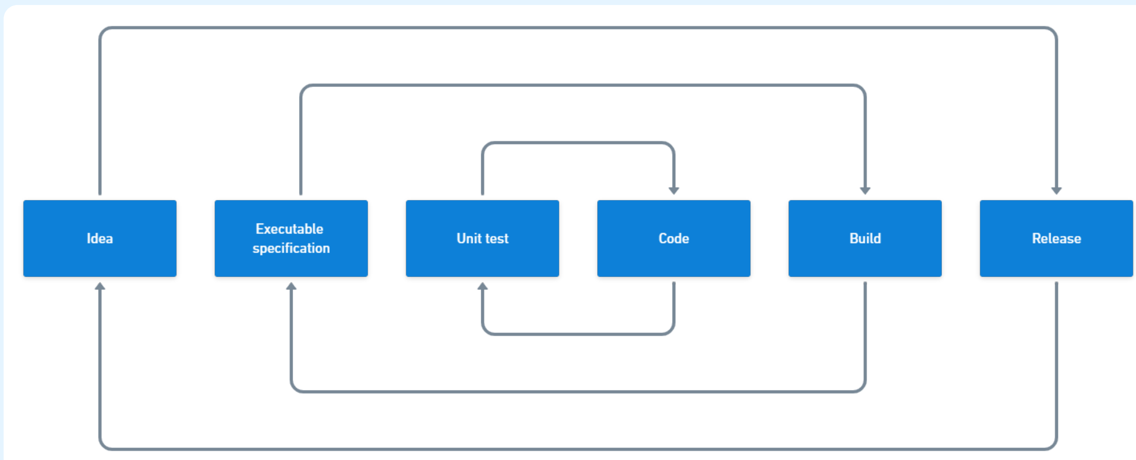
The diagram below shows Continuous Delivery driving constant feedback. This has a lowering effect on risk, lead times, rework, stress, and burnout. It also increases motivation, stability, time spent on new work, and commercial success.



Continuous feedback

Receiving feedback is a layered and non-linear process. If you place the code and test cycle in the center of a diagram, the feedback from each extra step wraps in a series of concentric circles, like an onion.

Many of these feedback layers are only internal indicators. Real feedback comes when the feature is available to users. These intermediate feedback loops are useful but can never confirm the feature will be a success. This is one reason you must shorten the lead time between having an idea and showing it to your customers.



The layers of feedback form an onion

When you use large batches, the genuine feedback from the outermost loop takes longer to arrive. When feedback's delayed, you might start treating internal feedback as a substitute for the real thing. When you make decisions based on proxy feedback, you increase the risk your users will reject the feature.

You need to make sure you get rapid and genuine feedback, and that usually involves proving you've solved a problem for a customer in a valuable way.

Tangible benefits

The State of DevOps Report categorizes organizations based on an assessment of their capabilities as described in Part 1. Based on this, they rate each organization as low, medium, high, or elite performers. This allows comparison of the four groups in the outcomes associated with the different levels of Continuous Delivery and DevOps adoption.

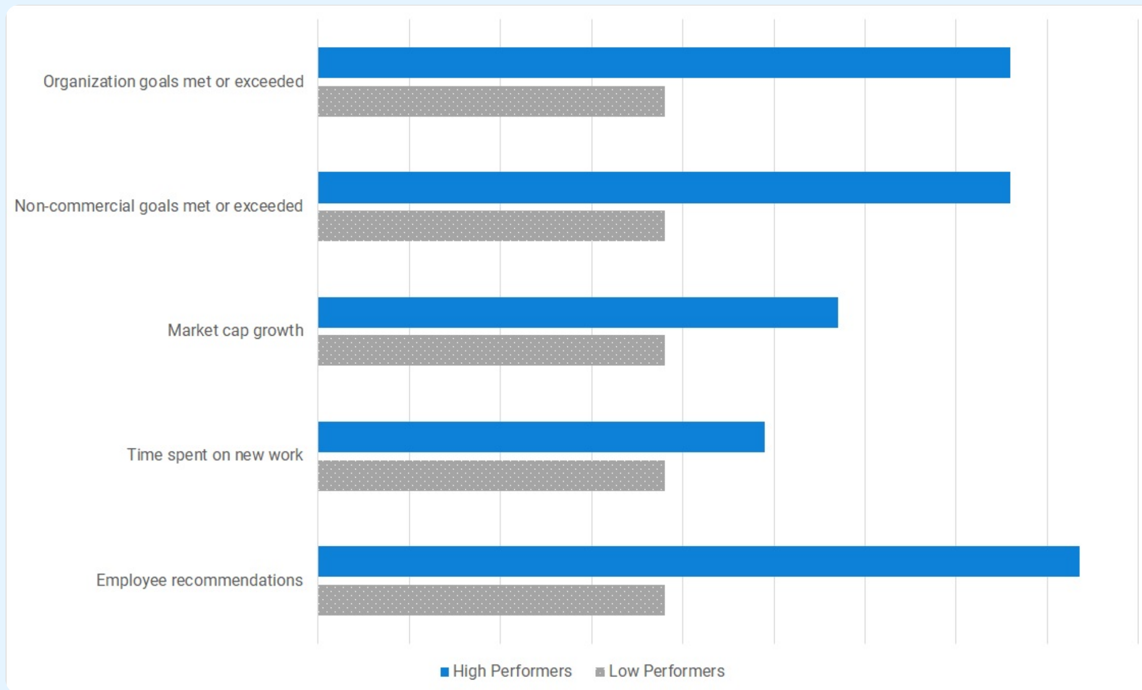
> An interesting finding in the report is the medium group under-performs in some areas as they work to introduce the principles and practices. If you start in the low-performance category, prepare for a shaky start that pays off once you hit high performance.
>
> Think of yourself as an aspiring Olympic athlete. You must swap the techniques that worked at regional level with high-performance techniques that slow you down as you first practice them. Once you master the new techniques, you can hit your personal best and take a shot at the podium.

We compare the low-performance group to the high performers for the following statistics. If you manage to meet the elite performance level, you can expect even greater benefits.

High performers are twice as likely to meet or exceed the organization's goals, such as productivity, profitability, and market share. They're also twice as likely to exceed non-commercial goals like quality and customer satisfaction. People working for high performers are 2.2x more likely to recommend working at the organization. They also spend eleven percentage-points more time on new work.

In 2014, market analysis found that high performers had 50% higher market cap growth than low performers.

Find a comparison of high versus low performers below, with high performers excelling in all measures. Increased deployment frequency is a key factor in these differences, driven by Continuous Delivery and its technical capabilities.



High performers vs. low performers

The **State of Continuous Delivery Report**[6] (compiled by developer economy analyst SlashData™ for the Continuous Delivery Foundation) echoes the contrast between high and low performers. The report, found that delivery speed and stability go hand-in-hand.

Though it's intuitive to believe that speed and stability represent a trade-off, delivery speed in fact predicts stability. The high performers for lead time were five times more likely to excel in stability than teams with long lead times. The table below shows lead times along the top and time to restore service on the left.

	> 1 month	**1-4 weeks**	**1-7 days**	**< 1 day**
> 1 week	**52%**	28%	17%	8%
1-7 days	19%	**24%**	21%	16%
1 hour to 1 day	23%	37%	**43%**	**46%**
< 1 hour	6%	11%	19%	**29%**

Highlighted cells are five percentage points above other performance segments.

In this table, you find speed and stability together. Slowing delivery to increase stability is a myth, as slow delivery is more likely to result in unstable systems.

Failure demand

In *Freedom from Command and Control*, John Seddon describes a concept called *failure demand*. This is work caused by a failure to do something, or do something right, for the customer. Because failure demand takes time and attention away from regular business, it's likely you'll see more failures in the future. This vicious cycle drastically reduces the capability across the entire business.

In software delivery, we refer to failure demand as bug fixing, remediation, or rework.

A case study at HP in The DevOps Handbook found Continuous Delivery increased time spent on new features from 5% to 40%, and reduced development costs around 40%. This needs discipline in detecting issues earlier in your deployment pipeline where tests are faster and bugs are cheaper to find and fix.

As you reduce the failure demand, you increase your capacity for new work without more hires or longer working hours. This gives you more capability to deliver features without the drawbacks linked with scaling the team or working at an unsustainable pace.

The people factor

DORA research found Continuous Delivery improves the way people feel about work and their organization. Staff experiencing less deployment pain are less likely to suffer burnout, and have a more positive view of the organization. The technical capabilities contribute to a strong team identity and reduce friction.

High performers do less manual work than low performers. People don't enjoy doing repetitive tasks, so it's likely work is more enjoyable when there are fewer routine tasks to do. Google refers to this routine and repetitive work as *toil*.

Manual Work	Low Performers	High Performers
Configuration Management	46%	28%
Testing	49%	35%
Deployments	43%	26%
Change approval	59%	48%

People are also motivated by seeing their work in the hands of their customers rather than checked into source control. Especially as features become more valuable to users as a result of faster feedback cycles. When people stay on the team longer, they keep essential knowledge, and you spend less time on recruitment and on-boarding.

A **DevOps trends survey**[7] commissioned by Atlassian found that DevOps improved many aspects of software delivery performance:

- 61% of organizations increased quality.
- 49% had a faster time to market.
- 52% saw faster recovery times.
- 99% said DevOps had a positive impact.

A key survey finding was performance increases with DevOps experience. This is also a finding of the State of Continuous Delivery report, which found higher performance where there was more software development experience.

If you want to scale a team, there's a limit to how much change you can realistically push per deployment. This means you must pay attention to a loosely coupled architecture and increase deployment frequency to keep batches small. To achieve a high deployment frequency, you need to automate more. This is where Continuous Delivery becomes not only desirable but essential. Your ability to add developers to your organization depends on the technical capabilities linked to Continuous Delivery. Your deployment pipeline sets the pace of change for your whole business.

You can still have manual approval steps, as long as the human is performing a valuable task as part of the approval. Where possible, approvals should happen automatically if the phase meets requirements. Each component should move through the deployment pipeline without waiting for other components, otherwise a problem in one component could stop all deployments. This is another reason to target a loosely coupled architecture.

The deployment pipeline performance will set the pace for the team. If changes can't progress any faster through the deployment pipeline, adding more team members will have no effect on software delivery performance.

Return on investment

Google Cloud and DORA created a **return-on-investment (ROI) model**[8] for organizations that want to move from IT treated as a cost center to IT as a value-driver for the business. The model factors in the four categories listed below:

1. Efficiency - reduction of non-value adding activities, such as rework and failure demand
2. New features - Increasing time spent on new revenue-generating features
3. Faster time-to-resolve, decreased downtime, and reduced downtime-related losses
4. Retention - high-performing teams are more than twice as likely to recommend the organization as a great place to work.

One example shows the return on investment for DevOps can be ten times the original investment with a short payback-term of one month.

If you focus only on cost reduction and ignore value-generating activity, you end up in a commoditization *race to the bottom*, where your prices must get lower over time. Value-generating activities maintain and increase your value to customers, protecting your revenue.

Case study

A **case study reported in InfoQ**[9] explored Continuous Delivery adoption at a company with 4000 employees and €6 billion turnover. They had 400 technical team members in teams commonly of four or eight, who used a mix of technology stacks (Java, Ruby, PHP, .NET). Here's a summary of the benefits and challenges discovered in a study of the first 20 internal business applications that adopted Continuous Delivery.

Benefits reported

Building the right product

- Frequent releases let application development teams get user feedback earlier, allowing more focus on useful features.
- If they found a feature wasn't practical, they spent no further effort on it.

Accelerated time to market

- Releases were more frequent and made available as soon as they were ready.
- Feature cycle times moved from months to just 2-5 days.

Improved productivity and efficiency:

- Significant time savings for developers, testers, and operations engineers through automation.
- Recovered 20% of time by making pre-live deployments a push-button activity.
- Live deployments went from days of work to a push-button activity.

Reliable releases

- Risks associated with a release significantly decreased, and the release process became more reliable.
- Found and fixed issues with the deployment process in pre-live deployments.
- Fewer changes included in each release, making it easier and faster to fix problems.

Improved product quality

- The number of open bugs and production incidents decreased significantly.

Improved customer satisfaction

- Achieved a higher customer-satisfaction level.

Obstacles

Organizational challenges

- The further your organization is from the intended design (the more divisions involved), the greater the challenge.

Process challenges

- Processes intended as quality gates or compliance requirements needed review to assess impact on throughput or workflow.
- If a team can deliver a feature in two days, but there is a two-week approval step, you must elevate this as the key constraint to resolve.

Technical challenges

- Attempting to create your own tools can be expensive and may not play to your strengths.
- Existing architecture may be challenging to bring into the deployment pipeline.

Summary

There is a strong link between Continuous Delivery and DevOps adoption and many organizational benefits. This relationship was tested using rigorous statistical approaches from tens of thousands of survey responses.

The structural equation model shows seven technical capabilities are drivers for Continuous Delivery. This is a significant predictor of the culture and performance delivering commercial and non-commercial goals for an organization. This software delivery method is the most thoroughly tested mechanism, and the best for delivering software we know of so far.

Because the practices have an amplifying effect on each other, you need to adopt them broadly. You also need to continually learn and improve based on the feedback you get through the deployment pipeline.

We highlighted some key points to remember below:

- Include everything needed to deliver software in your deployment pipeline.
- Everything is a target for automation.
- You should track security issues in the same place you track bugs and features.
- You don't have to get to 100% on day one. The process is one of continuous improvement.
- Make everything you learn available to the rest of the organization.
- You should aim to improve both flow (left to right) *and* feedback (right to left)

Continuous Delivery benefits include:

- Faster feedback
- Lower risk
- Less rework and remediation
- More time on new work
- Lower stress and higher motivation
- Organizational success
- Empowering teams
- Deployment flexibility

Thank you for reading this white paper on the importance of Continuous Delivery.

You can find more white papers on **octopus.com**[10]

References

1. https://download.octopusdeploy.com/files/whitepaper-automated-deployment-octopus-deploy.pdf
2. https://www.devops-research.com/models.html
3. https://www.databaserefactoring.com/
4. https://octopus.com/blog/ten-pillars-of-pragmatic-deployments
5. https://octopus.com/blog/common-deployment-patterns-and-how-to-set-them-up-in-octopus
6. https://cd.foundation/reports/
7. https://www.atlassian.com/whitepapers/devops-survey-2020
8. https://cloud.google.com/resources/roi-of-devops-transformation-whitepaper
9. https://www.infoq.com/articles/cd-benefits-challenges/
10. https://octopus.com/resource-center

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 Depl

Octopus Deploy
Level 4, 199 Grey St
South Brisbane, QLD 4101,

Email: sales@octopus.

Phone: +1 512-823-025

octopus.com