MINI-BOOK
MB01 – Sep 2022 v3

conflux

MODERN SOFTWARE DELIVERY

Continuous Delivery
Operability
SRE

Key industry insights in 6 articles

In association with TechBeacon

TechBeacon

Modern Software Delivery

Continuous Delivery | Operability | SRE

The rapid pace of change across IT leaves many organizations struggling to keep up. The complementary practices of Continuous Delivery, Operability, and Site Reliability Engineering (SRE) offer proven ways to make software delivery effective and responsive.

At Conflux, we've been leading industry thinking and practices for many years though our consulting, training, and publications. We're excited to present this collection of articles relating to modern software delivery in association with TechBeacon.

All the articles first appeared on TechBeacon.com and are reproduced here with permission.

conflux

In this mini-book:

9 ways organizations screw up continuous delivery

Matthew Skelton, Head of Consulting, Conflux

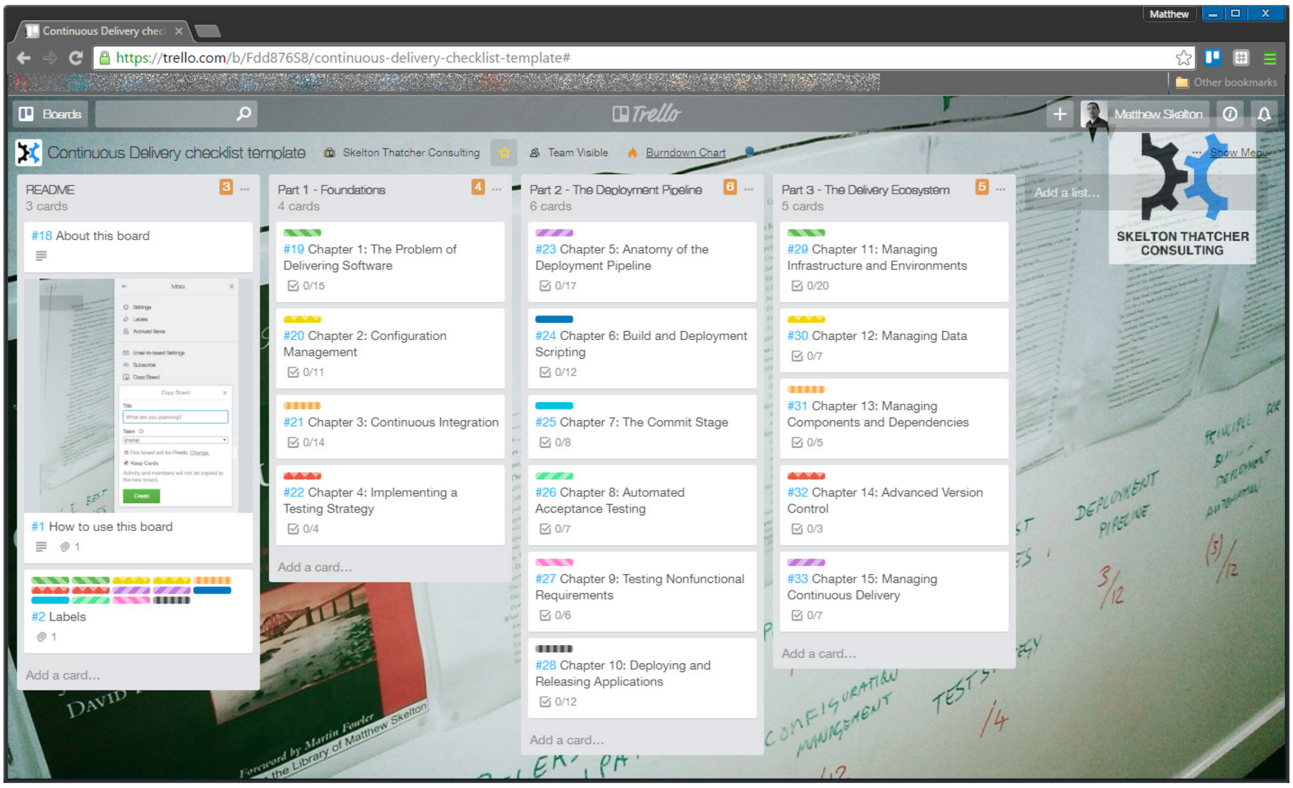
Continuous delivery (CD) is a specific set of practices for reliable software delivery that's achieved by automating build and deployment and testing software changes. However, many organizations screw up their approaches to CD by not adopting some key practices.

Over the past seven years, I have helped over 30 establishments in various parts of the world adopt CD, and I have seen several repeated mistakes. Here are nine you should avoid.

1. Not reading *Continuous Delivery*

Amazingly, some people claim to be doing CD, and yet they have not read the book *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, by Jez Humble and Dave Farley. This is where the term continuous delivery was defined. Yes, technically the phrase is also in the Agile Manifesto, but the notion of rapid, regular, reliable releases that most people think of with the phrase was codified in this 2010 book. Most importantly, CD does not require that code changes be pushed to production on every change; that's continuous deployment. That might be useful in your context, but you'll get massive benefits from following the CD practices even with more selective releases.

The way in which the chapter subheadings are written in the book means that you can use them as a kind of checklist or progress indicator for your organization. The contents pages from the book can be printed out and stuck on a wall to act as a progress chart. I have recently collated the recommendations from the book in a useful, free online tool at cdchecklist.info, where you can see the checklists in action.



https://trello.com/b/Fdd876S8/continuous-delivery-checklist-template#

Continuous Delivery checklist template — Skelton Thatcher Consulting — Team Visible — Burndown Chart

README (3 cards)
- #18 About this board
- #1 How to use this board
- #2 Labels

Part 1 - Foundations (4 cards)
- #19 Chapter 1: The Problem of Delivering Software — 0/15
- #20 Chapter 2: Configuration Management — 0/11
- #21 Chapter 3: Continuous Integration — 0/14
- #22 Chapter 4: Implementing a Testing Strategy — 0/4

Part 2 - The Deployment Pipeline (6 cards)
- #23 Chapter 5: Anatomy of the Deployment Pipeline — 0/17
- #24 Chapter 6: Build and Deployment Scripting — 0/12
- #25 Chapter 7: The Commit Stage — 0/8
- #26 Chapter 8: Automated Acceptance Testing — 0/7
- #27 Chapter 9: Testing Nonfunctional Requirements — 0/6
- #28 Chapter 10: Deploying and Releasing Applications — 0/12

Part 3 - The Delivery Ecosystem (5 cards)
- #29 Chapter 11: Managing Infrastructure and Environments — 0/20
- #30 Chapter 12: Managing Data — 0/7
- #31 Chapter 13: Managing Components and Dependencies — 0/5
- #32 Chapter 14: Advanced Version Control — 0/3
- #33 Chapter 15: Managing Continuous Delivery — 0/7

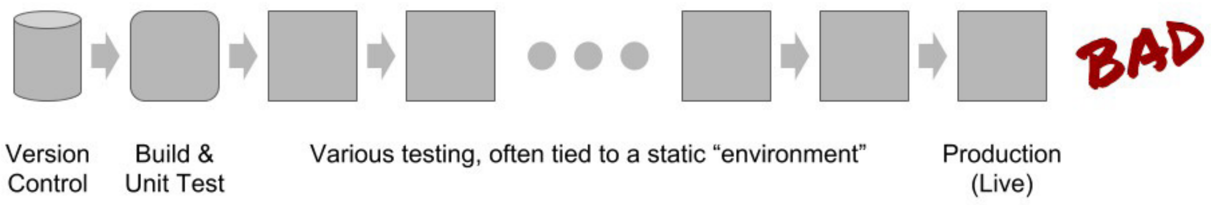
SKELTON THATCHER CONSULTING

The online continuous delivery checklist tool at cdchecklist.info.

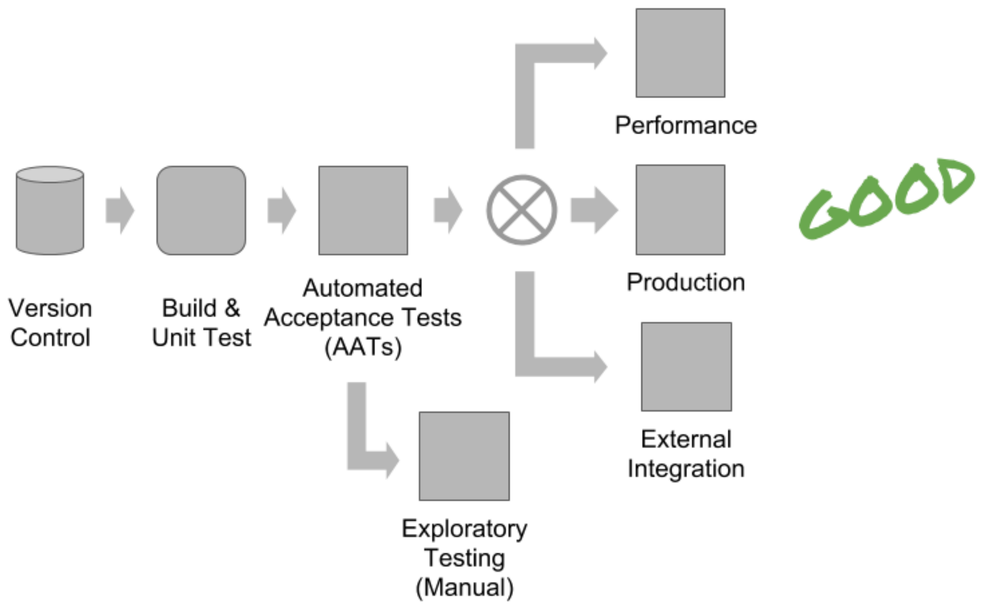
2. Using long, slow deployment pipelines

Another common screw-up I see is that organizations create deployment pipelines that have many steps in sequence, leading to a long wait between committing code to version control and seeing the changes in production (live). These long, slow deployment pipelines kill the rapid feedback essential for teams to be able to act effectively on failures and learn from monitoring.

Instead, use short, wide deployment pipelines that give teams and product managers enough flexibility to decide which tests to run based on the nature of each specific change. After committing code to version control, build the binaries and run unit tests (and perhaps code metrics checks). Then run a set of automated acceptance tests based on application features (which include



Long, slow deployment pipelines break rapid feedback from the deployment process.



Short, wide deployment pipelines optimize for rapid feedback and rapid deployment decisions.

both user-facing and operational features).

If these checks pass, then by definition the team or product manager may have the confidence to deploy to production. They also have the option to run additional checks if they wish, but the additional checks are not mandatory and depend on the context of the exact change made.

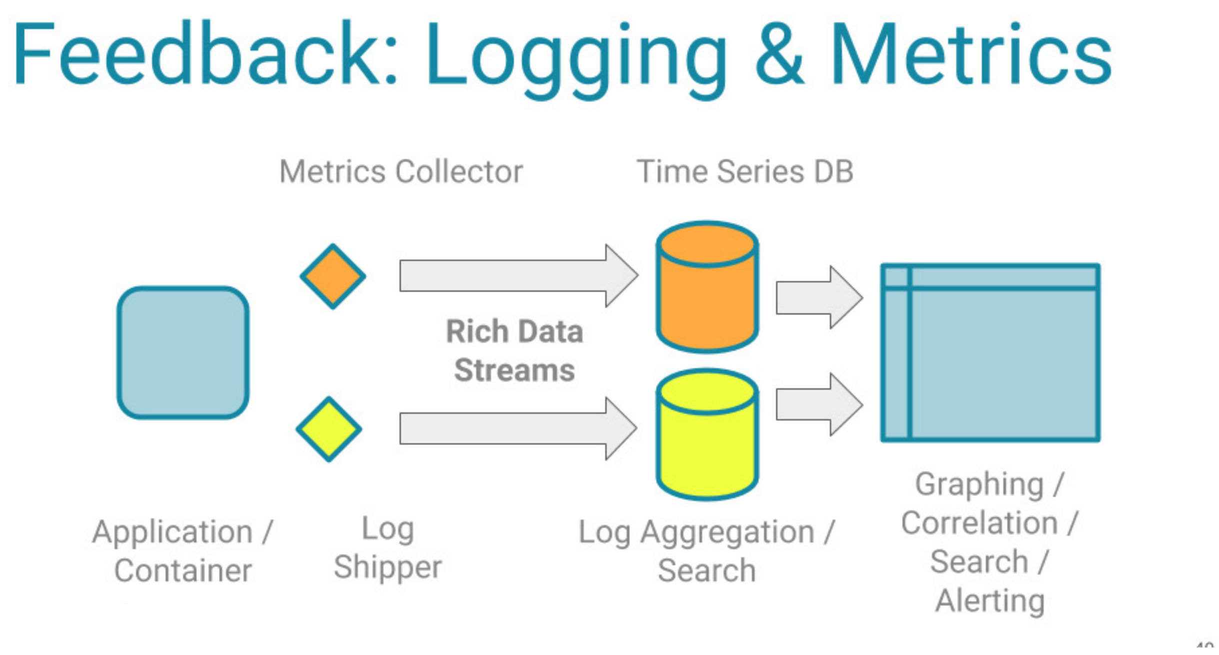
3. Thinking 'CD is not for us'

Together with a great team, I have run the Pipeline Conference in London since 2014. Pipeline was the first conference in Europe dedicated to CD. At both Pipeline and its sibling meetup group, LondonCD, we have had over 130 speakers from around the world and many different organizations talking about how they have used CD principles to make software delivery reliable and rapid. The variety of situations in which CD works is what makes it such a valuable approach.

The wide applicability of CD principles is borne out by my consulting experience. In fact, every software delivery situation I have encountered in the last seven years has (or could have) benefited from following almost every CD principle and practice—without exception. Yes, continuous delivery is for you!

4. Having no effective logging or metrics

Many organizations try to adopt the more rapid pace of change enabled by CD practices without a solid foundation of operational telemetry. This lack of metrics and logging begins to hurt as deployments become more frequent. Logging and metrics are crucial "sensing mechanisms" for teams building modern software systems, rather like the radar and video pattern-recognition systems that help autonomous vehicles to travel safely at speed on the highways.



Continuous delivery needs rich telemetry from modern logging and metrics tooling.

Modern software needs good logging. This is a defined set of event types agreed upon between developers and operations people, together with correlation IDs for cross-machine request traceability. Logs are automatically aggregated in a centralized, off-the-shelf logging system that provides access to production log data for developers, testers, and operations people alike through APIs and a browser interface. There is a similar need for aggregated metrics and a shared dashboard and query interface for teams.

5. Underinvestment in build and deployment

Understanding and evolving how we build and deploy our software is absolutely crucial for effective CD. Too many organizations seem to undervalue the importance of focusing on build-and-deployment activities; they prefer instead to add more development teams to focus on "features." These companies don't realize that the cause of delay with feature delivery is often the creaking build-and-deployment systems or the complexity in automated testing.

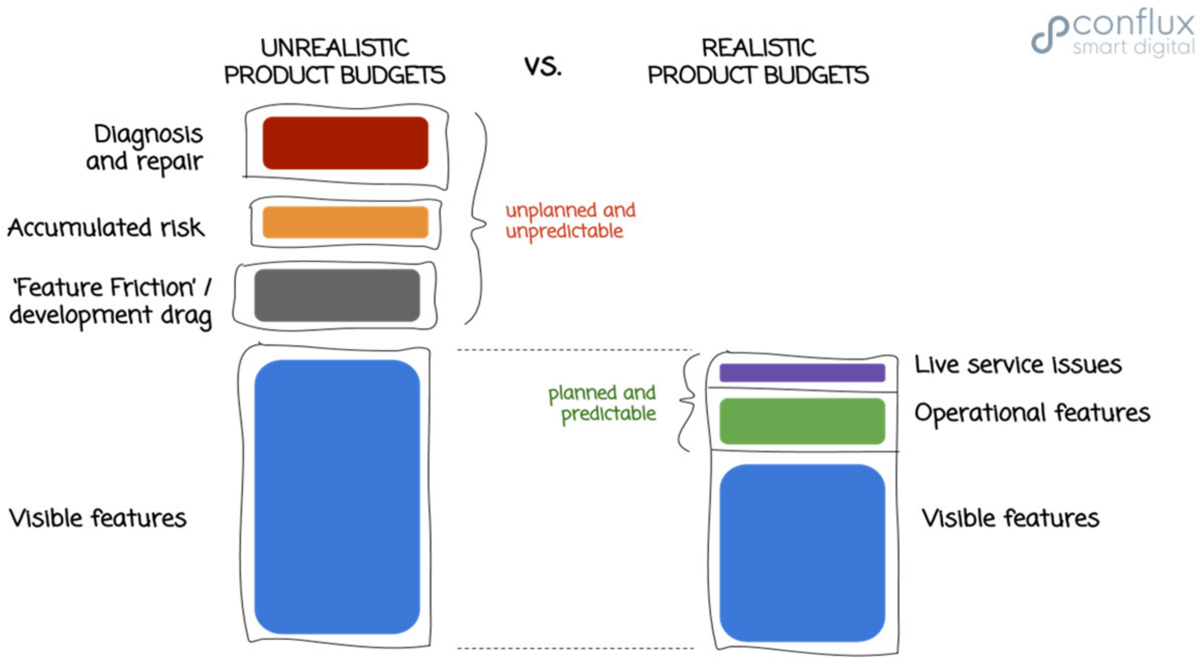
The organizations that succeed with rapid, reliable software releases invest heavily in build-and-deployment activities. A combination of outsourced (typically SaaS) tooling and a group of experienced build-and-deployment engineers helps to smooth and accelerate the flow of well-tested changes from development to production.

In my experience, a good rule of thumb is to budget for one full-time equivalent person per software development team of seven to nine people. So an organization with eight software development teams should expect to have around eight people dedicated to build and deployment.

Even if all the build-and-deployment tooling is SaaS-hosted, the build-and-deployment people will have a crucial role to play—in considering versioning approaches and interdependencies, evaluating new techniques, splitting and joining components, and ensuring SaaS infrastructure availability. If any tools are run in-house, then the team will need to deal with even more.

6. Not addressing operational aspects well

By adopting CD, organizations are able to respond to business needs more rapidly with an increased pace of delivery. This also means that problems with development practices (such as feature prioritization) are surfaced more rapidly than before. In particular, if operational aspects of the software systems are not addressed every week, features will become ever more difficult to add to the software. This is what I call "feature friction."



A focus on only visible features leads to increasing feature friction and live service issues.

A good approach is to spend around 30% of product budget each week on operational aspects of the software, such as logging, metrics, ops dashboards, deployment automation, and a planned allocation for dealing with live service incidents. More advanced organizations actually track the amount of time and effort spent on operational aspects and alert if the spend is outside of an accepted range, allowing rapid course correction within a week.

7. Forgetting the database

Let's be honest: CD with data and databases is difficult, although it's easier than it has been in the past. In some organizations, the production databases are "safeguarded" by database administrators (DBAs) and are seen as the definitive version of the database. Usually these production databases have many special tweaks not found elsewhere, such as data manipulation jobs, sharding, or replication settings. These all make it difficult for developers to test database changes in upstream environments, leading to problems in production, which lead to more restrictions on production access, and so on—a vicious circle.

A better approach is to make all database changes flow down from version control in development to production by using one of the many off-the-shelf tools (commercial or free) to evolve the database schema and settings. In this way, only transactional data and new content are created new in production; all other changes are tested before going live, in upstream environments. There are many off-the-shelf tools to choose from, including ApexSQL, ActiveRecord (and similar), DbMaestro, FluentMigrator, Flyway, Liquibase, and a comprehensive suite of tools from Redgate.

8. 'Just plugging in a deployment pipeline'

By its nature, CD emphasizes the importance of build, deployment, and release as key areas of focus for modern software. This means that you may need to refactor or possibly substantially rebuild your software systems to make them more suitable for rapid, reliable releases. Simply putting your existing clunky software into a deployment pipeline and hoping for the best is probably not the ideal approach. Instead, invest in taking the time needed to re-architect your software to make it more suitable for CD.

9. Coveting the 'latest shiny' thing

A perpetual problem in the software industry is the blind pursuit of the next thing—the "latest shiny"—without first addressing core practices and ways of working. At the moment, this is driving a rush toward containers and microservices. But in many cases, the added complexity of these approaches has been ignored. The rapid, reliable software releases we get from CD needs core engineering practices to be in place; otherwise problems occur.

For example, a company in London recently asked me to assess its readiness to move to containers for faster development and deployment. The office was cramped and way too small for the number of people working there, leaving no room for whiteboard discussions, team meetings, etc.

Also, there were no unit tests or integration tests for most of the code, few applications with effective logging or monitoring, and over 200 ETL data processing jobs that existed only in production. To make matters worse, the production database was running on a single server with no high availability or failover. To paraphrase Bridget Kromhout, containers were not going to fix their broken culture.

A checklist to remember:

Here are the nine steps to avoid screwing up CD:

1. Read the Continuous Delivery book by Humble and Farley.
2. Use short, wide deployment pipelines that empower decision makers.
3. Realize that continuous delivery is for you! Think of CD as a set of excellent practices for building working software systems of every kind.
4. Use good modern logging techniques together with details metrics to drive decision making.
5. Invest in a ratio of one full-time equivalent for build and deployment activities per nine-person development team.
6. Spend 30% of product budget on operational aspects every week.
7. Use an off-the-shelf tool for database changes and drive changes from version control.
8. Re-architect your software systems to suit CD.
9. Adopt good software development practices before adding technical complexity.

In the words of Dave Farley, co-author of the Continuous Delivery book, *Continuous delivery is an engineering discipline*; it needs investment to make it work.

[Original: https://techbeacon.com/devops/9-ways-organizations-screw-continuous-delivery]

How to find the right DevOps tools for your team

Matthew Skelton, Head of Consulting, Conflux

When you adopt a DevOps approach to building and operating software systems, you must rely on modern tools for almost every aspect of build, release, and operations activities. But before you get into the weeds of comparing one tool against another, you need to think more broadly about what you need.

And there are many types of DevOps tools to consider. With DevOps, many previously manual or semi-manual activities are fully automated, including version control (for application code, infrastructure code, and configuration), continuous integration (for application code and infrastructure code), artifact management (packages, container images, container applications), continuous delivery deployment pipelines, test automation (unit tests, component tests, integration tests, deployment tests, performance tests, security tests, etc.), environment automation and configuration, release management, log aggregation and search, metrics, monitoring, team communications (chat, video calling, screen sharing), and reporting.

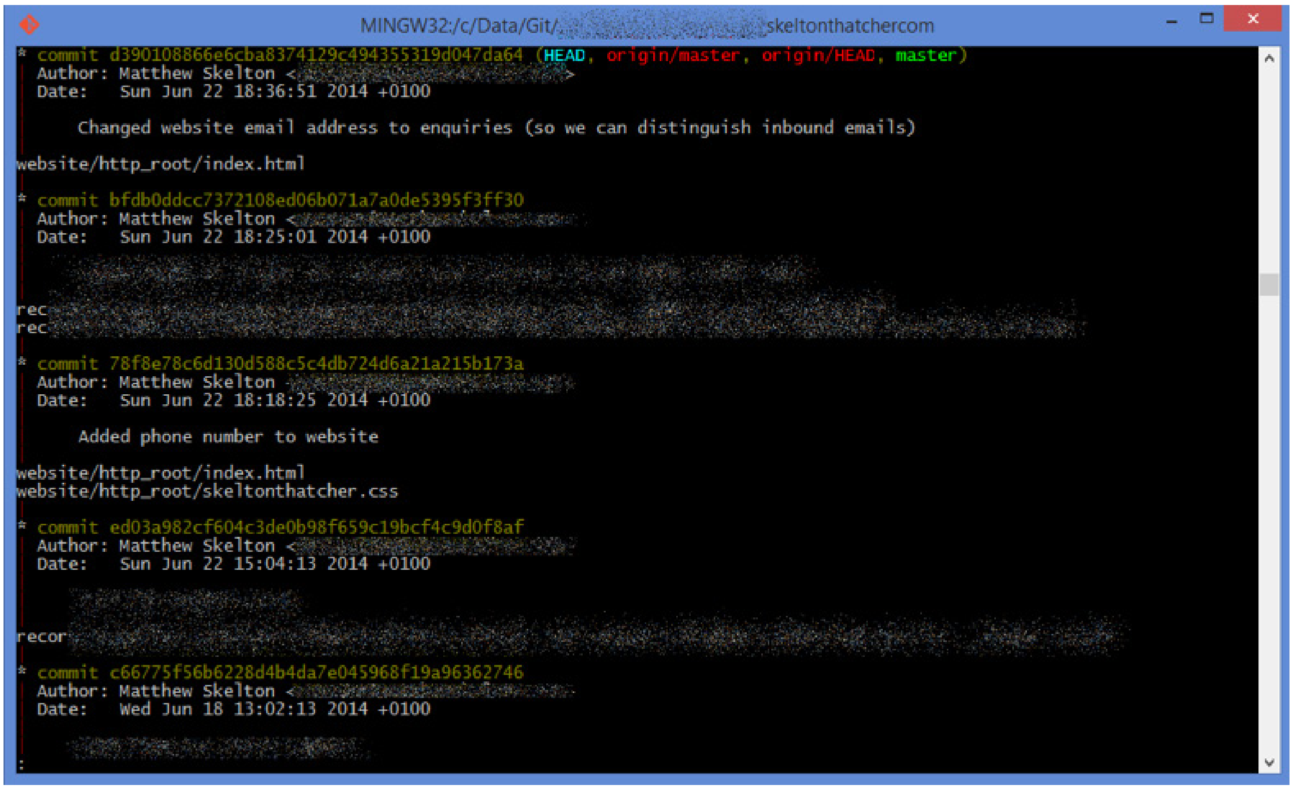
You'll find plenty of excellent tools in all of these categories, but it's easy to get hung up on the pros and cons of using one tool versus another. And while sometimes that's the right debate to have, confusion around tools may be a symptom of deeper problems with respect to the way in which your team uses those tools, or how you introduce those tools to the team.

I have been using the guidelines below with clients for about five years, and we've managed to solve tooling-related problems that would otherwise have descended into an unhelpful product X-versus-Y shooting match. To become a high-performing organization, you must take into account the social dynamics of your organization and the trajectory of the rapidly evolving public cloud vendors.

Choose tools that facilitate collaboration

Having highly effective collaboration between teams is critical for DevOps. Some people think they need to buy a dedicated collaboration tool for this purpose, but there are many different tools you can use to enhance collaboration.

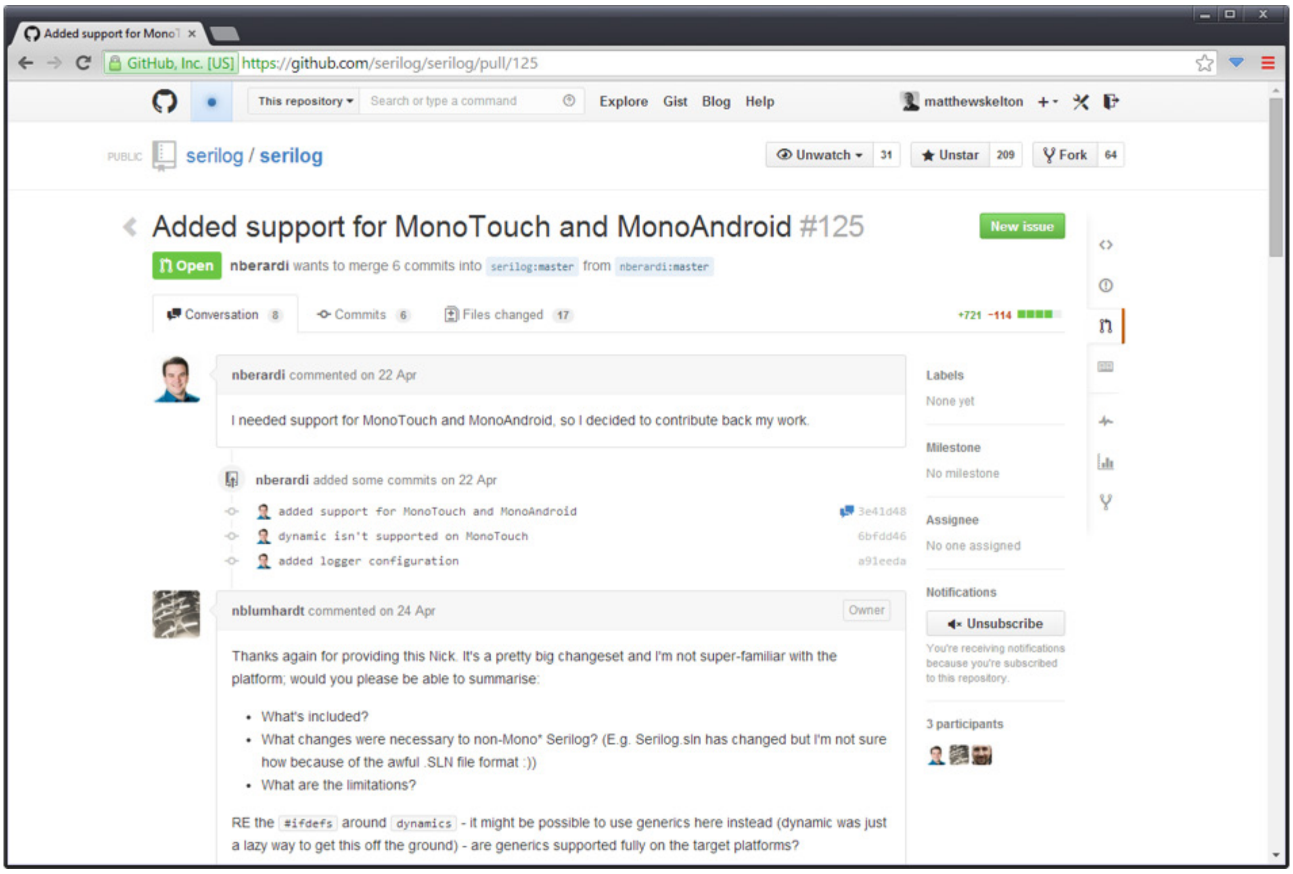
Consider one of the cornerstones of a DevOps approach: version control. Let's say you're trying to encourage more people in the organization to use version control, including for database scripts, configuration files, and so on. If you insist that everyone use only a command-line tool for version control, you'll miss out on collaboration opportunities:



```
* commit d390108866e6cba8374129c494355319d047da64 (HEAD, origin/master, origin/HEAD, master)
| Author: Matthew Skelton <[redacted]>
| Date:   Sun Jun 22 18:36:51 2014 +0100
|
|     Changed website email address to enquiries (so we can distinguish inbound emails)
|
website/http_root/index.html
|
* commit bfdb0ddcc7372108ed06b071a7a0de5395f3ff30
| Author: Matthew Skelton <[redacted]
| Date:   Sun Jun 22 18:25:01 2014 +0100
|
|     [redacted]
|
rec[redacted]
rec[redacted]
|
* commit 78f8e78c6d130d588c5c4db724d6a21a215b173a
| Author: Matthew Skelton [redacted]
| Date:   Sun Jun 22 18:18:25 2014 +0100
|
|     Added phone number to website
|
website/http_root/index.html
website/http_root/skeltonthatcher.css
|
* commit ed03a982cf604c3de0b98f659c19bcf4c9d0f8af
| Author: Matthew Skelton <[redacted]
| Date:   Sun Jun 22 15:04:13 2014 +0100
|
|     [redacted]
|
recor[redacted]
|
* commit c66775f56b6228d4b4da7e045968f19a96362746
| Author: Matthew Skelton <[redacted]
| Date:   Wed Jun 18 13:02:13 2014 +0100
|
|     [redacted]
:
```

Command-line tools can be a barrier to collaboration for some people.

The command-line view of version control is certainly part of a DevOps tool set, but it is unfamiliar to many people—especially non-developers—and has no obvious collaboration potential. But if you use a richly featured version-control platform such as Github, Bitbucket, or Gitlab, you can take advantage of discussion threads around file changes to get people talking about why a file changed. This helps you collaborate with people who have different skills, and encourages more people to learn how to use version control:



Added support for MonoTouch and MonoAndroid #125

nberardi wants to merge 6 commits into `serilog:master` from `nberardi:master`

nberardi commented on 22 Apr

I needed support for MonoTouch and MonoAndroid, so I decided to contribute back my work.

nberardi added some commits on 22 Apr

```
added support for MonoTouch and MonoAndroid     3e41d48
dynamic isn't supported on MonoTouch            6bfdd46
added logger configuration                      a91eeda
```

nblumhardt commented on 24 Apr

Thanks again for providing this Nick. It's a pretty big changeset and I'm not super-familiar with the platform; would you please be able to summarise:

- What's included?
- What changes were necessary to non-Mono* Serilog? (E.g. Serilog.sln has changed but I'm not sure how because of the awful .SLN file format :))
- What are the limitations?

RE the `#ifdefs` around `dynamics` - it might be possible to use generics here instead (dynamic was just a lazy way to get this off the ground) - are generics supported fully on the target platforms?

You're receiving notifications because you're subscribed to this repository.

Browser-based tools can help to encourage collaboration for people who are less technology-savvy.

Using a browser-based version-control platform opens up version control to a wider audience than just software developers, which in turn helps you to emphasize the importance of version control as a key DevOps practice. By choosing a version-control tool with discussion capabilities and making it available to a wide audience, you can enable rich communication between teams and groups within your organization.

The same approach works for many other tools, too. I once consulted with an organization that had a tool for log aggregation and search. The IT operations people found it valuable, but the developers did not have permission to search the logs from the production systems. Access was denied because, IT claimed, the data was of a sensitive nature. But the managers wanted to improve the way in which the IT Ops and Dev teams collaborated. So they opened up access to the log-search tool for developers and—surprise— developers and operations people collaborated more. The tool hadn't changed, but changing access permissions enhanced collaboration.

Key points:

- **Value collaboration as a key selection aspect of tools.**
- **Look behind the tool's main purpose to find collaboration opportunities.**
- **Ask, "How does our use of this tool help or hinder people in collaboration?"**

Favor tools with APIs

Modern software development needs delivery tools that are highly automatable, yet customizable. That means you need a fully featured API for each tool—preferably one that's HTTP-based. When you compose capabilities by gluing together API-rich tools, you enable easy wiring for alerts and other events. Avoid tools that try to do everything from within their own frames of reference; favor those that do one or more jobs well and integrate easily with other tools.

Given the speed of change in the software sector, it's particularly important to choose tools that meet these criteria. If you do, then when a new tool comes along you'll be able to replace your old tool with minimal disruption. Being stuck with a big, lumbering tool set that's only half-good at most things has been a source of significant pain for organizations trying to adopt DevOps. Keep your tooling nimble and composable to give your team the flexibility to adapt new approaches easily.

But beware of "spaghetti" tooling that's chained together with undocumented scripts. Treat your software delivery and operations tools like a proper production system. At the rapid pace enabled by DevOps, it's essential to be able to keep the tools you use for software delivery and operations running and working 24x7. Many companies make the mistake of adopting new tools without the operational support and care needed to make those tools work well. So when adopting new tooling, consider starting with SaaS-hosted offerings and running internal prototypes/demo versions before building an internal capability.

Key points:

- Choose tools that expose APIs.
- Aim for composition of new capabilities from multiple API-driven tools.
- Build and deployment are first-class concerns.

Favor tools that can store configuration in version control

One core tenet of DevOps is that you should store all configuration settings in version control. That includes the configuration not just for your custom software applications, but also for tools you use in software delivery and IT operations.

To be effective in a DevOps context, each tool must expose its configuration in such a way that you can store the configuration as text files that you can then expose to version control. Then you can track the history of configuration changes and test changes beforehand.

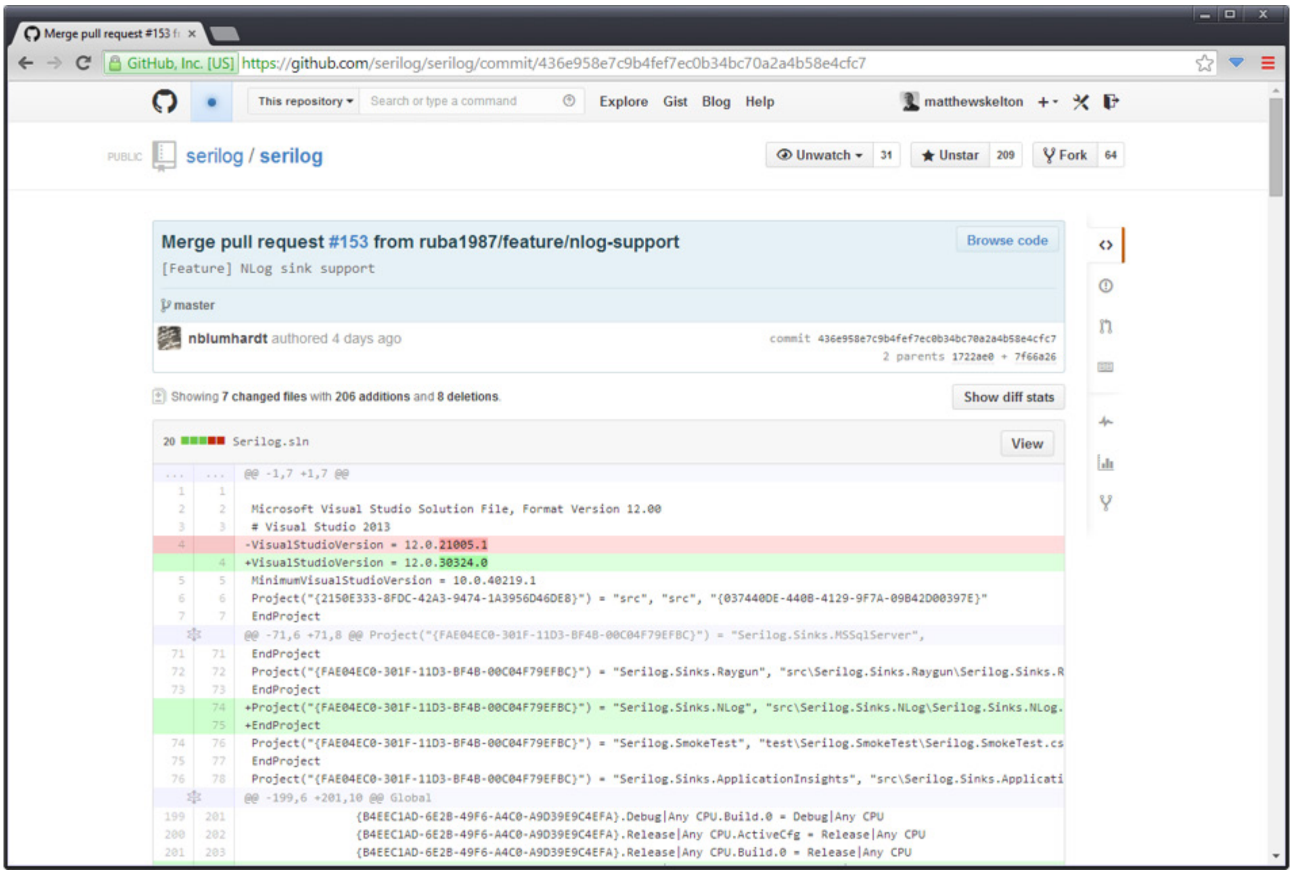
Why would you want to do that? If you cannot track and test configuration changes to your delivery and operations tooling, you risk breaking the machinery that makes DevOps work.

Key points:

- Choose tools that expose configuration to version control.
- Point-and-click is no longer acceptable for configuration of tools.

Use your tools in a way that encourages learning

Some of the tools useful for DevOps are quite involved and complicated, especially for people new to them; don't expect everyone to understand or adopt difficult new tools immediately. In fact, if you introduce a tool that is too tricky, some people may become hostile, especially if you don't provide training or coaching. That sometimes happens when organizations select best-of-breed tools without considering how easy they are to use.



Merge pull request #153 from ruba1987/feature/nlog-support

```
[Feature] NLog sink support
```

master

nblumhardt authored 4 days ago

```
commit 436e958e7c9b4fef7ec0b34bc70a2a4b58e4cfc7
2 parents 1722ae0 + 7f66a26
```

Showing **7 changed files** with **206 additions** and **8 deletions**.

20 🟩🟩🟩🟥🟥 Serilog.sln

```
@@ -1,7 +1,7 @@

 Microsoft Visual Studio Solution File, Format Version 12.00
 # Visual Studio 2013
-VisualStudioVersion = 12.0.21005.1
+VisualStudioVersion = 12.0.30324.0
 MinimumVisualStudioVersion = 10.0.40219.1
 Project("{2150E333-8FDC-42A3-9474-1A3956D46DE8}") = "src", "src", "{037440DE-440B-4129-9F7A-09B42D00397E}"
 EndProject
@@ -71,6 +71,8 @@ Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "Serilog.Sinks.MSSqlServer",
 EndProject
 Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "Serilog.Sinks.Raygun", "src\Serilog.Sinks.Raygun\Serilog.Sinks.R
 EndProject
+Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "Serilog.Sinks.NLog", "src\Serilog.Sinks.NLog\Serilog.Sinks.NLog.
+EndProject
 Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "Serilog.SmokeTest", "test\Serilog.SmokeTest\Serilog.SmokeTest.cs
 EndProject
 Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "Serilog.Sinks.ApplicationInsights", "src\Serilog.Sinks.Applicati
@@ -199,6 +201,10 @@ Global
                {B4EEC1AD-6E2B-49F6-A4C0-A9D39E9C4EFA}.Debug|Any CPU.Build.0 = Debug|Any CPU
                {B4EEC1AD-6E2B-49F6-A4C0-A9D39E9C4EFA}.Release|Any CPU.ActiveCfg = Release|Any CPU
                {B4EEC1AD-6E2B-49F6-A4C0-A9D39E9C4EFA}.Release|Any CPU.Build.0 = Release|Any CPU
```

Command-line tools can be daunting for some people and may hinder collaboration unless you provide training. Tools with a more friendly UI can help to bring people on board to new ways of working, giving them the confidence to adopt command-line tools later.

Assess the skills in your organization and devise a tools roadmap for moving teams to improved ways of working. Select tools that offer more than one way to use them (GUI, API, command-line) so people can learn at their own pace. And avoid leaving people behind on the climb to more advanced approaches by holding regular team show-and-tell sessions to demonstrate tools and techniques.

For example, you might start with the browser-based interface, such as the one below, for people new to version control, giving them time to adjust to this approach before training them on the command-line tools for version control.

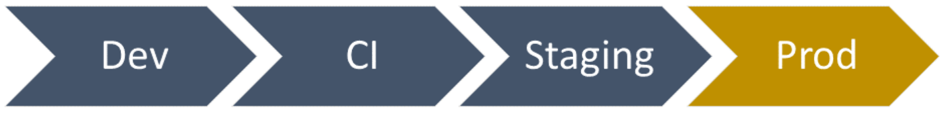
DevOps is a journey from mostly manual to fully automated, and not everyone starts from the same place. Give people time and space to become familiar with new tools and approaches. They might start with a simpler tool, then adopt a more powerful one later.

Key points:

- **Bring people with you on your DevOps journey.**
- **Prefer achievable gains now over possible future state.**
- **Avoid a fear of too-scary tools by stepwise evolution.**

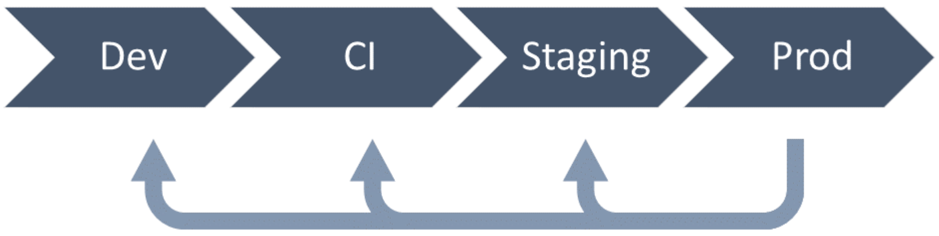
Avoid special production-only tools

The speed and frequency of change that DevOps gives you the means you need to emphasize the feedback loops within your delivery and operations processes. In particular, it is important that all technology people in your organization learn as much as possible about how the production environment works so they can build better-working, more resilient software. You also need to test changes to all parts of the software system before deploying new versions to production.



Production-only tools prevent teams from learning because production is treated as a special case.

For an effective DevOps approach, choose tools that work easily in nonproduction environments (development, continuous integration, staging, etc.). The tool should be cheap enough to buy or install so that you can install it in all environments, including developer laptops and the automated build-and-test system. A tool that is so expensive that you can only afford a license for production is not a good tool for DevOps. Such "singleton" tools tend to accrue an aura of magic, leading people to think that production is special. People become disengaged, and that's a bad outcome. Good tools for DevOps are also easy to spin up in different environments using automated scripts. A tool that needs manual installation is not a good choice for DevOps.



Running the same tools in production as in all other environments enables rapid learning and increases engagement within teams.

In some sense, this "run it anywhere" approach to tools for DevOps makes production less special, and rightly so. Many of the problems with older, fragile IT systems are the result of production being treated in a special way, preventing developers and testers from learning how production works. With a DevOps approach, your aim is to choose tools that are easy to install and can spin up in multiple environments, even if the feature set is less impressive than that of a tool that is more advanced but difficult to configure. Aim to optimize globally across teams that need to collaborate, not just locally for production.

Key points: Production-only tools...

- **Break the learning feedback loop.**
- **Make CI/CD more difficult.**
- **Underestimate the value of collaboration and learning.**

Choose tools that enhance inter-team communications

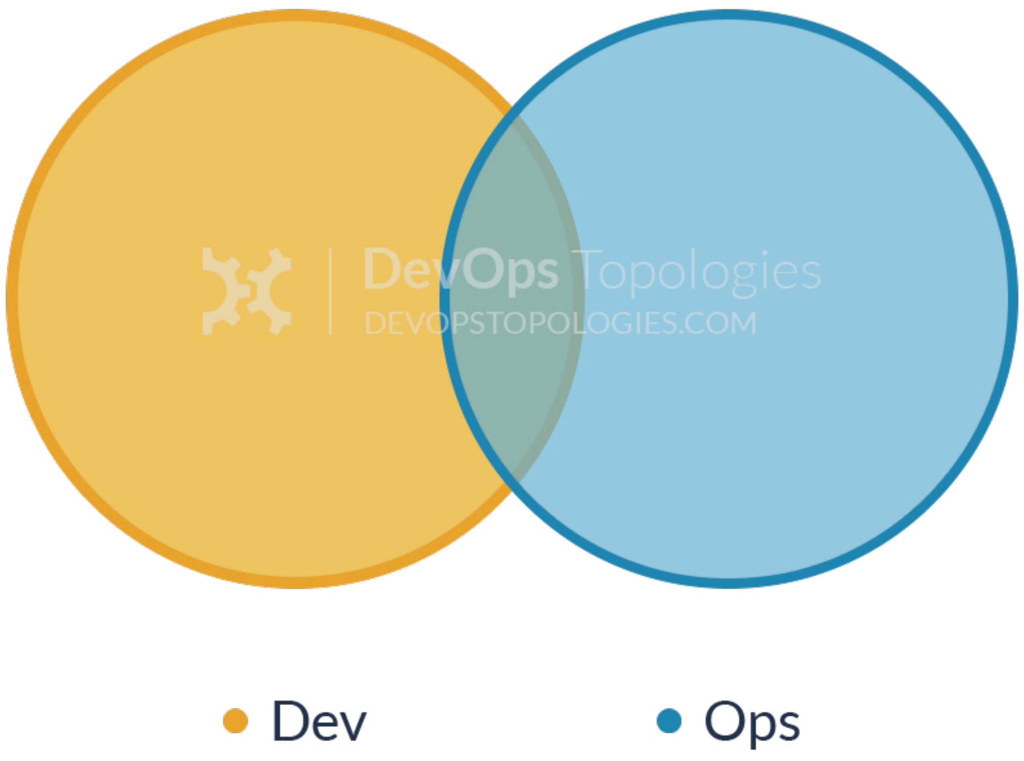
One of the most common problems I see in organizations struggling to build and run modern software systems in a DevOps way is a mismatch between the responsibility boundaries for teams or departments and those for tools. The organization either has multiple tools when a single tool would suffice (in order to provide a common, shared view), or it has a single tool that's causing problems because teams need separate ones.

In recent years, Conway's Law has been observed and measured in many studies. The communication paths in our organization drive the resulting system architecture:

> "Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations."
>
> — Mel Conway

You therefore need to be mindful of the effect of shared tools on the way in which teams interact. If you want your teams to collaborate, then shared tools make sense. But if you need a clear responsibility boundary between teams, separate tools may be best. Use my DevOps team topologies patterns to understand which DevOps model is right for your organization, and then choose the tools that fit that model.

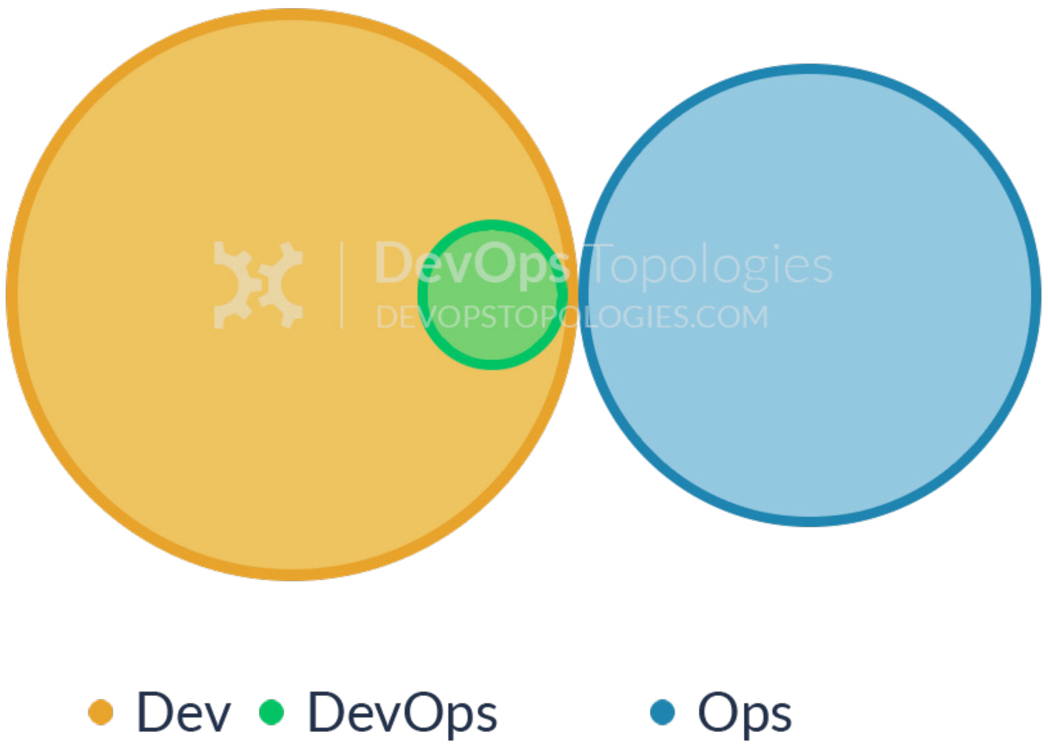
If you need the development team to work closely with operations (the Type



In a Type 1 platform model, smooth collaboration implies some shared tooling between Dev and Ops. Image from DevOpsTopologies.com and licensed under CC BY-SA license.

1 model), then having separate ticketing or incident management tools for Dev and Ops will result in poor inter-team communication. To help these teams collaborate and communicate, choose a tool that can meet the needs of both groups. But be sure that you understand the user experience needs of each group, since a tool that infuriates your engineers is a sure way to stop a DevOps effort dead in its tracks.

If, like many enterprises, yours is moving to a Type 3 (platform) model, then the platform team is not responsible for the live service of the applications; that's the responsibility of the product development teams. When responsibility boundaries don't overlap, you won't get much value from insisting on the same incident-tracking tool or even the same monitoring tool for the platform and development teams. This becomes even clearer when IT operations has outsourced to a cloud service provider, since in that case there's no question about forcing the same tools on two different teams.
In summary, don't select a single tool for the whole organization without considering team inter-relationships first.



A Type 3, IT Ops as infrastructure-as-a-service (platform) implies little need for shared tooling between Dev and Ops. Image from DevOpsTopologies.com and licensed under CC BY-SA license.

Key points:

- See the whole organization as a system you're building.
- Have separate tools for separate teams.
- Deploy shared tools for collaborative teams.

Optimize for learning, collaboration, automation, and team dynamics

When choosing tools for DevOps, it's important to avoid product X-versus-Y tooling shootouts that simply compare lists of features side by side. Sometimes that's needed, but only after you understand the broader implications of having one (or both) of those tools in place in your organization. Using tools in the wrong way—especially trying to make everyone use the same tool—can be counterproductive for DevOps.

Try to assess and understand where your team communication boundaries should be, using the DevOps team topologies patterns and Conway's Law, to avoid a one-size-fits-all approach to tools. Sometimes, using multiple, similar tools is the right approach, but that depends on your team boundaries.

Ensure that the tools you choose do not present a learning barrier to people who are new to DevOps approaches; expect to replace tools regularly as people develop their skills and establish new collaboration patterns.

Tools for DevOps need programmable APIs. Don't buy or use tools that need a human operator to click buttons on a browser application. With DevOps, you need to compose functionality from multiple, cooperating tools using APIs and "glue" scripts.

Finally, don't optimize for your production environment. A tool that exists only in the live production environment is a tool that you can't test upstream, and that's a dangerous approach in a fast-paced DevOps world.

[Original: https://techbeacon.com/devops/how-find-right-devops-tools-your-team]

Video training for fast flow

Lift awareness and capabilities at scale across your organization with expert-led video training on fast flow and Team Topologies.

confluxhq.com/training

conflux

5 proven operability techniques for teams

Matthew Skelton, Head of Consulting, Conflux

As software systems become more distributed and interconnected, you need to ensure that the software works well when operating live in production—what I call software "operability"—and you need to be able to observe its behavior.

With the goal of improving mutual understanding of software systems through collaboration—a true DevOps approach—here are five practical ways your team can collaborate to enhance the operability of your software systems.

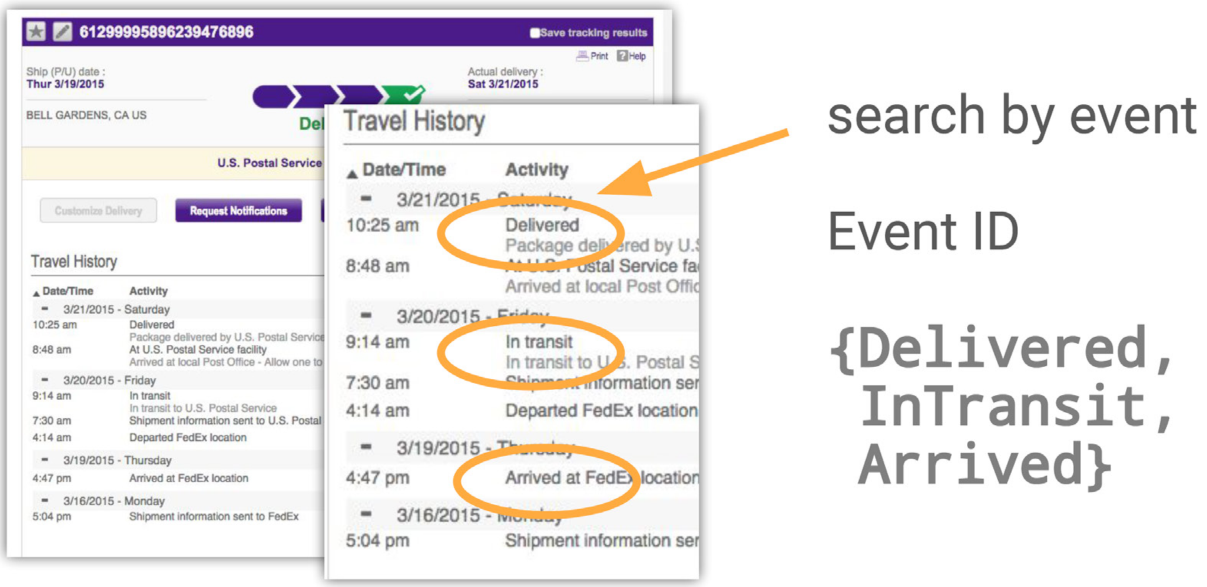
1. Collaborate on logging with event IDs to boost observability and awareness

Problem 1: Lack of observability for distributed systems.

Modern software requires teams to understand logging as a first-class concern. Using log traces from multiple machines is key to observing the behavior of runtime systems. With modern logging, you log to file (for servers) or STDOUT (for containers and serverless). The logs are then aggregated automatically into a central, searchable log store that's accessible with a browser UI and an HTTP API.

In the past, logging was seen as just a way to deal with errors, but this is a very ineffective use of logging. Since 2007, I have treated logging as a rich trace of application behavior, using unique identifiers such as an enum to represent distinct states that I also call event IDs. Examples of event IDs could be ServiceStarting, DatabaseConnectionOpened, PostcodeLookupFailed, NewUserRegistered, etc.

OpsLogger is a good example of a logging library that uses techniques



61299995896239476896

Ship (P/U) date : Thur 3/19/2015

Actual delivery : Sat 3/21/2015

BELL GARDENS, CA US

U.S. Postal Service

Travel History

```
Date/Time       Activity
- 3/21/2015 - Saturday
10:25 am        Delivered
                Package delivered by U.S. Postal Service
8:48 am         At U.S. Postal Service facility
                Arrived at local Post Office - Allow one to
- 3/20/2015 - Friday
9:14 am         In transit
                In transit to U.S. Postal Service
7:30 am         Shipment information sent to U.S. Postal
4:14 am         Departed FedEx location
- 3/19/2015 - Thursday
4:47 pm         Arrived at FedEx location
- 3/16/2015 - Monday
5:04 pm         Shipment information sent to FedEx
```

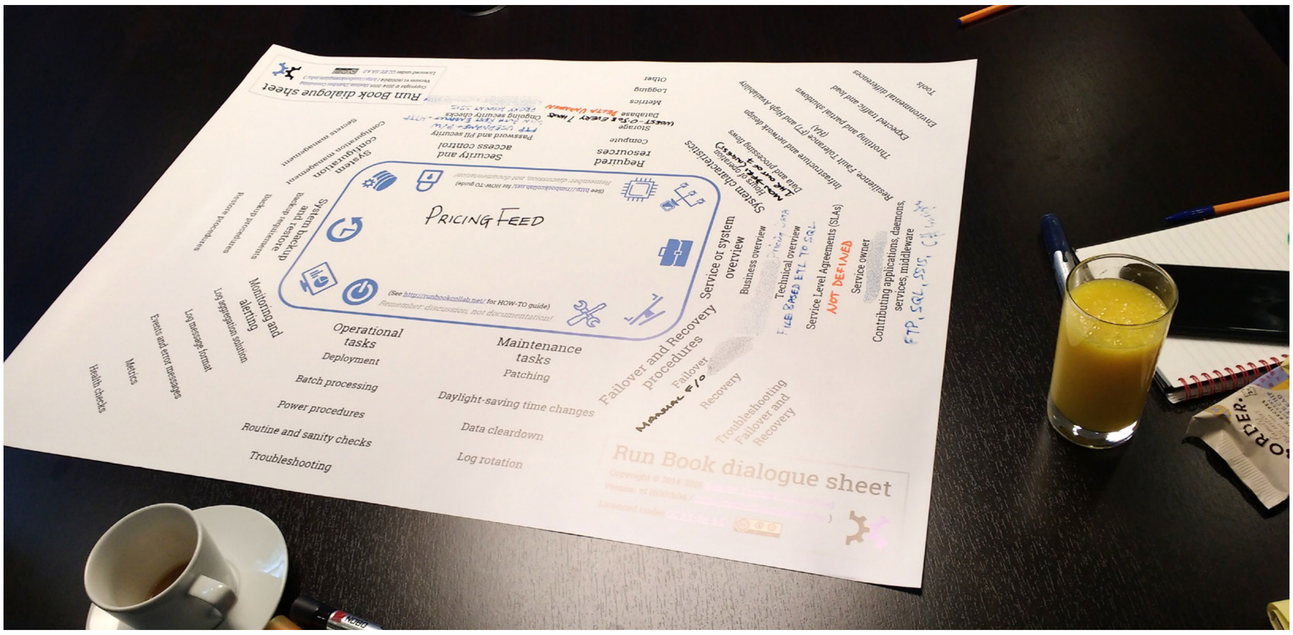
When shipping a parcel, the "interesting" event IDs might be ArrivedAtDepot, InTransit, and Delivered. We define equivalent states for our software that are interesting for all the teams involved. Source: Conflux

similar to the event ID approach.

By defining and collaborating on this set of "interesting" events, teams come to better understand the system they are building and running. No longer is logging "just for errors"; logging leads to vital ongoing insight into the runtime execution of the system.

You log only when you're representing an "interesting" software state, so you're forced to consider why you're logging at a particular point in the code. This in turns avoids what one might call "logarrhea"—too many arbitrary log lines. Combined with a structured logging library, you have a rich source of operational intelligence for our software, validated and curated by teams working with the systems.

Takeaway 1: Use enum-based event IDs with logging to explore system runtime behavior and fault conditions.



The supersized A1 format of the Run Book dialog sheet encourages round-the-table team collaboration on operational aspects. Source: Conflux

2. Use Run Book dialog sheets to identify operational requirements early

Problem 2: Operational aspects not addressed, or addressed too late in the cycle.

Too often, operational aspects of the software system are addressed either late in the process or not at all, leading to problems in the production environment. A technique I have found valuable with many teams is to use a Run Book dialog sheet, a large (A1 size) printed paper sheet with a set of typical operational criteria listed. There's also space for the team to write down answers or questions. The Run Book dialog sheet is licensed under Creative Commons Share Alike, so it's free to use.

The Run Book dialog sheet technique works best when the dev/delivery team takes the lead on defining the initial set of operational features, because the team typically has to reach out to more operations-focused teams to fill in the details. The software design may change at this point to better support operability.

Takeaway 2: Use Run Book dialog sheets to explore and establish operational requirements as a team, around a physical table, together.

3. Collaborate on endpoint health checks

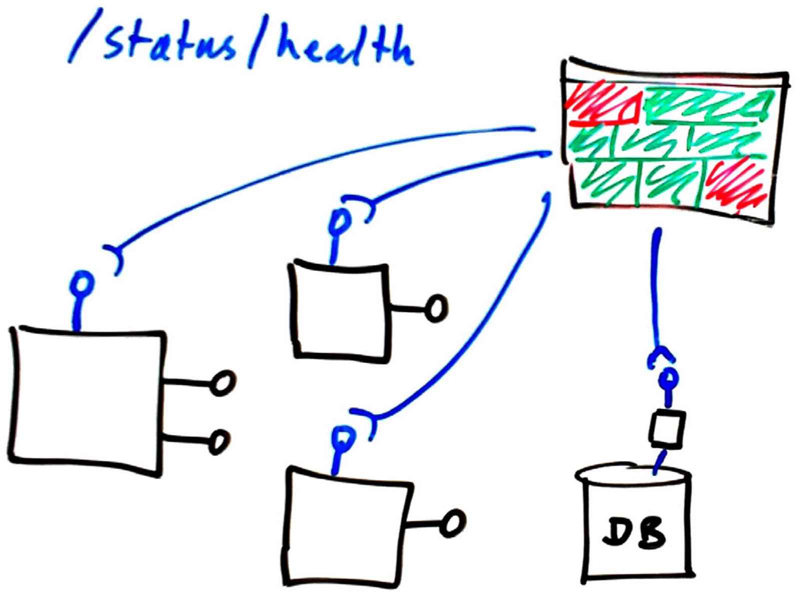
Problem 3: "Why has my deployment failed again?"

Deployment failures are really boring, particularly when they're due to environment misconfigurations. One practical way to remove much of the ambiguity from deployment problems is to use HTTP-based health checks for every component.

For every separate running component or service, you have a health-check endpoint that returns HTTP 200 if the service is healthy and HTTP 500 if the service is unhealthy. You can add more nuanced responses too.

Provide helper endpoints for services such as databases or queues that have no native HTTP capability. This lets you wire up a standard environment dashboard really easily, showing the health of all components at a glance. This technique is particularly powerful when teams collaborate on the conditions for "healthy" and "unhealthy." Why does component X need to see that external service? Why does component Y need four virtual CPU cores or a GPU? You very soon discover interesting runtime dependencies through the process of defining the health-check logic.

Takeaway 3: Use endpoint health checks (with HTTP 200/500 responses) to explore component health conditions.



Provide a small helper service to provide a health check endpoint for a component without native HTTP, such as a SQL database. Source: Conflux

4. Collaborate on correlation IDs for rich transactional tracing

Problem 4: "Which containers/servers handled the request?"

As the number of processing nodes—servers, containers, IoT devices, availability zones—increases, you need to be able to reconstruct a request as an execution trace across multiple nodes. Perhaps one or more nodes are faulty or misconfigured, such as having the wrong version of a container image deployed.

You need to understand exactly where processing delays occur so you can troubleshoot more quickly and resolve the bottlenecks. You can do this using correlation IDs, near-unique identifiers that you inject at the edge of the system and then pass down through downstream components.

Again, if you collaborate among different teams on the trace details, you gain rich operational insights into the running software system. Correlation IDs should help dev teams build better software every week, rather than merely being a special feature of the production environment.

Takeaway 4: Use correlation IDs to trace execution (synchronous and asynchronous) and increase team awareness about system behavior.

5. Use lightweight user personas to capture the needs of testers and ops people

Problem 5: Software is difficult to operate—poor UX for ops.

In your efforts to meet the needs of the primary users of your systems, you sometimes forget to meet the needs of secondary or internal users such as testers, release engineers, and ops people, which is a problem. If your software is difficult to test, difficult to deploy, and difficult to operate, you risk losing money or reputation fighting the software to resolve the problem when there is a failure of some kind.

Addressing the needs of secondary users helps to improve operability. Source: Conflux

You can use lightweight user personas to characterize the needs (motivations, goals, frustrations) of testers, release engineers, ops people, and others who need to interact with the software as part of their job.

Employed well, user personas help to build empathy with other people so you can discover ways in which the software needs to work better in production (or before production). By making software more testable, releasable, and operable, you improve operability overall and make the software more resilient—wins all around.

Takeaway 5: Use lightweight user personas to make sure that the needs of secondary users (ops, testers, etc.) are met during software development.

Create your operability action plan

Focus on operability, and you'll create software systems that work well in production. But to achieve good operability, you must encourage collaboration between different teams by using practical, team-friendly techniques.

[Original: https://techbeacon.com/app-dev-testing/5-proven-operability-techniques-software-teams]

Adapt ITIL to DevOps: continual service transition

Matthew Skelton, Head of Consulting, Conflux

If you work in a large organization with a traditional, sequential release process such as ITIL, the idea of a continuous flow of change may sound unfamiliar. That's been the case at several organizations with which I've consulted recently, so I use the term "continual service transition" as an easier way to explain continuous delivery (CD) to people familiar with ITIL—and you can too.

If your organization has adopted ITIL, you're probably already familiar with the concept of service transition, the process where a service moves from development into live operation.

In a traditional IT environment, various service readiness checks help teams assess and improve software before it goes live. In a continuous delivery world, however, you don't have discrete phases (such as service transition), but you can and should adopt best practices from ITIL around service readiness.

When I discussed new deployment models with an ITIL-trained release manager recently, he responded, "Ah, so we'll always be doing service transition." Exactly right. In a DevOps world, service transition as a separate phase after your software has been built makes little sense.

You need continual service transition, where you're always assessing service readiness and the operability of the changes flowing toward production. Here's how it works.

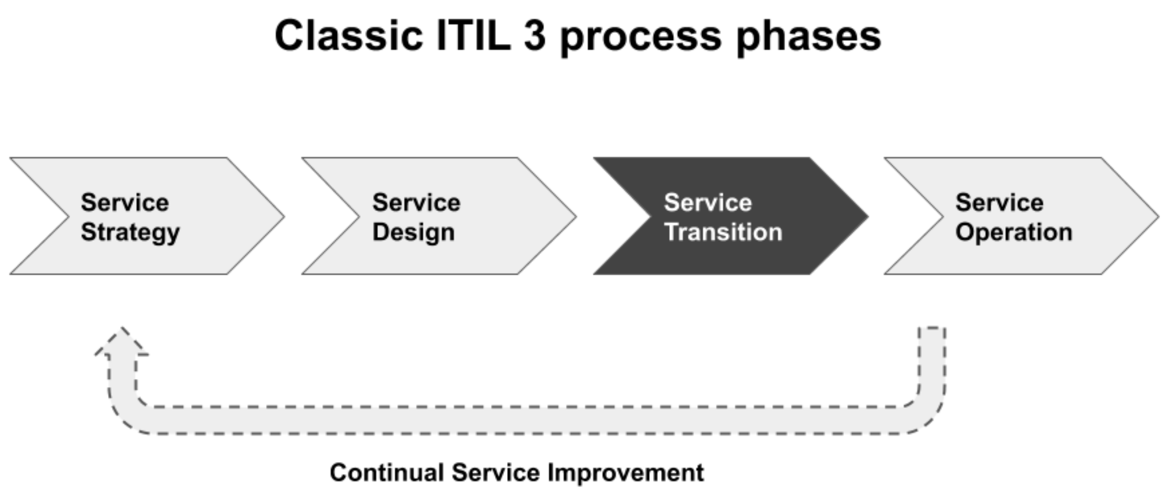


Figure 1: The traditional phases in ITIL 3 with service transition often happen only after the software has been mostly built. (ITIL 4 does aim to address the problems of a phased approach.) Source: Axelos.

The problem with service transition as a phase

The concept and details of service transition are excellent: assess service readiness in many different dimensions (supportability, performance, help desk, end-to-end physical plus digital interactions, etc.). But in a DevOps world, where there is value in making changes every day, a large service transition phase is less useful.

Many organizations assess service readiness and operability properly only during service transition, meaning that there is little time or possibility to fix operational problems.

What's needed in these waterfall-like situations is a way to continually assess service readiness and operability in a way that can meet ITIL standards but also allow for a rapid flow of small updates to software systems.

Axelos, which maintains the ITIL standards, launched ITIL 4 in early 2019 as a way to address the problems with a phased approach. This update focuses on practices rather than processes, which is welcome because practices are more easily adapted to new technologies and new technology dynamics than are processes. However, it will take many years for organizations with existing ITIL expertise to adopt ITIL 4.

How can you help organizations using ITIL move away from a waterfall-like large-phase approach and toward a more continuous flow of change? This is where the concept of continual service transition comes in.

Continual service transition for ITIL practitioners

Since 2014 I have helped many organizations around the world adopt continuous delivery practices. Many of these organizations used some aspects of ITIL to manage the release and change processes, and we had to adapt these to suit a more DevOps-oriented way of working.

For example, I recently worked as engineering lead for a UK government department, covering many teams, locations, and suppliers. As part of championing continuous delivery practices there, I had to find ways to help people with ITIL-specific roles such as release manager and change manager to see things in a different way.

To many people familiar with ITIL, software delivery is all about defining the scope, building something with all the functionality together, and then getting that software into production.

The idea of a continuous flow of small changes can be baffling to people who think in terms of releasing “complete” software. Jez Humble, co-author of the 2010 book *Continuous Delivery*, wrote a great article that covers in detail how to extend and adapt the concept of the ITIL standard change to be more suited to continuous delivery.

The concept of continual service transition seems to help people understand continuous delivery from an ITIL perspective. Service transition is a specific phase in ITIL 3 that occurs after software has been built and needs to be moved into live service.

In a DevOps world, however, you’re releasing small chunks of software whenever they’re ready. This continual flow of change helps you detect problems with the design and implementation, giving you the chance to fix problems quickly.

In this context, you need to apply service-transition principles all the time, assessing the operability of the software and validating service readiness. With DevOps, you have a very tight feedback loop between service operation (software running in production) and service design (software development).

All the key activities that used to happen during service transition in an ITIL 3 model now need to happen continually, for every small chunk of new software heading to production (see Figure 2).

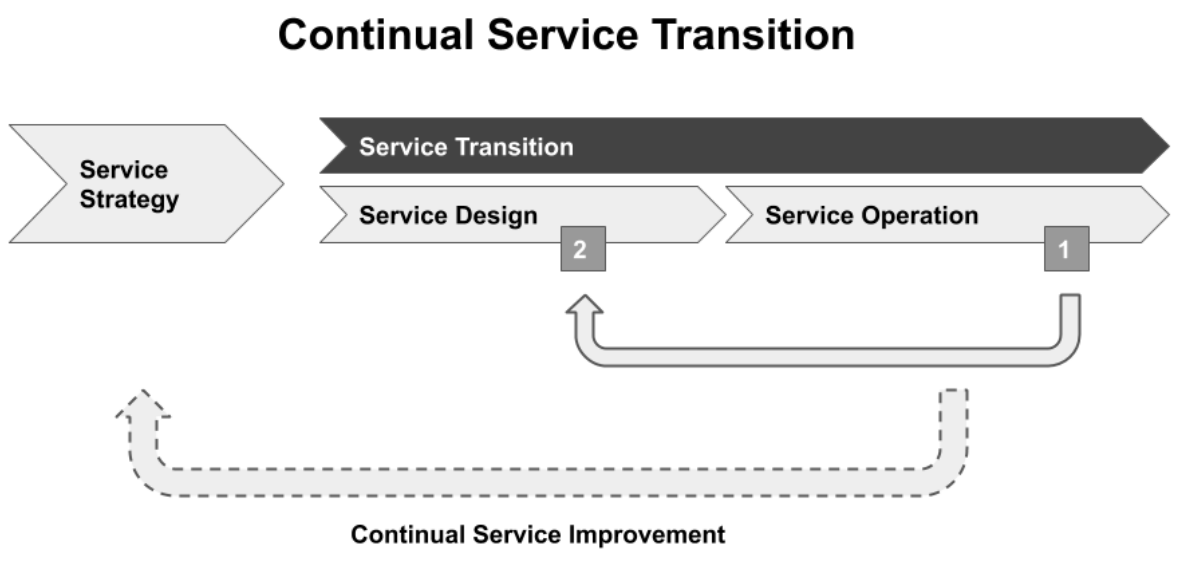


Figure 2: Continual service transition, with (1) feedback from software changes running in production (2) informing and influencing the next software change being written. Source: Conflux.

This continual service transition helps organizations make rapid course corrections based on small, low-risk changes. Co-opting the ITIL language of “service transition” allows people from an ITIL background to more readily grasp the nature of the new DevOps ways of working.

In addition, there are a few practical techniques that work well alongside continual service transition that can help your organization adopt continuous delivery.

Start by deploying a simple readme file

A useful technique for demonstrating the continuous delivery approach is to use a walking skeleton—essentially a bare-bones deployment pipeline that does very little initially.

Begin by deploying a plaintext file, such as readme.txt, to the production environment. The file will not be visible publicly, but the deployment steps will be tracked, logged, and visible to release managers and change managers.

Why start with a readme.txt? It's about minimal risk. The least risky software deployment is a simple text file with some searchable content. This means that from a risk perspective, the deployment is safe and of minimal complexity. Almost anyone can reason about the possible impact of a plaintext file in production.

As part of the deployment, show people the metrics, logs, and dashboards that track the deployment activities, and demonstrate that they have full visibility into changes. When you have won their trust, deploy a simple "Hello, world!" application (slightly more risk, but still understandable). Over time, deploy increasingly larger chunks of software, while bringing other people on the journey.

Design for operator experience

A key part of DevOps is making software easier to operate. Software with poor operability leads to extended service recovery periods, more damaging incidents, and higher burnout in IT operations teams. This is why SRE teams sometimes refuse to partner with development teams that have not invested properly in operability.

A key part of continual service transition (and DevOps in general) is therefore to define and address the experience of service and Ops teams when they're working with software: the operator experience. What do these teams need to see on dashboards? What query tools do they need to find failed transactions or stuck processing jobs?

Use standard user experience (UX) techniques to design a good UX for people running the software in production by treating live service/Ops/SRE people as a user of the software.

To achieve this, you may also need to review the wording of commercial or internal contracts to look for ways to increase responsibility for operational effectiveness in groups that are building the software.

It may be useful to insist that the group or supplier building the software define service readiness criteria, rather than the people running the software in production. Force commercial or contractual responsibility for the operability of the software being written.

Shift your mindset

The focus of service transition in ITIL on operability and service readiness is very valuable. Service transition as a separate phase only after software has been developed does not fit well with a modern DevOps-style incremental delivery of small changes.

Instead, the attention to service readiness and operability should happen all the time—"continual service transition"—with close collaboration among the groups involved.

The mindset shift is quite large, so use some techniques to help: Deploy a plaintext readme.txt file to production, and demonstrate the metrics, logs, and dashboards that track changes.

Explicitly design for the operator experience through the use of UX practices applied to your IT Ops team. And revisit and evolve commercial contracts to help ensure that operability and service readiness are built into the software all the time.

To improve your approaches to service readiness and operability, see The Site Reliability Workbook, from Google, which includes practical tips for SRE implementation based on work with Google's customers, and also the Team Guide to Software Operability, from Conflux Books, which contains practical, team-focused techniques for enhancing operability in modern software.

[Original: https://techbeacon.com/enterprise-it/how-adapt-itil-devops-continual-service-transition]

Group learning for fast flow

Expert-led group learning events with Conflux act as catalysts for lasting change in your organization, helping to adopt fast flow and Team Topologies.

confluxhq.com/learning

conflux

5 ways site reliability engineering transforms IT Ops

Matthew Skelton, Head of Consulting, Conflux

Traditional IT operations do not work at the speed of modern, cloud-native software delivery. That's why new approaches—particularly site reliability engineering (SRE)—are gaining traction across the industry.

But SRE, pioneered by Google, is radically different from IT operations of the past, due to its focus on the error budget, the inter-team relationships brokered by the error budget, the focus on everything as code, and the ability of SRE teams to push back on bad software.

Over the past 10 months I've worked with three organizations (two large and one small-to-midsize enterprise) to reconsider their approach to IT operations for cloud-native software. To do that, the teams explored the SRE model in some detail: the effect on supplier contracts, the dynamics of SRE as a service, the skills gap, and so on.

Here are the ways your large enterprise can take advantage of SRE, and what effect that has on IT operations for both leaders and hands-on managers.

1. Let software engineers design IT Ops

People on SRE teams are either software developers with strong operations knowledge, or IT operations people with strong software development skills. Either way, software is the approach that SRE teams use to solve problems.

If SREs have to undertake the same manual steps to restore service to an application more than a couple of times, they will write software to automate the task. And because SREs understand and practice modern software development techniques, the software they write to fix the problem will not be just a clunky shell script, but well-written software with test scaffolding running in a continuous integration environment.

This software-first approach to IT operations also extends to the role of the development team at times. If the SRE team that looks after a particular application or service finds that it is spending more than 50% of its time doing manual operational work to address problems in the software, the development team must pick up the slack.

This is done, according to Stephen Thorne, customer reliability engineer at Google, by:

- Monitoring the amount of operational work being done by SREs and redirecting excess operational work to the product development teams
- Reassigning bugs and tickets to development managers
- [Re]integrating developers into on-call pager rotations

All the redirection ends when the operational load drops back to 50% or lower.

So if a development team produces software that is too difficult to operate within the 50% balance for the SRE team, the development team must take on the operational tasks and help fix them, learning about operational aspects as needed.

This is a highly disciplined balance between leaning on the skills of SREs and retaining responsibility for the operability of the software within the development team.

2. Rigorously focus on error budget and SLOs

At the heart of the SRE approach is the SLO for the application or service that is being run by the SRE team. The product manager for the service must choose an appropriate SLO that gives her enough margin of possible downtime to cover unforeseen problems while delivering features and updates at a rate that users expect.

Because any service downtime is measured by "neutral" pervasive tooling, there is no dispute about the figures. The SLO approach also drives the adoption of synthetic transaction monitoring, an excellent practice for customer-facing systems. This tests whole customer journeys on a regular basis (usually in 5-to-10-minute increments) from an automated script. This in turn brings the service closer to the customer and, by extension, the dev and SRE teams closer to the customer as well.

As a product manager working with an SRE team, if you are unhappy with the restrictions on deploying new features because you have used all your error budget, what are your options? You can either redefine the SLO to be less available (and therefore possibly have more downtime) or put more effort into operational aspects of the software so that it has better operability and doesn't fail as much. It's a simple choice!

3. Treat IT Ops as a value center, not a cost center

SRE is a high-skill activity, and SRE experts are in short supply; even Google struggles to hire SREs. The unusual mix of deeply technical skills and customer-focused attention to SLO and error budget means that trying to reduce costs for an SRE team is not a wise move.

Enterprises that adopt SRE therefore need to stop treating IT operations as a line item subject to cost reductions. Instead they must treat IT operations as a value center that can help the company avoid downtime and maximize revenue and service availability.

Instead of hiring large numbers of junior, lower-skilled front-line staff, SRE demands that we select high-skilled, experienced, committed staff who will automate their way out of mundane activities. This is analogous to hiring aircraft pilots who have many years of long-distance flight experience rather than junior ground staff. Modern software systems are complicated, expensive machines; why would we hire low-skilled staff to run them?

Thankfully, SRE teams are optional. That's right; not every development team at Google uses SRE. "Downscale the SRE support if your project is shrinking in scale, and finally let your development team own the SRE work if the scale doesn't require SRE support," said Jaana B. Dogan, SRE at Google.

So enterprises can retain a small SRE footprint for critical services, but leave the IT operations of smaller and less proven services to development teams, who are well-placed to support the service they are building because they know it well.

4. Let SRE jump-start cloud-native IT Ops

For enterprises beginning to move to cloud-based platforms and delivery models, the array of options for automation and team responsibilities can be a bit daunting. The range of different ways to do DevOps can be confusing, partly because context makes a huge difference to the effectiveness of these different options.

The case of Poppulo is typical. Damien Daly, head of engineering at the software company, explained why Poppulo created an SRE group: "As we are getting bigger, concentrating our platform development and reliability expertise [in SRE] will allow us to more effectively develop both. Reliability and our platform are first-class concerns and need to be treated with the respect they deserve."

The SRE model presents a clear, specific set of practices and team dynamics that works for large organizations. If you are in an enterprise that needs to move rapidly to cloud-native IT operations from a more traditional setup, then adopting SRE could work well—though only if you adopt it properly and not just rename existing teams.

You may be able to to bypass some of the organizational awkwardness of other delivery models by adopting SRE, but beware of halfhearted implementations that do not set up the required, careful balance of responsibilities.

5. Use managed services to adopt SRE quickly

One way to get the benefits of the SRE discipline quickly, without hiring lots of expensive SRE people, is to use an external provider for SRE. Although SRE was developed and codified at Google using in-house teams, we are beginning to see some emerging SRE-as-a-service offerings from capable outsourced managed service providers.

The SRE-as-a-service model might seem strange at first for IT organizations familiar with collaborative, in-house DevOps approaches to building and running software systems. But, as with many aspects of SRE, if we respect the delicate dynamics involved, then SRE as a service can work well.

The SLO and well-defined standard operating procedures that are at the heart of the SRE approach lend themselves well to a commercial contract. Keep in mind, though, that the details of the commercial contract need to be quite different from typical outsourced IT operations contracts.

We can see this "contract boundary" between dev and SRE in the well-known DevOps team topologies pattern Type 7. (The "DevOps," shown in green, is the collaboration between dev and SRE, not a separate team):

At Google and other large organizations with in-house SRE staff, the "contract" is one of mutual trust around the SLO; for organizations using managed SRE services, the contract will have a commercial element. With a managed SRE service, Russ McKendrick, SRE practice lead at UK-based managed service provider N4Stack, highlights the importance of the SRE team having the authority to say no. "The ability of the SRE team to insist on good operability is a crucial reason for the success of the SRE approach," he said.

This means that a commercially managed SRE contract will include clear terms for the way in which the managed service provider will push back on software that does not work well. In practice, the SRE provider will probably help the dev team improve the operability before releasing to production, possibly through a parallel time-and-materials arrangement.

Another aspect of success with managed SRE is the use of tooling to define

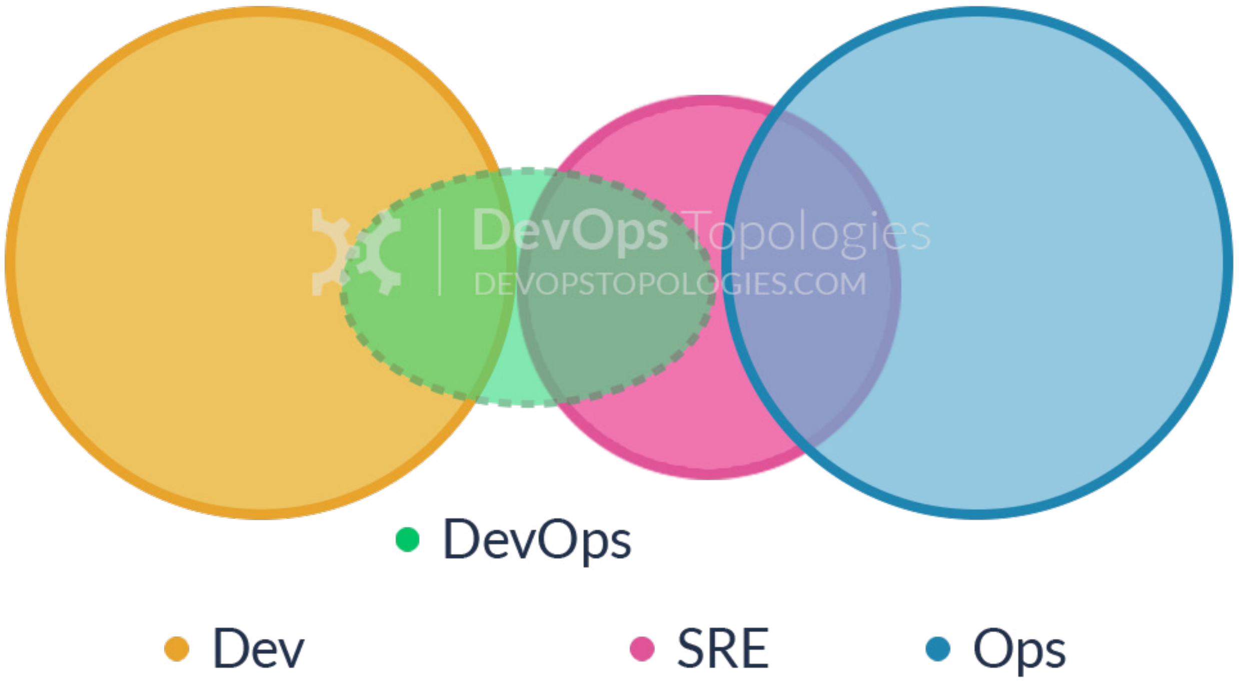


Diagram of SRE responsibilities in relation to dev teams. Image from DevOpsTopologies.com and licensed under CC BY-SA license.

and automate the standard operating procedures needed to keep software running in production. Procedures written in Word or PDF documents are not going to work.

As DevOps luminary Damon Edwards, co-founder of Rundeck, stated in a blog post on operations as a service: "Standardizing procedures helps SREs save time, reduces errors (especially under pressure or when a procedure is critical but run infrequently), and makes it easier to spot anomalies (the outcome is different than expected, or log output is unexpected)."

When adopting a managed SRE approach, you should expect to invest time on an ongoing basis to create and evolve standard operating procedures using a software tool shared with the managed SRE partner (and with the procedures probably stored in version control such as Git).

Engineer-driven

Ben Treynor, vice president of engineering at Google, said SRE is "what happens when you ask a software engineer to design an operations function." This may sound strange to people from an IT operations background, but essentially it means that people on SRE teams have excellent coding skills and—crucially—have a strong drive to automate repetitive ops tasks using code, thereby continually reducing toil.

Google's SRE experts have helpfully written a book, freely available online. In a nutshell, they define SRE as a high-skill operating model for online, high-traffic software services. (Note: The "site" in SRE means website, not geographical or office location.)

It's fair to say that the SRE model balances several metrics and team dynamics—including the following—in a highly effective but rather delicate equilibrium:

- Product dev teams begin by running their own services, including being on call for incidents.
- If and when the service reaches a high-traffic state, the dev team may request support from an SRE to take on running the service in production, leaning on the SRE's reliability and to-scale engineering skills.
- The product owner for the service must define a service-level objective (SLO) based on the downtime deemed acceptable. So, 99.9% availability equates to just 43 minutes of downtime per month, whereas 99.99% availability leaves just 4 minutes of downtime each month.
- The acceptable and available downtime becomes the error budget for the service, which the dev team can spend how it likes. This includes trying out new features, improving operability, etc. But if the service goes down for more than the budgeted time in a month, no new changes are permitted.
- To be permitted to deploy again, the dev team must demonstrate increased reliability through automated operational tests.

This creates a very powerful dynamic for addressing operational problems rapidly and keeping product owners honest about both the required SLO and the operability level in their software service. Any services that must be highly available need huge investments in automation and testing to enable a continual flow of user-visible changes.

Get started with SRE

SRE is a specific approach to IT operations for large-scale, cloud-native software systems. The SRE model sets up a healthy and productive interaction between the development and SRE teams using SLOs and error budgets to balance the speed of new features with whatever work is needed to make the software reliable.

SRE therefore needs quite special skills to succeed, along with strong trust between teams. SRE might be suitable for some enterprises looking to adopt cloud-native approaches quickly, possibly by using an SRE-as-a-service offering from an outsourced provider.

The SRE model is one of several team patterns for modern software delivery explored in the forthcoming book Team Topologies, by Matthew Skelton and Manuel Pais. Follow @TeamTopologies on Twitter for more details.

*[Original:
https://techbeacon.com/enterprise-it/5-ways-site-reliability-engineering-transforms-it-ops]*

SRE in practice: 5 insights from Google's experience

Matthew Skelton, Head of Consulting, Conflux

As site reliability engineering (SRE) becomes more commonplace across IT organizations, what lessons can you learn from Google, one of the originators of SRE? I have recently been helping several organizations understand and adopt SRE practices; as part of this, I spoke to David Ferguson, EMEA lead for customer reliability engineering at Google, to understand how SRE actually works at the firm.

Many organizations face challenges when building reliable services. They often find that just renaming an operations team "SRE" doesn't meaningfully solve their problems. And even if they have staff with SRE skills, they need to create an organizational environment to set them up for success.

Google's SRE guidelines and procedures provide useful insights for other organizations about how they might approach SRE deployment in their own environment. Here are five that your team should pay attention to.

1. A separate SRE team is always optional

One of the most important aspects of SRE at Google is that only some services get SRE involvement. That's correct: SRE is optional. Software development teams cannot assume they will get SRE support for their software.

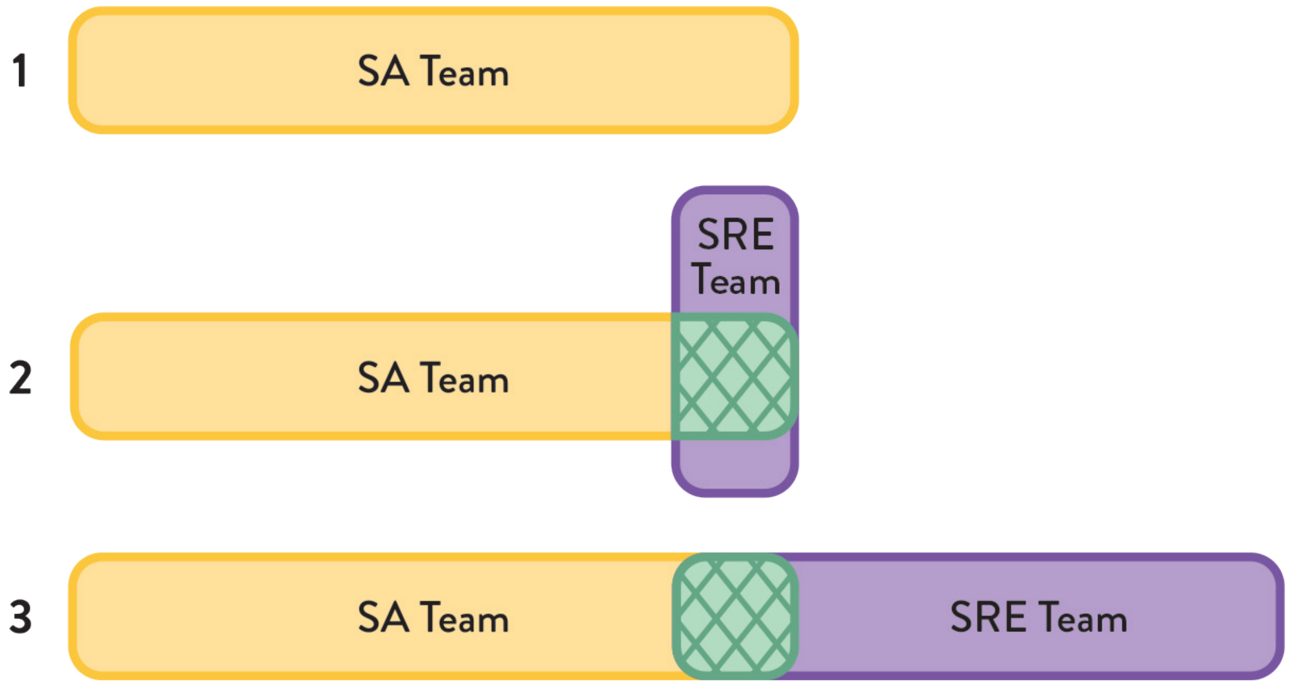
"Make it a privilege to have SRE involvement, not mandatory," Google's Ferguson said.

Think of it like this: A software engineering team should expect to build and run its software entirely by itself, possibly forever. That is the default position at Google.

If the service never reaches sufficient scale to merit SRE involvement, then the software engineering team must continue to build and run the software in production. This is shown as Scenario 1 in the diagram below (where "SA" stands for "stream-aligned," the name for cross-functional software engineering teams we use in the book I co-authored with Manuel Pais, *Team Topologies*).

As a software service begins to scale and needs some operational expertise to help improve its resilience, an SRE team might help the stream-aligned software engineering team to understand scaling, reliability, observability, and how to use platform components that it manages. This is Scenario 2 in the diagram below. However, at this point, the SRE team is not yet running the software in production.

Finally, if the stream-aligned software engineering team can persuade the SRE team that its software is operationally ready, the SRE team might choose to partner with it to run the software in production, ensuring reliability as the



Scenario 1 shows the default position of a stream-aligned (SA), cross-functional software engineering team building and running software by itself. Scenario 2 shows an SRE team helping the stream-aligned software engineering team to understand operability better. Scenario 3 shows the SRE team taking responsibility for running the software in production. Source: *Team Topologies*, by Matthew Skelton and Manuel Pais.

software scales. (This is Scenario 3 below.) Software engineering teams must demonstrate a high degree of operability and service readiness before an SRE team will partner with them to help improve the software's reliability. SRE teams are free to turn down a request for help from a software engineering team if they think the operational burden would be too high or if there are no clear opportunities for engineering projects.

Is your SRE team empowered to do that?

2. Software product teams must care about operability from day one

A software engineering team must demonstrate that its software is ready for production by continuously focusing on operability. Only by showing that the software is operationally ready will an SRE team be persuaded to partner with the software engineering team to help with reliability as the software scales.

As part of this focus on operability, it's important to make sure that organizational incentives (including pay and career progression) help to drive the right behaviors.

At Google, "we incentivize software engineers to get things into and used in production" and not just checked in and passing tests, Ferguson said. The company also rewards SREs when they "actually identify and do engineering work on services," he said.

And at Google, software engineers still need to have some contact with production—through looking after things in early development, business hours on call, and in other ways.

With this approach, Google avoids a "hard boundary" between software engineering and SRE teams. Software engineers have incentives to understand production, and SREs have incentives to understand the business context of the software, thus encouraging shared ownership.

3. You can achieve many of the benefits of the SRE approach without a separate SRE team

A key part of SRE-based discipline is the use of the error budget to control when to focus on improving reliability and operability. But the error budget model can work without a separate SRE team. It just needs discipline from the software team and product owner to play by the rules of the error budget and stop deploying new features when service availability has been breached for that month.

Google's Ferguson called the error budget mechanism "an appropriate control process for agile teams" that "want to go fast." He describes the error budget discipline like this:

- Define what matters to your users.
- Measure it and define the guardrails that you care about.
- Decide what you are going to do when you hit the guardrails.
- When you hit the guardrails, actually do the thing you said you would.
- Be transparent with your data and your actions.

When outlined in this way, it's clear that a single stream-aligned software team can be empowered to enact error budgets without needing a separate SRE team. What you're doing here is keeping a laser-like focus on what matters to the end user and making sure that you know when the end-user experience is starting to degrade.

4. Choose business-relevant service-level objectives

At the heart of the SRE approach is the service-level objective (SLO) for the application or service that is being run by the SRE team. An SLO is a performance or availability target for that service, a degree of performance or availability that meets business expectations at an acceptable cost.

Conformance to SLOs is measured through the use of a service-level indicator (SLI), or perhaps several SLIs. An SLI is a single quantitative measure of some aspect of the behavior or performance of a service or system. SLIs are generally closely tied to characteristics that users of the service care about:

response time (for web applications), durability (for data persistence), error rate, or perhaps the availability of a multi-step flow.

Synthetic transaction monitoring is good to have in place because the monitoring is driven from external locations, generating a similar experience to that seen by end users. However, teams at Google often go beyond synthetic monitoring, as Ferguson explains. "We are actually scoring each and every interaction with the service."

Synthetic monitoring is useful because it represents an expected load on the system but rarely covers the full breadth of interactions that matter. This is particularly true with horizontally scaled components, where the synthetics may tend to probe only one instance of each clustered service.

This focus on the ways in which users experience the running services and applications is a key aspect of the SRE approach. Instead of simply monitoring uptime (whether a process is running or a webpage is present), SLOs driven by SLIs encourage attention to the quality of the interaction experienced by the user. Ultimately, quality is one of the most important criteria for successful software.

5. Reliability is about more than just avoiding downtime

If a service is down, it's fairly straightforward to understand the impact on users: They cannot use the system at all. However, end users can be affected by services performing slowly or intermittently, or in unusual ways at certain times. These more nuanced aspects of reliability are crucial to measure as part of SRE.

"Our approach isn't just about hard downtime," Ferguson said. An SLO of 99.9% over a month means that around 1 in a 1,000 requests fail (or go too slow) over that period.

The Google SRE approach weights in the real-time aspects of reliability, such as temporary slowness during garbage collection or peak time. It also biases toward busy-hour; if 10% of your traffic is at busy hour, then you will burn error budget much faster if you have an outage during that time.

It also makes it clearer that everyone has a part to play. When you focus just on outages, it tends to look like operations fault and we can convince ourselves that it can't/won't happen again. When you look at request-by-request performance, you'll quickly see that you never really have 100%, and you don't want to spend the time or money getting to 100%

Modern large-scale software needs high-quality metrics on performance—covering request/response time, latency, throughput, variability, outliers, data persistence, and more. These things all contribute to the reliability of the software as seen by end users, so we need to measure and understand all these dimensions.

Ferguson said, "Done right, [with SRE] we are just helping the organization keep to the promises/decisions it made about how good it wanted its products to be."

Leverage the underlying dynamics of SRE, not just the name

The SRE approach can clearly be a key part of success with large-scale cloud software. However, simply adding a separate SRE team misunderstands how Google implements the concept. In fact, SRE teams are optional at Google and software engineering teams must work hard to persuade SRE teams that their software has good operability.

"If you don't care about your reliability, they shouldn't have to, either," Ferguson said.

So, keep SRE teams as a privilege for the most deserving services, define your error budget and use that as a control mechanism for software teams, keep a relentless focus on the operability of the software you're building, and choose SLIs and SLOs wisely to make sure that you're measuring what actually users really care about.

To improve your approaches to SRE and operability, see The Site Reliability Workbook from Google*, which includes practical tips on SRE implementation based on work with Google's customers, and the* Team Guide to Software Operability *from Conflux Books, which contains practical, team-focused techniques for enhancing operability in modern software.*

[Original: https://techbeacon.com/enterprise-it/sre-practice-5-insights-googles-experience]

Transformation for fast flow

Conflux works with key groups in your organization over six to 18 months to increase awareness and capabilities around fast flow and Team Topologies.

confluxhq.com/transform

conflux

About the author

Matthew Skelton is the co-author of the books *Team Topologies: organizing business and technology teams for fast flow.* Head of Consulting at Conflux (confluxhq.com), he specialises in Continuous Delivery, operability, and organization dynamics for software in manufacturing, ecommerce, and online services.

Recognised by TechBeacon in 2018, 2019 and 2020 as one of the top 100 people to follow in DevOps, Matthew curates the well-known DevOps topologies patterns at devopstopologies.com. and is co-author of the books *Team Topologies* (IT Revolution Press, 2019), *Team Guide to Software Operability* (Skelton Thatcher Publications, 2016), and *Continuous Delivery with Windows and .NET* (O'Reilly, 2016), along with several key reports on SRE.

Matthew founded Conflux in 2017 to offer training and consulting to organizations building and running software systems.

Twitter: @matthewpskelton | LinkedIn: linkedin.com/in/matthewskelton/

About Conflux

At Conflux we help organizations to adopt and sustain proven, modern practices for delivering software rapidly and safely using consulting, training, and our own range of books. We specialize in applying Continuous Delivery, software operability, and team-first organization design using Team Topologies across organizations of all sizes, from startups to multinational corporations.

Led by well-known consultant, speaker, trainer, and author Matthew Skelton, Conflux brings a holistic approach to sustainable software delivery for all organizations.

confluxhq.com

conflux

consulting + training + books for effective software delivery

confluxhq.com