Octopus Deploy

Measuring Continuous Delivery and DevOps

Contents

Introduction

You might be introducing new practices and capabilities as part of Continuous Delivery and DevOps adoption, or making changes as part of continuous improvement. In either case, you need a way to tell if changes are improving your ability to deliver software and, ideally, if they help your organization achieve its goals.

In this white paper, you find several approaches for measuring your progress. There are statement-based assessments and metric-driven measurements. You can use them at different times or combine them to create a custom view of your organization.

Your deployment pipeline should generate 2 kinds of feedback:

- Fast feedback that tells us quickly if something is wrong
- Real feedback from customers and users, which you can only get by putting software into production

> **If we take feedback seriously, we want lots of it.** - Dave Farley

When designing measurements for your deployment pipeline, software delivery, and organizational performance, you must find a balance between feedback that:

- You can get fast and often
- May not be as readily available but reflects actual outcomes

You can use several approaches to measure your DevOps and Continuous Delivery performance. You can also combine different methods to suit your needs.

- *Continuous Delivery statements* prompt you to re-focus on the key goals of your deployment pipeline
- *DORA (DevOps Research and Assessment) metrics* are a simple way to measure your software delivery and operational performance
- *The SPACE framework* extends measurement to business outcomes

Before introducing these frameworks, here's an overview of metric design to give you a healthy set of options for measuring software delivery.

Metric design

The frameworks for measurement provide a starting point designed to avoid common metric problems. A poorly designed measure can cause people and teams to work against the organization's goals. Getting your metrics right is crucial to a successful outcome.

As you eventually want to branch out from the canned frameworks, this section has tips for designing robust metrics with minimal unintended side effects.

Start purposefully

When anyone introduces a measurement, they usually intend to cause a positive change. Despite good intentions, it's common to get a different shift than expected. You can increase your chance of success by being open about the purpose of the measurement.

Before you start collecting data, ask yourself these 2 questions:

1. How will I respond to this measurement?
2. How might other people behave differently?

For example, here are sample answers for measuring build times:

How will I respond to this measurement?

If builds take more than 5 minutes, I will look at ways to make builds quicker because fast feedback for every change is essential.

If builds take less than 5 minutes, it might hint we're missing an opportunity to detect faults early in the process.

How might other people behave differently?

People might remove important items from the build to make it faster, reducing the feedback's value. Managers might track build times per developer to use as a performance measurement. They might make people spend unreasonable amounts of time working on builds and infrastructure.

You should publish the answer to the first question. By being up-front about the metric's purpose, people will understand what you hope to achieve. Often, they will help you achieve the goal, even when the metrics are imperfect.

Your answers will also give you ideas for collecting and displaying data to avoid undesirable behaviors. For the build times example, you could communicate why you want them to remain around the 5-minute mark. When you visualize build times, you can add fixed guides to charts to show the desired range and set up any automated alerts to align with your goals.

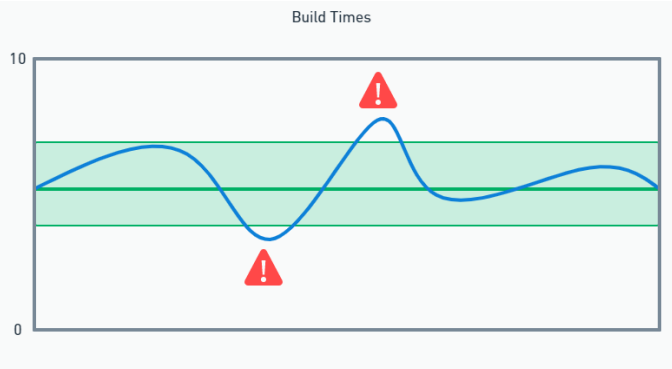


Chart with min and max guidelines

The most common unintended consequence of measurement is local optimization. This is where an individual or team increases their output at the cost of the broader value stream. For example, if you measure individual output, people may be less willing to pause work to help out a blocked team member. There will also be less motivation to coach other people.

Sometimes, the side effects will make you think twice about collecting the data. There are some techniques later in this white paper to help you avoid trouble. In some rare cases, you might decide to avoid the metric altogether.

You may also want to consider demand characteristics, like the Hawthorne Effect[1], which found short-lived improvements can happen due to introducing measurement.

Types of measurement

There are many different ways to collect measurements. As software professionals, we tend to prefer instrumented system data, which is concrete and easy to get. Those responsible for running production systems will prefer early system indicators that help avoid system outages.

There are many cases where going outside your comfort zone will offer insights that machine-generated data can't. Let's look at:

- The different ways to categorize data
- Alternative mechanisms for measuring data outside your systems

1 https://en.wikipedia.org/wiki/Hawthorne_effect

Leading versus lagging indicators

Leading indicators help you predict future performance. You can use their early signals to predict changes in performance and take action to adjust the future outcome.

Lagging indicators tell you what happened in the past, but rather than being predictive, they're typically factual.

When you monitor a software system, leading indicators help you take action before a system becomes unavailable. For example, tracking CPU usage allows us to see when a feature uses too many resources and puts other features at risk. This allows you to scale your resources or disable the feature before the problem affects users.

If you waited for a lagging indicator, like screen loading times, your users would notice the problem before you do. That doesn't mean you shouldn't measure lagging indicators, as these show the *real* performance of the system. Leading indicators can alert you to a potential problem earlier, but they may change for other reasons than you expect.

Hopefully, this explains the importance of using both leading and lagging indicators to measure software delivery capability. The leading indicators help you react earlier, but the lagging indicators tell you the actual state of the whole value stream.

The most important measurements to your organization are likely to be lagging indicators. For example, your organization cares about *revenue*. This measurement tells you about money that has (or hasn't) come into the business. A predictive indicator, like net promoter score (NPS), can help you spot problems that might affect future revenue, but the money that turns up is a cold hard fact.

Leading indicators are often imperfect. We accept this because of the value of being able to take action sooner.

You should combine leading and lagging indicators to measure Continuous Delivery, DevOps, and software delivery performance.

Instrumented versus perceptual data

Instrumented data is a measurement you can collect automatically, like temperature. Perceptual data covers the human experience, like if the temperature is comfortable for you.

Instrumented data is popular because it's easy to get. However, perceptual data can capture complex relationships across many factors.

When you ask someone if a room temperature is comfortable, their answer goes beyond measurable temperature. The answer could include factors like:

- Humidity
- Outdoor conditions
- Their activity
- Clothing
- Personal preference

If you ask employees if they feel energized or burned out, you find they report feeling burned out before their productivity drops.

Most organizations are good at collecting instrumented data. You should become just as good at collecting perceptual data as it often provides uniquely nuanced insights.

System versus survey data

System data is information represented in your existing systems and databases. You can use system data to maintain a nearly real-time view of metrics.

Survey data needs a periodic collection of responses to questions, so the data is not as readily available.

When people use system data, there is a tendency to trust the numbers. You must be careful to ensure data is complete, comprehensive, and correct.

For example, if a team uses an internal tracking tool for features and a customer-facing tool to capture bugs, you must collect data from both systems. If you only collect metrics from the internal tracking system, this work will get prioritized and optimized at the cost of customer-reported issues.

You must also ensure that teams use tools as expected. In factories with time clocks, it was common for the first worker to clock their colleagues in to stop them from getting in trouble. According to the clock, worked hours would be far more than the actual hours, and any decisions based on this data were likely to be wrong.

In modern work tracking systems, users can mark work as complete before deploying it to production. This means inaccurate lead times.

Survey data needs more effort, but it can capture information not stored in a system. You can collect perceptual data from surveys and instrumented data that you can't get from a system. For example, if you haven't installed tools to support deployment automation, you could use a survey to ask how often a team deploys to production.

When designing your surveys, you should consider the frequency and effort needed to respond. It would help if you coordinated your efforts to avoid flooding people with surveys from different parts of the organization.

You may need some statistical skills to account for bias and help establish significant relationships in the data.

For both system and survey data, if the organization isn't a safe space for honest feedback, the data will be what people believe to be the 'correct answer'.

People can manipulate system data as easily as survey data. Marking work as complete before it becomes overdue and then tracking the work outside the system to finish it, for example.

You can't measure a complex system (like the software delivery process) on system data alone. You should use both system and survey data in your measurement approach. Use the natural conflicts to discover and fix weaknesses in your measurement approach.

Levels of measurement

There are four general levels of measurement, which can help you understand how a measure relates to the broader system. The different levels of measurement are in the table below, with an example based on measuring an electric heater.

Measurement level	Software example	Heating example
Activity	Lines of code	Power consumption
Outputs	Features per week	Heating element temperature
System output	Lead time	Room temperature
Outcomes	User value	People are comfortable

When an organization depends on activity metrics to measure productivity, the result is often a lot of motion without much progress. The number of lines of code in your software doesn't relate to your software delivery capability or your organization's performance.

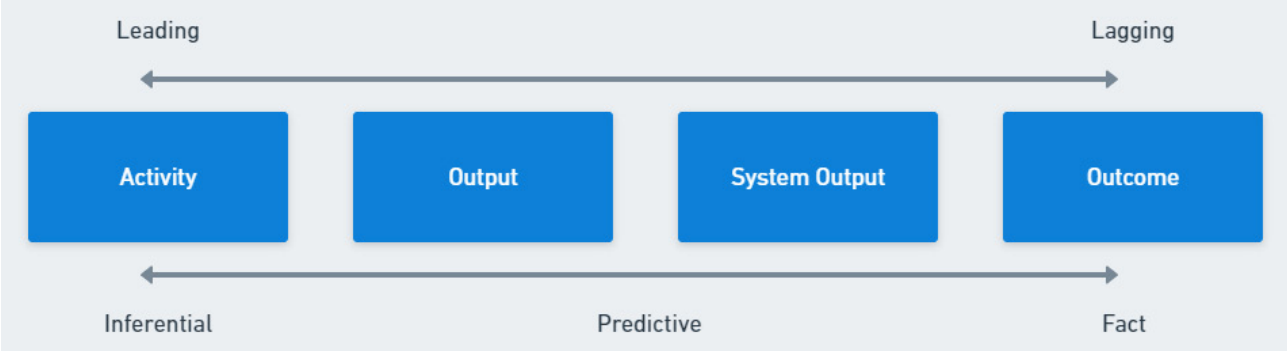
Activity metrics can be helpful as a part of your efforts to reduce the activity without negatively impacting the outcome. In our heating example, it would be desirable to reduce power consumption if you can do so without people being uncomfortable.

Output metrics are often available within the tools you already use to deliver software. Work tracking tools can tell you how many tickets a team closes each day, and version control can tell you how many commits or pull requests the team completes. There is a high risk of local optimization with these metrics if you use them in isolation.

Let's return to the heating example. If you select heaters based on the highest heating element temperature, manufacturers could get higher scores by removing the vent. While heating element temperatures would increase, rooms would be colder, and people less comfortable.

System output metrics aim to prevent local optimization by finding ways to assess the whole value stream continuously. DevOps focuses on system-level metrics that drive software delivery performance, like our thermostat in measuring room temperature.

Outcome metrics provide the most realistic measure of how successful we are. You should find ways to measure the whole system outcome to get a true reflection of software delivery and organizational performance. This type of feedback arrives later, which is why you need to use early indicators to improve your software delivery capability.



Early measurements allow you to respond faster, but outcome-based measurements are the truth

There is usually a trade-off between early availability and accuracy. You should balance your measurement strategy by including:

- Early indicators that let you react faster
- Later, outcome-based numbers that tell you whether you got the expected result

Avoiding unintended consequences

Whatever you decide to measure, you need to watch out for unintended consequences of the measurement.

> **Tell me how you will measure me, and then I will tell you how I will behave. If you measure me in an illogical way, don't complain about illogical behavior.** - Eli Goldratt (Critical Chain. 1997.)

Goldratt's suggestion isn't a bad place to start. If you tell people how you plan to measure them, they can understand the reasoning and help you achieve the goal, even if the metrics seem flawed. When you share your plan, teams can tell you what negative responses you may get, which can help you improve the design of your metrics.

There are 4 techniques that can help you refine your measurement strategy:

- Availability
- Aggregation
- Calculation
- Combination

You can use these to reduce the chance of local optimization or unfavorable impacts.

Availability

In some cases, keeping data private can prevent unintended alternate uses. This doesn't mean secretly tracking measurements, as this lacks transparency and will erode trust. Instead, you should allow teams to track their own data, giving them autonomy to act on it without publishing it to a broader audience.

The *velocity* metric in Scrum serves this purpose, allowing a team to plan without becoming a productivity measure. If you publish velocity too widely, someone might use it to compare performance across different teams or individuals.

When you decide to make data more available, you should also share the purpose of the measurement. This is the answer you gave to the question: 'How will I respond to this measurement?' You may need to adjust the description for your intended audience.

Aggregation

You can encourage the virtuous use of data with aggregation. With aggregation, you reduce attribution fidelity to prevent misuse.

For example, work tracking tools usually track items assigned to specific people. If you report on the work items per person, you make it difficult for an individual to pause their work to help someone else. An individual's performance may increase at the cost of the team and organization's performance.

As data moves up through your organization, you can limit this misuse by aggregating data at the appropriate level. This also increases the signal-to-noise ratio as information rises through the organization, as you manage fewer lines of data.

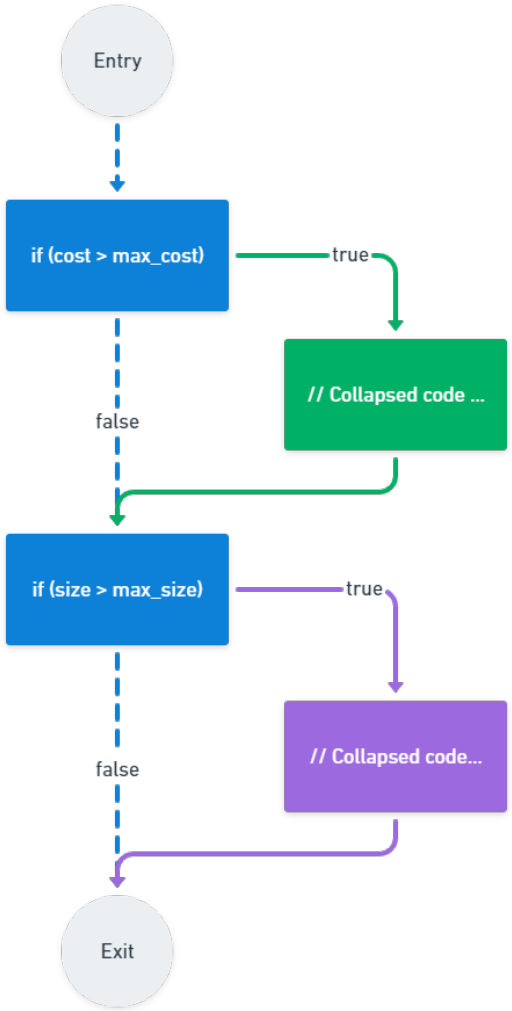
Calculation

Like aggregation, calculation removes some of the granularity in the data. Instead of supplying several raw metrics, you can combine them into a score.

Imagine your organization monitors the number of lines of code under maintenance in each team and the team size. By converting these into a calculated *maintenance load*, you avoid leaking the lines of code metric, which could get misused.

The *maintenance load* below converts the data into a metric you can use across teams. You must ensure you don't lose sight of any weaknesses inherent in the original data when you use them in a calculation. Lines of code might not be a good proxy for maintenance load, and team size doesn't necessarily provide a linear reduction in this load. This is where a perceptual survey can help determine whether the measurement is a reasonable reflection of reality.

Lines of code	Developers	Lines per developer	Maintenance load (÷ 10,000)
1,000,000	6	166,667	1.67
1,000,000	8	125,000	1.25
1,000,000	10	100,000	1.00
1,000,000	12	83,333	0.83
1,000,000	14	71,429	0.71



Two if-statements create 3 paths through a function, so the cyclomatic complexity is 3

A more robust example of calculated metrics is *Cyclomatic Complexity*, created by Thomas McCabe in 1976. It creates a graph of code statements, calculates the number of paths through the code, and provides a score per function, file, or system to indicate its complexity. This same technique predicts the minimum test number needed to exercise all lines of code.

Combination

You can publish several metrics together in combination to create a balancing effect. Imagine a market trader who wakes up early each morning and buys fresh strawberries to sell. You might track the daily spending on strawberries, but you must balance this by measuring sales. The more expensive strawberries might be more popular or sell with a higher margin. If you only tracked the cost, it would result in purchasing the cheapest strawberries, which would damage your business.

By plotting metrics in combination, you can discover if there's a relationship between them and avoid focusing on a single metric to the exclusion of all others.

The DORA metrics and SPACE framework balance competing software delivery demands using metric combination.

Other measurement considerations

The scientific method emerged in the 17th century, and we haven't yet found a better way to learn. The core element of the technique is to observe something, form a falsifiable theory, and then test it with an experiment.

When you spot something interesting in data, it can be tempting to invent a narrative to explain it. You must complete the process by capturing your insight as a hypothesis and running an experiment to test it. The software industry has been moving towards a scientific approach for several decades. We're now in an era where research-backed insights allow our industry to make fantastic progress. See our white paper on The importance of Continuous Delivery[2] for more.

If you use the scientific method, you can find metrics that predict success for your team and organization.

Metrics with low frequency can be dangerous, so try to increase the measurement frequency wherever possible. Imagine driving a car with a speedometer that only updates once every 30 minutes; you'd spend most of your time driving at the wrong speed.

Data with high variability can tempt you to smooth charts using averages or medians, but you miss out on what the outliers tell you. When reporting on an average, you should also use a technique to highlight distribution and outliers, as these often provide the most interesting insights.

Avoid selecting visually pleasing charts when you create dashboards or charts to track your progress. Sticking with reasonably boring line charts, scatter plots, and box-and-whisker charts will provide a richer, more actionable dashboard than pie charts, gauges, and radar charts. You can typically divide charts by geometry, and rectangles outperform circles.

2 https://octopus.com/whitepapers/lv-the-importance-of-continuous-delivery

Metric design summary

As you implement Continuous Delivery and DevOps measurements, apply metric design techniques to ensure they promote a healthy and productive environment for software delivery. The goal of high-performance in software delivery is to help the organization achieve its goals, so all improvement efforts should consider the complete value stream.

If you create more metrics to drive improvement in software delivery, use the purpose questions to assess the chance of misuse. Use the techniques for availability, aggregation, calculation, and combination to limit unintended consequences.

When you visualize your metrics, stick to chart types that focus on the data rather than selecting visually pleasing ones. The data should be the star of your dashboard rather than your exciting display choices.

Statements of Continuous Delivery

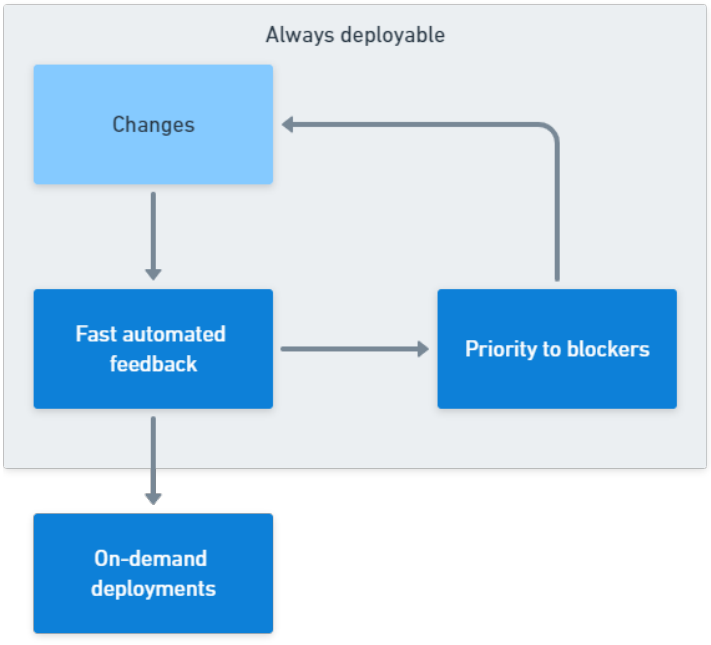
ThoughtWorks originally designed the statements of Continuous Delivery. The statements help to focus efforts on the key goals of Continuous Delivery and complement the metric-based measures described later.

You can use the statements to review your Continuous Delivery performance and find improvement areas.

1. Software is always deployable
2. You prioritize work to keep software deployable
3. You have fast, automated feedback for every change
4. On-demand automated deployments for any version and environment

Always deployable

To keep your software deployable at all times, you need to detect functional bugs early in your deployment pipeline. You must also continuously maintain the system's quality attributes, like security and performance.



The relationship between Continuous Delivery statements

When working on a new feature, you should use feature toggles or keystoning techniques to allow deployments to continue without unfinished features being available to users or API consumers. You also need database refactoring strategies to de-couple database and application deployments.

'Keystoning' is a technique inspired by the wedge-shaped architectural stone at the top of an arch. It's the last stone put in place, allowing the arch to bear weight.

In software, keystoning involves putting a feature in place without exposing it to users. When the feature's ready, you add the keystone (like the user-interface update) to make the feature visible to users.

Feature toggles are an even better way to control feature visibility. Unlike keystoning, you can change feature toggles independent of deployments. You can use feature toggles to enable a feature for everyone, a group of users, or disable it if you detect a critical issue.

Prioritize blockers

When an issue stops software from being deployable, you should prioritize it over feature development. New features are of little use if you can't deploy them. Plus, you can't fix critical problems you discover in production until you can deploy the software.

Blockers may include:

- Bugs that would prevent a deployment
- An unexpected dependency between components
- A problem with your deployment pipeline

Fast, automated feedback

Each time the software changes, fast, automated feedback should be available to the whole team. This lets you respond quickly to any issues preventing deployments. When you discover a problem, part of the fix should be to introduce a test to detect the issue at an earlier stage of your deployment pipeline, if possible.

If you can detect problems early, you can quickly fix the issue to unblock the deployment pipeline. Without early detection, you must manage the impact across lots of other work, causing a retrospective batch of interdependent changes.

On-demand deployments

You need to be able to deploy any version of the software to any environment easily. This is often called a *push-button deployment*, or an *on-demand deployment*.

You can deploy on-demand if your deployments are fast, automated, repeatable, and reliable. Many high-performing organizations deploy infrastructure with the same level of automation as their application deployments.

Continuous Delivery statements summary

You can assess your Continuous Delivery adoption using 4 statements. Each statement is either true or not. Your answers will give you a strong idea of where to focus improvement efforts.

1. Software is always deployable
2. You prioritize work to keep software deployable
3. You have fast, automated feedback for every change
4. You can perform push-button deployments for any software version and environment

If you want a more concrete way to measure DevOps and Continuous Delivery, read on to find out about the DORA metrics.

DORA metrics for DevOps delivery

DORA (DevOps Research and Assessment) are the team behind the Accelerate State of DevOps Report, a survey of over 32,000 professionals worldwide. Their research has linked the technical and cultural capabilities that drive software delivery and organizational performance.

DORA recommends an approach to measure software delivery that relies on 5 metrics:

- Throughput (measures the health of your deployment pipeline)
  1. Deployment frequency (DF)
  2. Lead time for changes (LT)
- Stability (helps you understand your software quality)
  3. Change failure rate (CFR)
  4. Mean time to recovery (MTTR)
- Operational (measures operation performance)
  5. Reliability

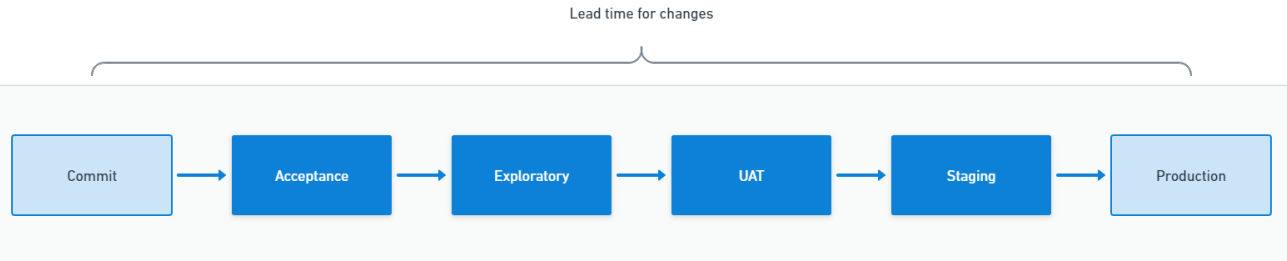
We explain all 5 metrics below.

DevOps insights (early access) in Octopus gives you better visibility into your DevOps performance by surfacing deployment throughput and stability metrics. These metrics help you qualify the results of your DevOps performance, as well as gain insights into areas for future improvement. Learn more in our Octopus DevOps Insights documentation[3]

3 https://octopus.com/docs/insights

Lead time for changes

You may have seen several definitions for 'lead times' in software delivery and manufacturing, so it's worth being specific about the DevOps' definition. Lead time for changes refers to the time it takes for a code change to reach the live environment. We measure this from code commit to the production deployment.



Lead time for changes spans the deployment pipeline

If you push build metadata into your deployment automation tool, you can calculate the lead time for changes by finding the oldest commit in the deployment.

If you don't push metadata from the build system into the deployment automation tool, you can use the time from package upload to deployment. You also need to keep track of the build time separately. You shouldn't let build time be a blind spot as it's often affected by automated test duration and increases over time if not measured.

High performers have lead times of less than one week, and elite performers have lead times under an hour.

Deployment frequency

Deployment frequency measures how often you deploy to production or to end users. You can measure this with your deployment automation tool, which sees the deployment rate to the production environment.

Change failure rate

Your change failure rate is the percentage of changes that result in a fault, incident, or rollback. To track change failure rates, you need to keep a log of all individual changes that result in a production issue.

Your work tracking tools may have a feature to link a bug request to the original change. Otherwise, you may be able to add a custom field to retrospectively mark a change as "failed" to use in reporting.

Mean time to recovery

Your mean time to recovery is the average time between a failure and full recovery, whether due to a code change or something else. You can collect this from your work tracking tools by marking work items as a production fix and measuring the time it takes to complete the work.

When you need a code change to resolve a fault, your lead time will factor in the recovery time. A short lead time can be helpful as it allows you to deploy fixes without a special process to fast-track the change.

Traditionally, you measure operations on availability, which assumes you can prevent all failures. In DevOps, we accept there will always be failures outside our control, so the ability to spot an issue early and recover quickly is valuable.

DORA metric performance levels

Based on survey responses, DORA grouped organizations into performance levels. Organizations in the elite performance group not only had better software delivery capability but also saw greater organizational success.

Performance level	Lead time	Deployment frequency	Change failure rate	Mean time to resolve
Elite	< 1 hour	Multiple times per day	0-15%	< 1 hour
High	1 day - 1 week	Weekly to monthly	16-30%	< 1 day
Medium	1-6 months	Monthly to biannually	16-30%	1 day - 1 week
Low	> 6 months	Fewer than once every 6 months	16-30%	> 6 months

Operational performance

The DORA metrics focus on software delivery performance. However, the 2021 State of DevOps Report found the operational capability of *reliability* drives benefits across many outcomes. Reliability refers to teams prioritizing meeting or exceeding their reliability targets.

You find the quality of internal documentation will be a key to high performance in reliability. Teams with high-quality documentation were more than twice as likely to meet or exceed their reliability targets. Documentation also improved performance against the other DORA metrics. You should measure reliability against the service level objectives of your software.

When you exceed your service level objectives by too much or for too long, other systems using your service will start to depend on the high availability you achieved. Rather than anticipating downtime and handling it gracefully, many may assume your service will always be available and cause problems when there's an outage.

You can use short and deliberate outages to bring availability closer to the service level objective and test system resilience. This will help ensure other systems handle outages better.

DORA metric summary

The DORA metrics use system-level outcomes to measure software delivery and operational performance. How an organization performs against these measures predicts its performance against its goals. Elite performers outpace competitors in their industry.

By removing obstacles to the fast flow of changes to production, you can:

- Deliver value to customers
- Experiment with features
- Get feedback quickly.

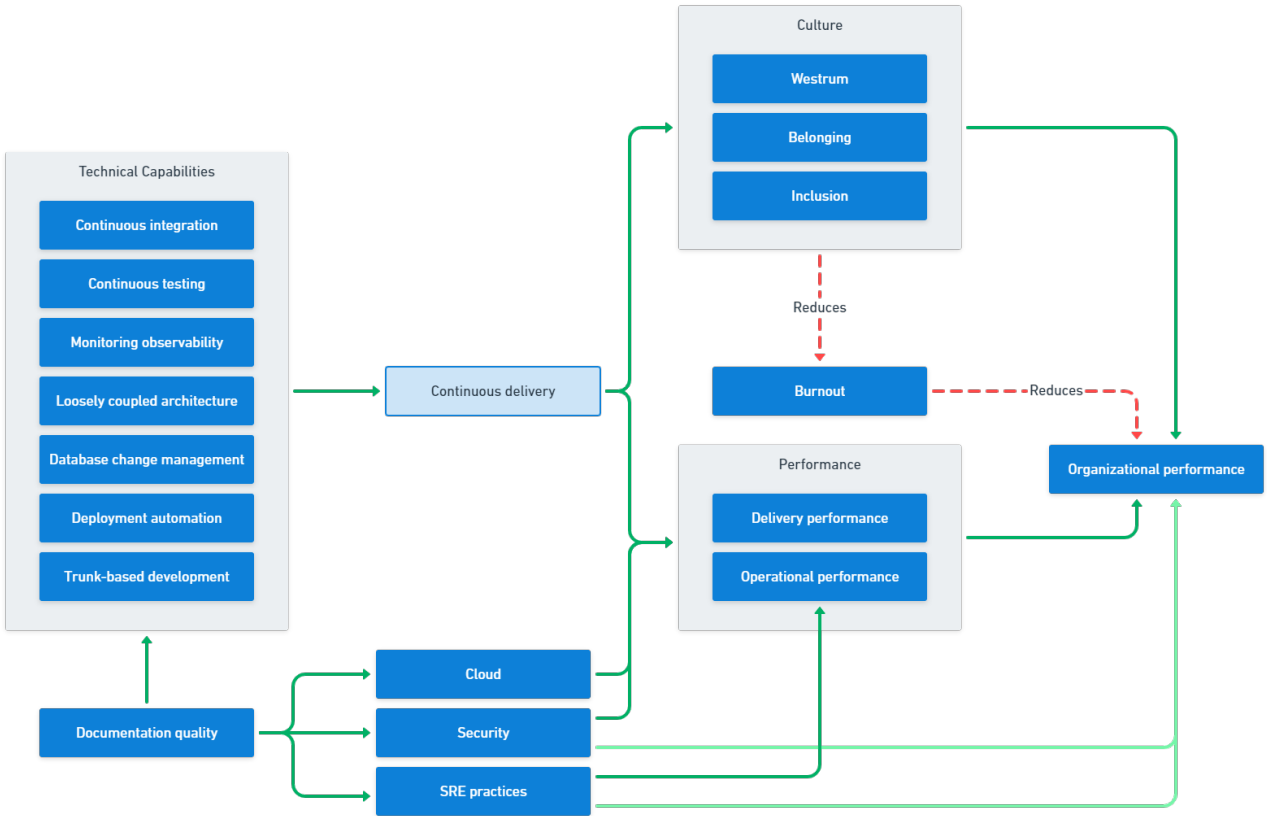
With the DORA software delivery and operations metrics in place, you can experiment with changes to your deployment pipeline and answer the questions:

- Does this make us deliver software faster or more frequently?
- Does this make our software more stable?

Often, improvements in software delivery performance result in increased speed and stability. This is one of the key findings of the State of DevOps report. Teams who deliver software faster also write better quality software. The elite performers can:

- Change their applications faster
- Deploy them more often
- Have fewer failures
- Recover quickly from faults.

It's rare to find a trade-off between speed and stability, even though this is counter-intuitive. If you find a trade-off emerging, you can use the Continuous Delivery statements and DevOps capabilities list in DORA's structural equation model to check if you're missing a critical practice.



The DORA structural equation model

Continuous Delivery helps you achieve high performance. The relationship between speed and stability will help amplify your improvements.

The SPACE framework

The SPACE framework[4] focuses on the deep relationship between wellbeing, satisfaction, and productivity. It uses a multi-dimensional measurement system that helps organizations understand how people and teams work.

You can use the SPACE framework to find a balance between individual, team, and organization optimizations. There is a natural tension captured in the framework.

For example, individual productivity reduces when someone helps the broader team. That means optimizing for personal productivity would damage your team's productivity. Equally, subverting individuals to the team's service would mean they get none of their work done.

You should use a mix of measures from the 5 dimensions, mixing instrumented and perceptual measurements. Perceptual data is the only way to determine how people feel, which is a crucial driver of productivity.

The SPACE framework includes 5 dimensions:

- Satisfaction and wellbeing
- Performance
- Activity
- Communication and collaboration
- Efficiency and flow

We explain the dimensions below and some ideas for data you could capture. The goal isn't to put in place all these metrics but to choose measurements based on your circumstances.

Satisfaction and wellbeing

Satisfaction refers to how fulfilled people feel with their work, team, tools, and culture. Wellbeing captures how healthy and happy they are.

This dimension often predicts future performance. People rate their satisfaction and wellbeing lower *before* their productivity falls.

4 https://queue.acm.org/detail.cfm?id=3454124

System data

- Retention - how many people stay on a team and within the organization

Survey questions

- How satisfied are you with other employees?
- Would you recommend your team to other people?
- Do you have the tools you need to get your job done?
- How would you rate your energy and enthusiasm for work?
- How satisfied are you with the software delivery system?

Performance

The performance dimension captures the outcome of a system or process.

System data

- Quality - how many defects you have and the ongoing system health and reliability
- Impact - feature usage, cost reduction, and business won and kept

Survey questions

- Customer satisfaction with a feature, the product, and your organization

Activity

Activity metrics track the work done, like how many bugs you've fixed. You should never use activity metrics in isolation. Increased activity does not necessarily mean improved outcomes.

System data

- Number of work items, commits, pull requests, builds, and deployments
- Number of incidents and issues and their severity

Communication and collaboration

Team communication and collaboration often come at a cost to individual productivity. Yet, collaboration drives higher team performance.

System data

- Speed of integration (for example, if the team uses pull requests, how long does it take requests to get reviewed and merged)
- Presence and quality of documentation
- Onboarding speed (how long before a new developer has code in production)

Survey questions

- How would you rate the quality of team meetings?

Efficiency and flow

Individuals need to get solid chunks of focus time to get into a flow state and stay productive. This dimension refers to the flow of work and information.

System data

- Number of hand-offs in a process
- Whole value-stream cycle time
- Amount of value-adding time in the value stream
- Amount of wait time in the value stream

Survey questions

- Can you regularly get uninterrupted flow time?
- How often are you interrupted?
- How much time do you spend on interruptions?

SPACE framework summary

The SPACE framework doesn't need every metric measured all the time. You should select a mix of instrumented and perceptual measures across at least 3 dimensions. Over time, you can adjust by adding and removing metrics to nudge behavior and communicate what the organization values. You can use the metric selection principles to reduce unintended side effects.

Keep individual data private and only share aggregated values at the team and organization levels. Don't feel tempted to ask survey questions that compromise anonymity, as this will limit how honestly people can answer. Even asking for a job title or department can discourage openness, especially for people in unusual roles or small teams.

Watch for external forces that might influence your data. Activity metrics often drop when a team or organization invests in the future by attending training and conferences or when people take holidays.

You should also pay attention to bias in your system, which you should attempt to uncover in the data. Look out for skewed code reviews or longer waits for pull requests for specific team members, as these signal biases you need to tackle. You can use survey responses to check for this, too.

Personality differences can impact survey responses. Asking individuals to compare their feelings to a previous period can help reduce individual measurement bias. For example, here are 2 ways to ask for the same information. The second question asks the individual to compare to their previous experience:

1. How many interruptions do you get? Not many, a few, or a lot.
2. Compared to last week, how many interruptions did you get? Fewer, about the same, or more.

Make sure you don't discourage vital but invisible work. Focusing too much on individual productivity or instrumented data will discourage teamwork, like showing someone new how to make a change in a complex system. If you set individual focus on personal productivity, like the number of commits, essential teamwork will not happen.

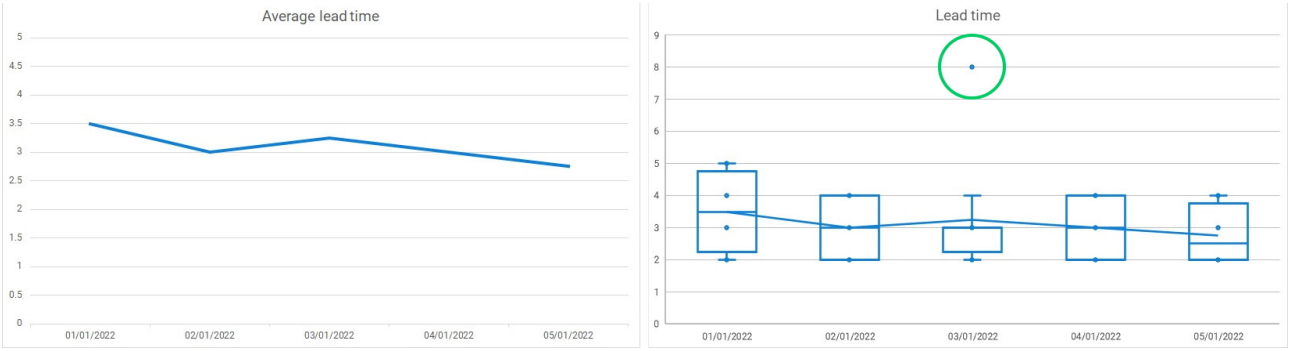
The SPACE framework attempts to balance the natural tension between individual, team, and organization performance. Other productivity measurements that hide these tensions result in sub-optimal performance.

Measuring Continuous Delivery summary

Several pre-baked measurement sets help you start a more scientific approach to your work. There are also reminders to look beyond software and measure the performance of the whole organization.

You need to find a place to put all your metrics and measurements. The goal should be to have them available on a *single pane of glass*, so you can look at the relationships between the numbers. A charting tool can visualize outliers, and anomalies can help avoid blind spots.

Your chart selection should favor surfacing information and highlighting trends and outliers. For example, a line chart showing average lead times might hide occasional long waits. A scatter plot or box-and-whisker chart will make the outliers more obvious. The exceptions often offer the best insights.



A line chart of average numbers hides an interesting outlier

Wherever you store the data, you should ensure it's cheap and easy to change what you measure regularly.

Where to start

If you have a small number of systems, implementing the DORA metrics is an excellent start. If this needs collection from many data sources, starting survey data collection as part of the SPACE framework may be a faster way to start.

While you build your data capability, you can start collecting survey data, which you replace once systems are in place. For example, until you can get data from your deployment tools, you could ask individuals: "How often does your team deploy to production?"

You can also use periodic surveys to detect issues with your automated data collection. If your system lead times are shorter than people report in a survey, you may have a problem with your system data or how people use the tools.

The DORA metrics are an easy way to measure your deployment pipeline and its impact on software delivery performance. Once you master collecting and responding to data, look at extending metrics to include broader organizational outcomes.

Continuous improvement

All the measurements described in this white paper can help reveal where you can make improvements. They also provide a mechanism to design your improvements deliberately and scientifically. If you make a change to increase deployment rates but instead deploy less, you know the change didn't have the intended effect and can try something else.

Over time, you need to respond to weaker signals rather than believing there are no further improvements to make. Teams and organizations who continuously improve will outperform those that think they're done.

As you increase your expertise in software delivery, you need to use new techniques to generate learning. For example, you may need to inject deliberate failures to see how the whole system (not just the software) responds to the fault. This can provide new opportunities to improve.

You should have conversations to find the reasons for improvements and declines in performance. Both give key insights you can use to create new theories to test.

Build habits

You can use short-term measurements to build habits. For example, timing branch lifespans helps highlight branches kept too long and encourages commits to the main line more often.

You can demote metrics once you solve problems, leaving an automated alert to warn you if things slip back.

Follow the constraint

An assumption in most literature is that software delivery is the constraining factor for organizations. This may be true in most cases, but if software delivery isn't the constraint, optimizing software delivery won't make a meaningful impact on your organization.

In these rare cases, you need to find the actual constraint and elevate it, even where this lowers the software team's performance.

In cases where software delivery *is* the constraint, the measurements in this white paper can help make a meaningful difference to software delivery performance and help organizations reach their goals.

Further reading

- Continuous Delivery. 2011. Humble, Farley.
- Accelerate. Forsgren, Humble, Kim. 2018.
- Project to Product. 2018. Kersten.
- The DevOps Handbook (Second Edition). 2021. Kim, Humble, Debois, Willis.
- Modern Software Engineering. 2022. Farley.

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.