New Relic.

DevOps Done Right

Best Practices to Knock Down Barriers to Success

Table of Contents

Introduction

Your team has embraced DevOps. You're establishing new processes, adopting new tools, and forming a culture that emphasizes cross-functional collaboration. But you haven't yet reached maximum velocity. There's something missing, something that's keeping your organization from truly becoming a high-performing DevOps machine.

Often that missing piece is measurement of data. Although measurement is one of the five pillars of the CALMS framework (Culture, Automation, Lean, Measurement, Sharing) coined by DevOps expert Jez Humble, it's frequently neglected by DevOps teams in their push for increased velocity and autonomy. However, this can create huge problems, as accurate data is critical to the successful functioning of a DevOps team—from effective incident response to navigating microservices complexity and more.

This ebook is for all of the teams and organizations that have been dipping their toes in the water and are now ready to take the plunge into all things DevOps. It's also aimed at those who are treading water without making the DevOps progress they need to achieve a true digital transformation.

By sharing real-world experiences—particularly lessons we've learned here at New Relic—we want to help you knock down your remaining barriers to DevOps success. From understanding how to set reliability goals to untangling the unique communications and development requirements of your microservices approach, we're bringing together proven best practices that show you how to move faster and more effectively than ever before.

THEN	NOW
Monolith Ruby	200+ microservices
Siloed teams	50+ engineering teams with embedded site reliability engineers
Infrequent releases	Up to 70 deploys per day
Reactive response	Proactive monitoring and response

New Relic then and now: The Journey to DevOps

CHAPTER 1

Balancing SLO With Fast Application Delivery

Balancing SLO With Fast Application Delivery

Your development cycles are faster and you're deploying code more frequently, but how's your reliability? Quality and reliability are equally important outcomes of a successful DevOps approach.

That's where SRE comes in. Site reliability engineering (SRE) is a cross-functional role, assuming responsibilities traditionally dedicated and segregated within development, operations, and other IT groups. Because SRE relies on both dev and ops collaboration, it goes hand-in-hand with the DevOps culture. And while DevOps and SRE have much in common, SRE elevates the focus on continuous improvement and managing to measurable outcomes, particularly through the use of service level objectives (SLOs).

Let's start with some important definitions:

TERM	DEFINITION	EXAMPLE
Service level indicator (SLI)	The SLI is your core measurement of performance.	"Customers can log in and view their data ..."
Service level objective (SLO)	SLOs are the target values or goals for the performance of your system. SLOs represent an ongoing commitment.	"99.9% of the time ..."
Service level agreement (SLA)	The SLA defines what happens if you don't meet your SLI/SLO commitments.	"... or they can request a refund for losses incurred due to unavailability of the service."

Learn more about SRE in our ebook, Site Reliability Engineering: Philosophies, Habits, and Tools for SRE Success

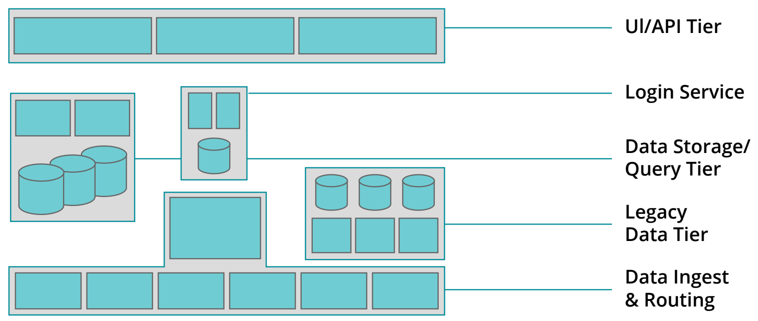
"Fundamentally, [SRE is] what happens when you ask a software engineer to design an operations function."

Ben Treynor Sloss, Vice President of Engineering, Google

Setting appropriate SLIs and SLOs

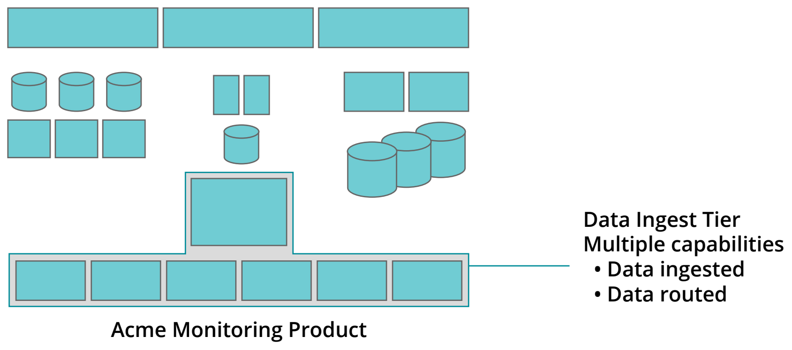
While industry best practices for SRE call for setting SLIs and SLOs for each service that you provide, it can be quite challenging to define and deploy them if you haven't done this before. Here are seven steps that we use at New Relic to set SLOs and SLIs:

1. **Identify system boundaries:** A system boundary is where one or more components expose one or more capabilities to external customers. While internally your platform may have many moving parts—service nodes, database, load balancer, and so on—the individual pieces aren't considered system boundaries because they aren't directly exposing a capability to customers. Instead multiple pieces work together as a whole to expose capabilities. For example, a login service which exposes an API with the capability of authenticating user credentials is a logical group of components that work together as a system. Before you set your SLIs, start by grouping elements of your platform into systems and defining their boundaries. This is where you'll focus your effort in the remaining steps because boundary SLIs and SLOs are the most useful.



Systems and boundaries within a platform

2. **Define capabilities exposed by each system:** Now group the components of the platform into logical units (e.g., UI/API tier, login service, data storage/query tier, legacy data tier, data ingest & routing). Here at New Relic, our system boundaries line up with our engineering team boundaries. Using these groupings, articulate the set of capabilities that are exposed at each system boundary.



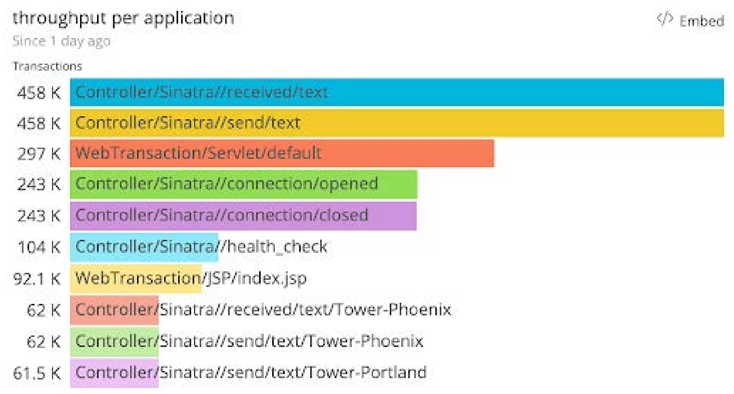
Capabilities defined at a system boundary

3. **Create a clear definition of "available" for each capability:** For example, "delivery of messages to the correct destination" is a way to describe expectations of availability for a data-routing capability. Using plain English to describe what is expected for availability—versus using technical terms that not everyone is familiar with—helps avoid misunderstandings.

4. **Define corresponding technical SLIs:** Now it's time to define one or more SLIs per capability using your definition of availability of each capability. Building on our example above, an SLI for a data-routing capability could be "time to deliver message to correct destination."

5. **Measure to get a baseline:** Obviously, monitoring is how you'll know whether you are achieving your availability goals or not. Using your monitoring tool, gather baseline data for each SLI before you actually set your SLOs.

6. **Apply SLO targets (per SLI/capability):** Once you have the data, but before you set your SLOs, ask your customers questions that help you identify what their expectations are and how you can align your SLOs to meet them. Then choose SLO targets based on your baselines, customer input, what your team can commit to supporting, and what's feasible based on your current technical reality. Following our SLI example for data routing, the SLO could be "99.5% of messages delivered in less than 5 seconds." Don't forget to configure an alert trigger in your monitoring application with a warning threshold for the SLOs you define.



SLI Example for Data Routing Capability

7. **Iterate and tune:** Don't take a set-it-and-forget-it approach to SLOs and SLIs. You should assume that they will, and should, evolve over time as your services and customer needs change.

Additional tips for SLOs and SLIs

- **Make sure each logical instance of a system has its own SLO:** For instance, for hard-sharded (versus horizontally scaled) systems, measure SLIs and SLOs separately for each shard.
- **Know that SLIs are not the same as alerts:** The SRE process is not a replacement for thorough alerting.
- **Use compound SLOs where appropriate:** You can express a single, compound SLO to capture multiple SLI conditions and make it easier for customers to understand.
- **Create customer-specific SLOs as needed:** It's not unusual for major customers to receive SLAs that give better availability of services than those provided to other customers.

"To achieve operational excellence, we measure everything. Only in that way can we manage and improve everything."

Craig Vandeputte, Director of DevOps, CarRentals.com

CHAPTER 2

Creating a Fair and Effective On-Call Policy

Creating a Fair and Effective On-Call Policy

The next step in improving reliability while accelerating deployments is to make sure that your organization can handle any software issues that arise—anytime day or night—quickly and effectively. For this, you need an on-call policy.

Wait ... don't skip to the next chapter yet. We know the "on call" term can evoke many emotional responses in people. But that's primarily because many organizations get the concept of on-call rotation wrong. And getting it wrong means not only the stress and negative attention of missing your SLAs with customers, but working in an unproductive, unpleasant culture with a team of exhausted and frustrated engineers.

Start with the fundamentals

An effective and fair on-call policy starts with two important prerequisites:

1. **Structured system and organization:** Responding to issues effectively is far easier when both your systems (services or applications) and your product teams are well organized and structured into logical units. For instance, at New Relic our 57 engineering teams support 200 individual services, with each team acting autonomously to own at least three services throughout the product lifecycle, from design to deployment to maintenance.
2. **A culture of accountability:** With DevOps, each team is accountable for the code that it deploys into production. Teams naturally make different decisions about changes and deployments when they are responsible and on call for the service versus traditional environments where someone else is responsible for supporting code once it's running in production.

Apply these best practices to improve your on-call practice

Structure your team and organization fairly

Here at New Relic, every engineer and engineering manager in the product organization rotates on-call responsibilities for the team's services. Teams are responsible for at least three services, but the number of services supported depends on the complexity of the services and the size of the team. For your organization, look at the size of the total engineering organization and of individual teams before choosing an on-call rotation approach. For instance, if the team has six engineers, then each engineer could be the primary person on call every six weeks.

Be flexible and creative when designing rotations

Consider letting each team design and implement its own on-call rotation policy. Give teams the freedom and autonomy to think out-of-the-box about ways to organize rotations that best suit their individual needs. At New Relic, each team has the autonomy to create and implement its own on-call system. For instance, one team uses a script that randomly rotates the on-call order of the non-primary person.

Track metrics and monitor incidents

An important part of making the on-call rotation fair and effective is monitoring and tracking incident metrics. Here at New Relic, we track number of pages, number of hours paged, and number of off-hours pages. We look at these metrics at the engineer, team, and group levels. Tracking metrics helps draw attention to teams that are faced with unmanageable call loads (if a team averages more than one off-hours page per week, that team is considered to have a high on-call burden). Staying on top of these metrics lets us shift priorities to paying down a team's technical debt or providing more support to improve the services.

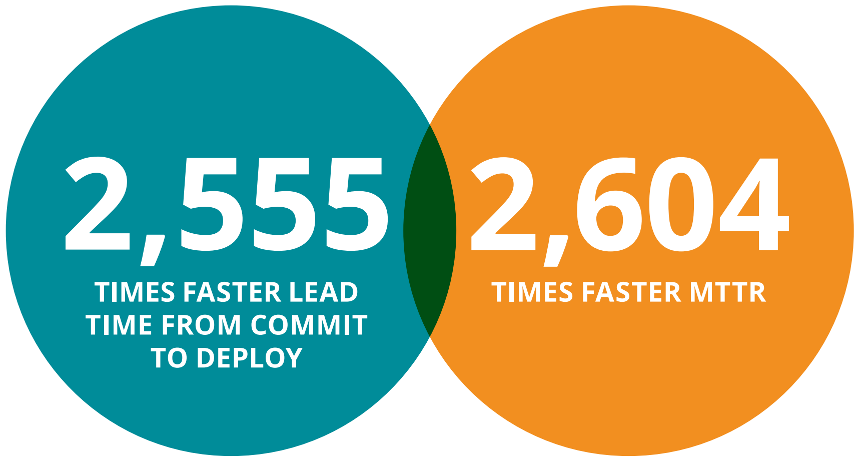
"[DevOps] high performers were twice as likely to exceed their own goals for profitability, market share, and productivity." [1]

Adapt your policy to align with your company's situation

An on-call policy that works for a team at New Relic might be completely unsustainable for your company. To create an on-call rotation that is both fair and effective, consider additional inputs such as:

- **Growth:** How fast is your company and your engineering group growing? How much turnover are you experiencing?
- **Geography:** Is your engineering organization centralized or geographically distributed? Do you have the resources to deploy "follow-the-sun" rotations?
- **Complexity:** How complex are your applications and how are they structured? How complex are dependencies across services?
- **Tooling:** Do you have incident response tools that give engineers automatic, actionable problem notification?
- **Culture:** Have you made being on call an essential part of the job in your engineering culture? Do you have a blameless culture that is focused on finding and solving the root cause instead of seeking to lay blame?

COMPARING THE ELITE GROUP AGAINST THE LOW PERFORMERS, WE FIND THAT ELITE PERFORMERS HAVE...



Source, "Accelerate: State of DevOps 2018: Strategies for a New Economy," DORA

1: Source, "2017 State of DevOps Report," Puppet and DORA

CHAPTER 3

Responding to Incidents Effectively

Responding to Incidents Effectively

Concomitant to on-call rotations is the concept of incident management. What's an incident? That's when a system behaves in an unexpected way that might negatively impact customers (or partners or employees).

A core competency within the "you build it, you own it" DevOps approach, incident management is often given short shrift, with teams losing interest once an issue is resolved. Often organizations without effective incident management take on "firefighting" responsibilities using ad-hoc organization, methods, and communications. When something blows up, everyone scrambles to work out a plan to solve the problem.

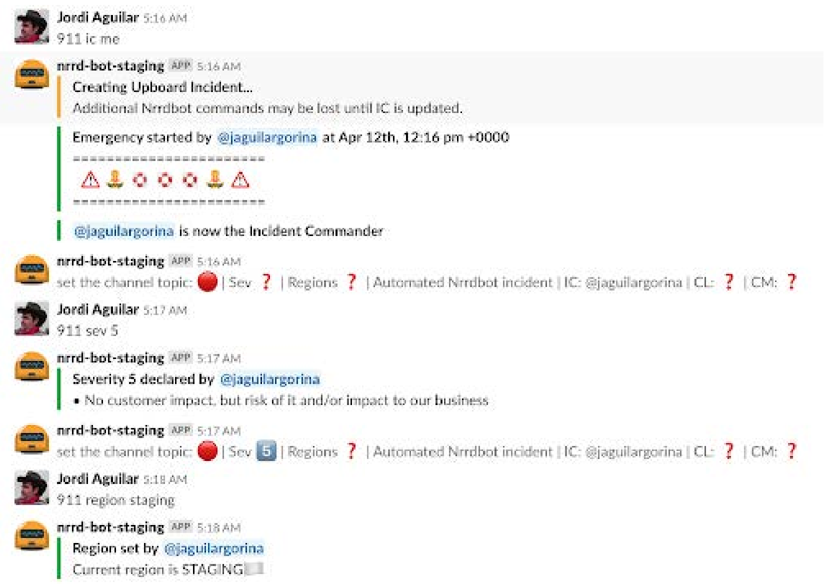
There's a much better way to approach incidents, one that not only minimizes the duration and frequency of outages, but also gives responsible engineers the support they need to respond efficiently and effectively

Creating an effective incident management process

1. **Define severities:** Severities determine how much support will be needed and the potential impact on customers. For example, at New Relic we use a scale of 1 to 5 for severities:
    - Level 5 does not impact customers and may be used to raise awareness about an issue.
    - Level 4 involves minor bugs or minor data lags that affect, but don't hinder, customers.
    - Level 3 is for major data lags or unavailable features.
    - Levels 2 and 1 are serious incidents that cause outages.
2. **Instrument your services:** Every service should have monitoring and alerting for proactive incident reporting. The goal is to discover incidents before customers do to avoid worst-case scenarios where irritated customers are calling support or posting comments on social media. With proactive incident reporting, you can respond to and resolve incidents as quickly as possible.
3. **Define responder roles:** At New Relic, team members from engineering and support fill the following roles during an incident: incident commander (drives resolutions), tech lead (diagnoses and fixes), communications lead (keeps everyone informed), communications manager (coordinates emergency communication strategy), incident liaison (interacts with support and the business for severity 1s), emergency commander (optional for severity 1s), and engineering manager (manages the post-incident process).
4. **Create a game plan:** This is the series of tasks by role that covers everything that happens throughout the lifecycle of an incident, including declaring an incident, setting the severity, determining the appropriate tech leads to contact, debugging and fixing the issue, managing the flow of communications, handing off responsibilities, ending the incident, and conducting a retrospective.
5. **Implement appropriate tools and automation to support the entire process:** From monitoring and alerts, to dashboards and incident tracking, automating the process is critical to keeping the appropriate team members informed and on task, and executing the game plan efficiently.

New Relic.

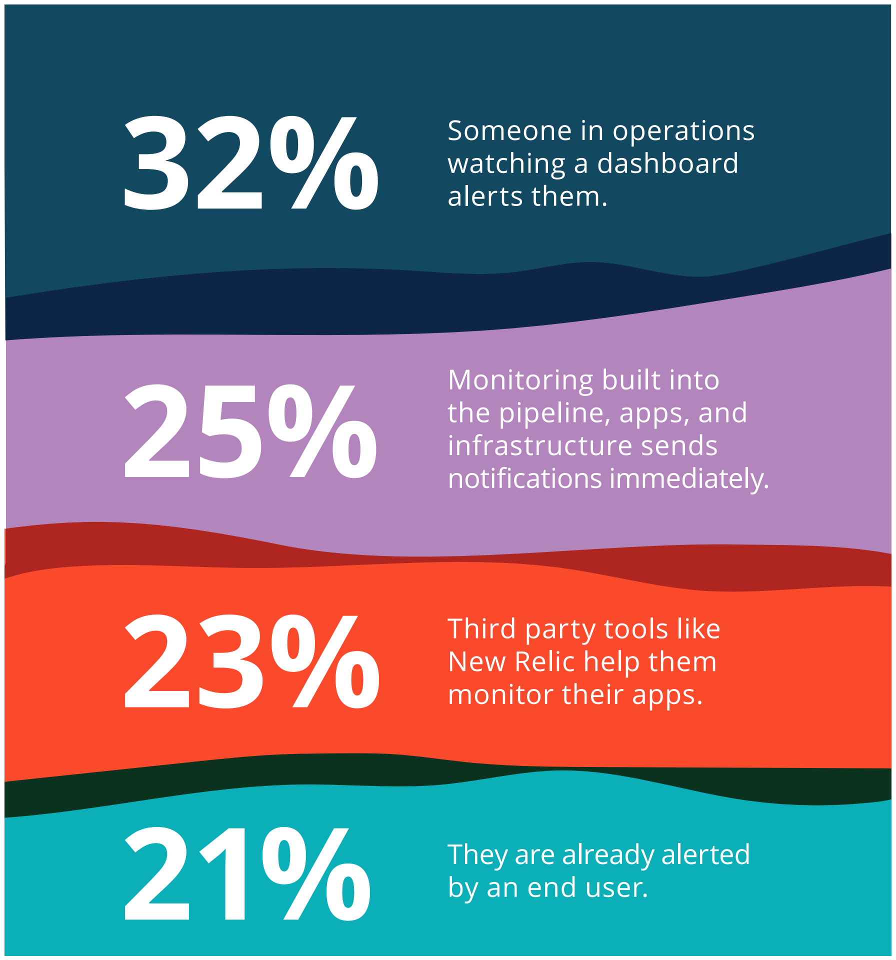
6. **Conduct retrospectives:** After the incident, require your teams to conduct a retrospective within one or two days of the incident. Emphasize that the retrospective is blameless and should focus instead on uncovering the true root causes of a problem.
7. **Implement a Don't Repeat Incidents (DRI) policy:** If a service issue impacts your customers, then it's time to identify and pay down technical debt. A DRI policy says that your team stops any new work on that service until the root cause of the issue has been fixed or mitigated.



```
Jordi Aguilar 5:16 AM
911 ic me
nrrd-bot-staging APP 5:16 AM
  Creating Upboard Incident...
  Additional Nrrdbot commands may be lost until IC is updated.
  Emergency started by @jaguilargorina at Apr 12th, 12:16 pm +0000
  ========================
  ⚠ ⚓ ⭘ ⭘ ⭘ ⚓ ⚠
  ========================
  @jaguilargorina is now the Incident Commander
nrrd-bot-staging APP 5:16 AM
set the channel topic: 🔴 | Sev ❓ | Regions ❓ | Automated Nrrdbot incident | IC: @jaguilargorina | CL: ❓ | CM: ❓
Jordi Aguilar 5:17 AM
911 sev 5
nrrd-bot-staging APP 5:17 AM
  Severity 5 declared by @jaguilargorina
  • No customer impact, but risk of it and/or impact to our business
nrrd-bot-staging APP 5:17 AM
set the channel topic: 🔴 | Sev 5 | Regions ❓ | Automated Nrrdbot incident | IC: @jaguilargorina | CL: ❓ | CM: ❓
Jordi Aguilar 5:18 AM
911 region staging
nrrd-bot-staging APP 5:18 AM
  Region set by @jaguilargorina
  Current region is STAGING
```

Example incident declared in Slack

HOW DEVOPS TEAMS FIND OUT ABOUT ISSUES[2]



2: Source: "DevOps Survey Results," 2nd Watch, 2018.

CHAPTER 4

Overcoming Microservices Complexity

Overcoming Microservices Complexity

Like peanut butter and chocolate, microservices and DevOps are better together. By now, companies understand that transforming monolithic applications into decomposed services can drive dramatic gains in productivity, speed, agility, scalability, and reliability.

But while teams recognize the changes required in developing, testing, and deploying microservices, they often overlook the substantial changes required in collaboration and communications. Engineers at New Relic developed the following best practices to foster a collaborative environment that simplifies the complexities and communication challenges inherent in a microservices world.

Good communication practices for a microservices environment

- **Let upstream and downstream dependencies know of major changes:** Before you deploy any major changes to your microservice, notify the teams that depend on it both upstream and downstream so that in case any issues arise, they won't waste time trying to track down the root cause.
- **Communicate early and often:** This is particularly important for versioning, deprecations, and situations where you may need to temporarily provide backwards compatibility.
- **Treat internal APIs like external ones:** Make your API developer-friendly with documentation, informational error messages, and a process for sending test data.
- **Treat downstream teams like customers:** Create a README with an architecture diagram, description, instructions for running locally, and information on how to contribute.
- **Create an announcements-only channel:** It's essential to have a single source of truth for important announcements rather than expecting teams to glean important information from multiple boards, discussions, and emails.
- **Focus on your service "neighborhood":** Your upstream, downstream, infrastructure, and security teams are all "neighbors" of your service. As a good neighbor, you should attend their demos and standups, give your neighbors access to your service's roadmap, and maintain a contact list.

Using data to better understand how microservices are working

Breaking up your monolithic applications into microservices isn't an easy task. First you need deep understanding of a system before you can partition it into service boundaries. Even then, partitioning it accurately so that you create true microservices (ones that are single-function, fine-grained, and only loosely coupled with applications and other services) can be tricky.

Here are some metrics and monitoring tips you can use to sniff out microservices in your environment that still have too many interdependencies with other services and applications:

1. **Deployments:** Is your team synchronizing deployments across upstream and downstream teams? If you see deployment markers in your monitoring solution that are synchronized across multiple services, you haven't truly decoupled your services.
2. **Communications:** A microservice should only need minimal communication with other services to execute its function. If you see a service that has many back-and-forth requests to the same downstream services, that's a clear sign that it's not decoupled. Throughput is another marker to check. If the number of calls per minute for a given microservice is significantly higher than the throughput of the application overall, it's a leading indicator that the service is not decoupled.
3. **Data stores:** Each microservice should have its own data store to prevent deployment problems, database contention issues, and schema changes that create problems for other services sharing the data store. Proper monitoring can show you whether each microservice is using its own data store.
4. **Scalability:** In a true microservices environment, spikes in services populating on hosts should correspond with spikes in throughput on individual services. This indicates dynamic scaling, one of the top benefits of microservices. On the other hand, if you see corresponding spikes across all services and hosts, there's good reason to believe that your services aren't decoupled.
5. **Developers per application:** If you have effective communication across your microservices teams, then you won't need "architecture gurus" who touch every microservice to ensure they play nicely. If you have 100 engineers and 10 services, and two of your engineers are developing on all 10 services, that's probably a sign that those services aren't really decoupled, and that the development of all your services relies on the tribal knowledge and communication skills of those two developers.

MOST BELIEVE THAT MICROSERVICES AND CONTAINERS ARE ESSENTIAL[3]

80%

Believe microservices and container-based enablement capabilities are essential, very important, or important, but only **one in four** believe their organization can quickly deliver those capabilities.

3: Source: "Enterprise Priorities for Hybrid Cloud Management," Ponemon Institute, June 2018.

CHAPTER 5

Using Data to Speed Software Development

Using Data to Speed Software Development

The fundamental driver for DevOps is speed—faster delivery of software, faster resolution of problems, faster innovation. But how can your business achieve the speed it needs to grab nascent market opportunities, out-innovate the competition, and keep customers happy?

Here at New Relic, we know that data is the fuel for DevOps success because it helps you:

- Measure and track DevOps performance
- Provide instant feedback that gets everyone focused on the right things
- Optimize software delivery, performance, and business results

One of the most common questions we get from customers regarding DevOps is "How should we start?" Here are some core principles that should guide your journey to DevOps at scale:

Eliminate data silos

While measurement is a core tenet of DevOps, many teams don't realize that data silos mean that everyone is looking at something different when it comes to performance. When there are silos of performance data, there's no cohesive, common language across applications, infrastructure, and user experience. Deploying instrumentation across all of your systems to see everything in one platform gets everyone on the same page, using common data and metrics to bring together different functions into one team.

Here at New Relic, our engineering teams were struggling with a proliferation of tools and competing priorities. Getting everyone to agree on which SLOs were most important, and then putting those SLOs in shared dashboards, allowed these teams to starting pulling in the same direction.

Simplify the complex

While modern application architectures help simplify and accelerate development in many ways, the dynamic quality of today's modular architecture creates a new type of complexity, with many individual components that make up a cohesive whole. Getting a complete view of your architecture, no matter how ephemeral, is key to coping with this complexity. Having the right data will tell you what's working and where to focus your team.

Understand the impact of changes

With DevOps, code deploys and changes are more frequent and your team needs to stay on top of them to avoid potential issues. Make sure you have a reporting capability that shows recent deployments and the before/after impact on application performance and customer experience, including any errors that occurred. This way you can quickly correlate changes to potential impacts and allow your team to respond quickly and rollback a release or provide a quick resolution to any incidents that have occurred.

Conclusion

The popularity of DevOps has reached mainstream proportions, with organizations across many different industries pursuing improved speed, productivity, quality, and innovation through the adoption of DevOps principles. By reaching optimal digital velocity, high-performing DevOps organizations have been shown to deploy far more frequently with dramatically faster time to recover from downtime than their low-performing peers.

The fuel for a high-performance DevOps engine is data. Best practices, like those in this ebook, are all about how to use that data effectively to drive success. New Relic gives you the end-to-end visibility you need to monitor your DevOps efforts and continuously improve outcomes at every stage.

Successful DevOps starts here. Get started at newrelic.com/devops

Good monitoring has been a staple of high-performing teams. In previous years, we found that proactively monitoring applications and infrastructure, and using this information to make business decisions, was strongly related to software delivery performance.[4]

4: Source: "Accelerate: State of DevOps 2018: Strategies for a New Economy," DORA

New Relic®