



DevOps in Practice

Christof Ebert and Lorin Hochstein

From the Editor

DevOps connects development, delivery, and operations and, thus, facilitates a fluid collaboration of these traditionally separated silos. As an agile method, DevOps is today used across industries and is not limited anymore to IT services and specific technologies. Lorin Hochstein and I present a brief overview on DevOps best practice. A Netflix case study emphasizes DevOps in practice. I look forward to hearing from you about this column and the technologies that matter most for your work.—*Christof Ebert*

DevOps IS THE culture, process, and technology to foster close collaboration of software development and IT operations.^{1,2,3} It connects different disciplines to accelerate software development and deployment. It combines software development (Dev) and system administrators (Ops) but also quality assurance and the user community through aligned processes and software tools. It is intended to improve software quality, the speed of development and delivery, and the cooperation among the teams involved.

While DevOps needs heavy technology changes to facilitate the seamless flow of software, service

requests, and the necessary development and operations artifacts, it is a paradigm shift above all.¹ Instead of distributed silo-like functions, it establishes cross-functional teams to work on continuous operational feature deliveries. This integrative approach helps teams deliver value in a faster and continuous way, reducing problems generated by miscommunication among team members and enhancing a faster resolution of problems.

The generic process is indicated in Figure 1. Its promise and goal are to better integrate the development, production, and operations business processes with adequate technology—thus, not staying on highly artificial process concepts that will never fly but, rather, setting up a

continuous delivery (CD) process with small upgrades. Companies such as Amazon, Google, and Netflix (see “DevOps at Netflix”) have streamlined this approach, achieving cycle times of minutes. This obviously depends on the deployment model; for example, a single cloud service is easier to facilitate than actual software deliveries to embedded systems.

DevOps Technology Learnings

DevOps applies very different delivery models and must be carefully tailored to the environment and product architecture. DevOps does not mean a delivery every few seconds, as is possible in centralized IT systems, such as the Google or Amazon server farms. DevOps can also be tailored

Digital Object Identifier 10.1109/MS.2022.3213285
Date of current version: 23 December 2022

to critical systems, as we have managed for different companies. Even in such critical systems, upgrades can be planned and delivered in a

the necessary documentation must be carefully updated. With semi-automatic verification and validation tool chain and document

which are swapped to active mode after in-depth security and verification approaches.

Being aware of fast deployment quality risks, DevOps must consider cybersecurity. Combining security with DevOps toward DevSecOps provides a path to efficiently implement cybersecurity. DevSecOps is an organizational software engineering practice that aims at unifying software development (Dev), security (Sec), and operations (Ops). DevSecOps allows faster and more secure software delivery with consistent governance and control, such as automated security testing in each mini cycle. The main characteristic of DevSecOps is to automate, monitor, and apply security at all phases of the software lifecycle.

While DevOps is an engineering paradigm that applies for the entire lifecycle, DevSecOps benefits can be best shown when it comes to security testing. The benefits are obvious, not only with the earlier

DevOps is intended to improve software quality, the speed of development and delivery, and the cooperation among the teams involved.

fast and reliable scheme, as the recent evolution of automotive software over-the-air upgrades shows.

DevOps for embedded systems is more challenging than cloud and IT services due to the dependence on legacy code and architecture as well as trying to fit it into a CD approach. For instance, in safety-critical systems, the safety case with

management, this is feasible in a relatively short time compared to traditional one-week cycles. Aside from the highly secured cloud-based delivery model, such delivery models also need dedicated architecture and hardware changes—for instance, a hot-swap controller concept, where one half is operational, and the other half builds the next updates,

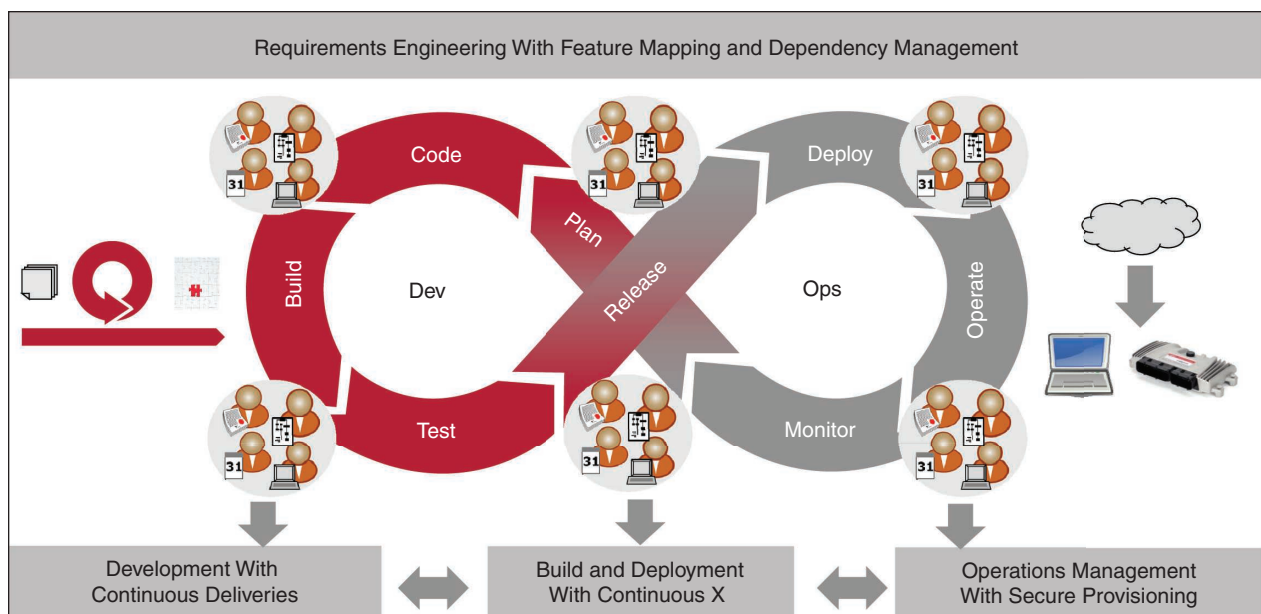


FIGURE 1. A generic DevOps process.

detection of variabilities. More important is that it breaks down walls and enables hypercollaboration among teams. DevSecOps is a key ingredient for any software company to combine speed with high cybersecurity. Having introduced it in different environments, we see benefits in the achieved cycle time and efficiency due to less wait time after a software update.

A warning here: optimizing for velocity is dangerous, as it increases technical debt. Many software and IT companies outsource and distribute their development activities. This can be done with DevOps yet should avoid fragmentation. Often, we see companies with ticket-based requirements and change management, which drives a highly fragmented and isolated development approach. A designer would take the ticket, implement, test, and forward to the operations and release teams. As developers are paid by finished ticket, those tickets get increasingly fragmented to achieve a higher throughput. There is time neither for impact analysis nor for clean documentation, such as traceability and regression testing of the full environment. A subsequent change done by another developer might cause unwanted side effects due to not being aware of the previous change and its impacts.

While we aspire to the value of “loosely coupled, highly aligned,” we sometimes joke about being “loosely coupled, loosely aligned.” The system is effectively optimized for high velocity for individual teams, but the resulting tradeoff is that it is more difficult to execute on initiatives that involve work across multiple teams. Consider the challenge of migrating all of the service teams onto newer technologies (e.g.,

migrating applications that use the previous generation internal Guice-based Java libraries onto the newer Spring Boot-based platform). Often, the motivation for doing these migrations is that they will unlock a new capability, but that advantage will only be realized when the entire fleet of apps has been migrated. However, the service teams will not realize any immediate value by migrating to the new technology, and it may even break them. This means that doing the migration from their perspective has a downside (engi-

neering effort and risk) but no near-term upside. As a result, migrations can sometimes take very long periods of time to complete.

The distributed nature of the system also makes it effectively impossible for engineers to reason about the broader impact of a change. For instance, Netflix offers a single service to its customers; any change in the system could potentially affect service behavior. Furthermore, the system is designed to enable change without coordination, which makes fault localization more difficult when a change made by team A results in an operational impact to a service owned by team B, who did not even know about the change. For example, a change in one part of the system can lead to unforeseen changes in traffic patterns

in a completely different part of the system. The paradigm “You build it, you run it” is dangerous, as it results in a large variation in operational expertise across the organization. Typically, engineers on the more critical service teams have more operational skills because their services are more likely to be involved in an incident if there is a problem, which means they have more experience dealing with operational issues. However, teams whose services are not typically involved in incidents will necessarily

The paradigm “You build it, you run it” is dangerous, as it results in a large variation in operational expertise across the organization.

have less experience doing operational work. This means that, in the rarer cases where their service is somehow involved in an incident, it can be more difficult to diagnose and remediate.

Even identifying the criticality of a service is a challenge. For instance, not all services in the Netflix architecture are critical for users being able to watch video; some of them provide value, but the system can degrade gracefully if one of these fails. For example, there is a “bookmarks” service, which keeps track of what point in time a user stopped watching a video so they can resume from that point in time. If this service fails, resulting in us not being able to retrieve the time where they previously were, we can fall back to starting at the beginning of the video.

DevOps AT NETFLIX

While Netflix engineers don't use the term *DevOps* internally when describing the interaction of development and operations processes, Netflix's actual practices for bringing code into production are very DevOps-y. The success of the model is due to multiple factors, including architecture, culture, tooling, and skill. Figure S1 shows the interaction of these different factors and what it means for the technology success of DevOps.

ARCHITECTURE: MICROSERVICES

Netflix was one of the early adopters of what today is known as a *microservice architecture*. The Netflix control plane is implemented as a large collection of services. When a device makes a request against the control plane, that request may trigger requests to many different services. A visualization of a microservice architecture is shown in Figure S2. (Note that the labels are meaningless in this diagram.) Each bubble represents a microservice, except for the leftmost bubble, which represents the source of all of the requests and is labeled "Internet." All requests flow from left to right, starting from the "Internet" bubble. The initial services that

receive traffic are colloquially referred to as the "edge" layer, doing work such as authentication and routing to other microservices. The rightmost services (the ones that do not call other services) are typically the "persistence" layer: the databases. The collection of other services is generally referred to as *midtier*.

Each service in the control plane is owned by exactly one team, although some teams may own multiple services. A critically important property of this architecture is that these services can be deployed independently of each other. This means that a team can push out a new version of their service without having to coordinate with other teams. This reduction in coupling across teams increases the deployment velocity of the individual teams.

CULTURE: YOU BUILD IT, YOU RUN IT

Netflix's microservice architecture enables the operational model used in the company: you build it, you run it. Development does not hand off code to operations to deploy the code. Instead, the software engineers who write the code are also responsible for deploying the code into production as well as operating the code while it is in production. The software engi-

neers are in an on-call rotation, which means they are the ones who will be paged in the middle of the night if something goes wrong with the service that is impacting customers.

Development is incentivized to deliver features, and operations is incentivized to keep the system up and running. Delivering features requires making changes to production, but every change can potentially lead to an outage. When development and operations are separate agents, this velocity–stability tradeoff has to be resolved by a negotiation mechanism between the agents. However, when the same agent is responsible for both velocity and stability, then the engineers can navigate this tradeoff by applying their own judgment based on the context, the sort of work that is at the heart of all engineering. It also creates an incentive

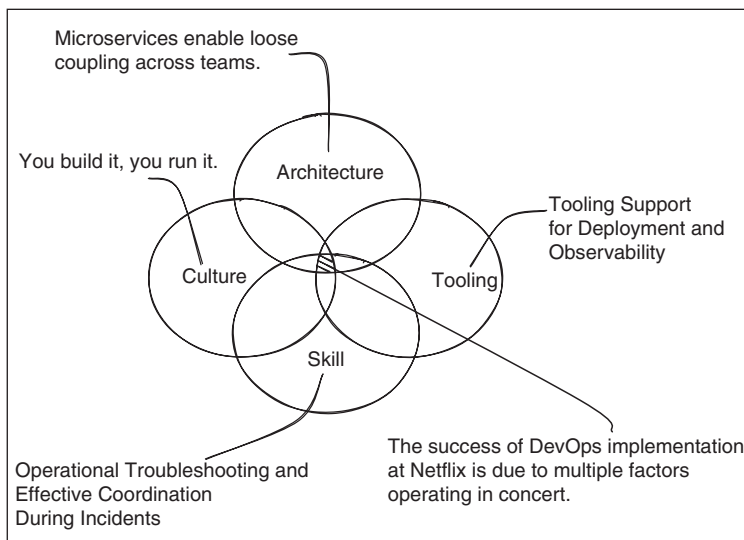


FIGURE S1. The combination of architecture, culture, tooling, and skill are necessary for the success of the operational model at Netflix.

(Continued on next page)

DevOps AT NETFLIX (CONT.)

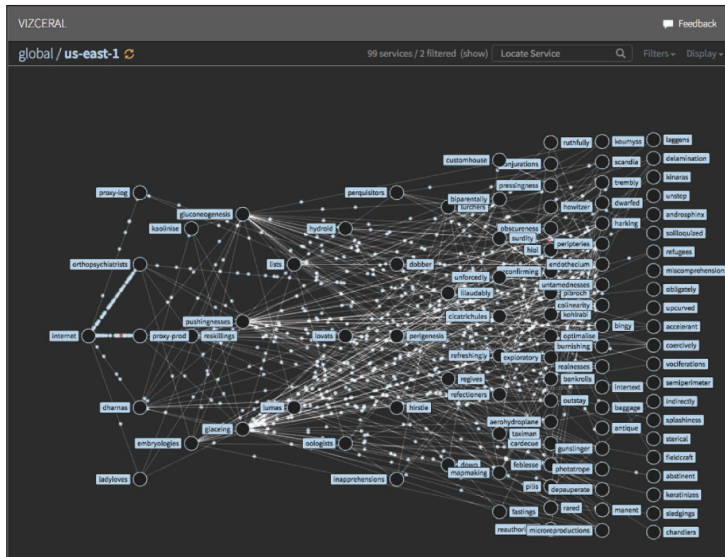


FIGURE S2. A microservice architecture visualization. (Source: <https://github.com/Netflix/vizceral/blob/master/vizceral-example.png>; used with permission.)

for the software engineers to build more operable software because they are the ones who directly feel the pain of operations when something goes wrong.

TOOLING: PLATFORM ENGINEERING PROVIDES THE “PAVED ROAD”

The role of the platform engineering organization is to enable the software engineers to effectively develop, deliver, and operate their software by providing tooling. This tooling is generally referred to as the *paved road*. Netflix has a “freedom and responsibility” culture: engineers have considerable autonomy with regard to the technical choices they make, but, if they choose technologies that are not supported by centralized teams, then they must shoulder the burden for any difficulties that arise due to using unsupported tech. Engineers are encouraged to use the paved road and will get full support from the platform engineering organization if they do.

From a DevOps perspective, the two key technology areas are

- delivery
- operations.

DELIVERY TOOLING

Figure S3 depicts the continuous integration (CI)/CD tooling that Netflix engineers use to go from source code to production. In particular, Netflix has separate tooling for doing CI and CD.

Netflix uses the open source Jenkins system for providing CI functionality: Netflix engineers write CI scripts, and Jenkins workers check out code from the SCM system and run these scripts. Netflix has historically been a Java shop, and the open source Gradle is the build tool of choice inside Netflix. Netflix engineers curate a set of open source Gradle plug-ins, called *Nebula* (Netflix Build Language), to add additional build functionality to Gradle. Build scripts typically either publish a build artifact (e.g., a Java client library JAR

file, Debian package, or NPM package) to the internal artifact repository or publish a container image to the container repository.

Netflix uses the open source Spinnaker system for providing CD functionality. Because Netflix was an early adopter of cloud computing (starting around 2008), much of the platform tooling that it needed did not exist at the time and so had to be built in house. For example, Netflix’s first-generation delivery system, Asgard, and its second-generation delivery system, Spinnaker, were both developed in house and were both released as open source. Spinnaker enables engineers to specify their deployment pipelines using a web interface to control how their application code gets delivered to production. Work is currently underway on a third-generation delivery system that focuses on a declarative, as-code approach to describing delivery workflows.

Historically, Netflix deployed services such as virtual machines on top of AWS’s EC2 platform. Several years ago, Netflix also added support for deploying services as containers. Netflix uses an internally developed

(Continued on next page)

DevOps AT NETFLIX (CONT.)

container platform called Titus, which has been released as open source. Titus is currently implemented on top of Kubernetes and also uses EC2 as the underlying compute platform.

OPERATIONS TOOLING

To operate a service effectively, engineers must have insight into how the service is behaving and whether any remediation actions are required. Historically, Netflix invested heavily in a telemetry-based approach to observability: developing an in-house telemetry platform called *Atlas* (which was also released as open source). Leveraging *atlas* is a self-service dashboarding platform called *Lumen* for displaying collections of *Atlas* graphs and a system for creating alerts-based *Atlas* data, so engineers can be notified when they need to take action. More

recently, there has been increasing investment in developing better tooling around inspecting logs and traces.

SKILLS: OPERATIONAL EXPERTISE

Giving a software engineer a pager and access to observability tools does not magically bestow operational expertise upon that software engineer. The skill set for operating a service is distinct from the skill set for developing a service. Addressing operational issues in production, such as dynamic fault management, is qualitatively different from debugging. This type of work involves time pressure, uncertainty, and risk tradeoffs. The operator must take actions to ensure that the service gets back to healthy. The system behavior can change with time (things can get worse!), and potential remediation actions carry the

risk of making the problem worse. Many operational failure modes are related to services becoming overloaded (e.g., using too much CPU or memory) after a sudden unexpected change in traffic patterns.

Engineers must also work together effectively during an incident to remediate. Because of the distributed nature of the system, any incidents require engineers working together to make sense of the problem.

On the author's team, to help improve the operational skills on the team, we have a weekly standing meeting, "This Week in Operations", where we reflect on the operational surprises that occurred in the past week and how the responders dealt with them. This enables us to learn from the experiences of others.

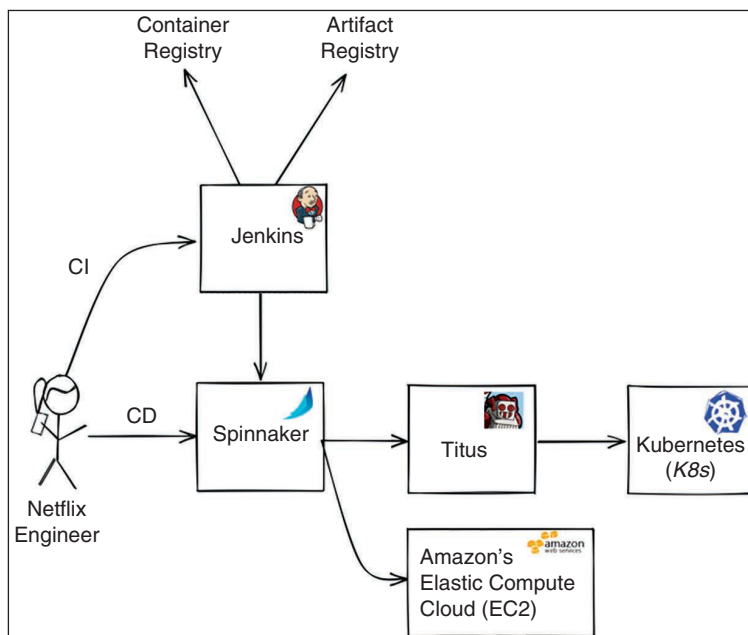


FIGURE S3. The CI/CD toolchain used by Netflix.

In principle, we can spend more of our reliability efforts on the services that are critical for streaming for which we have no meaningful fallback behavior. However, just identifying these services turns out to be surprisingly challenging. One challenge is simply defining what functionality should be considered critical: there's an inevitable tradeoff between an intuitively satisfying definition that is not operational and an operational definition that is not intuitively satisfying, leading to endless discussions. We once worked on an ultimately failed project called “service tiers” that attempted to validate empirically whether an assumed set of critical services was the correct set by running end-to-end tests where we would inject failure into all of the noncritical services for the synthetic client used for the testing. The results of such tests were often ambiguous. When one of these tests failed, trying to figure out why it failed was extremely difficult, and, in the end, the conclusion was that it simply wasn't worth the engineering effort to do the troubleshooting given the other priorities of the organization.

When embracing a DevOps culture, developers take a full-stack developer approach and must take responsibility of the testing and release environment. Developers need to master an extended skill set beyond code knowledge, including DBA and testing. Furthermore, as the boundaries of functional silos blur, a more intense collaboration with other team members is required. Test is a critical part of the development process, and test-driven development and continuous integration (CI) are testing practices performed by the development team. In this scenario, testers can be paired with

developers, and both can gain technical knowledge. The quality assurance responsibility is to ensure that all test cases are automated and that full code coverage is achieved.

Mastering the DevOps Transformation?

At Vector, we have supported several companies in improving efficiency with DevOps and CD. A key learning in all of these cases is that the culture shift should not be underestimated. Here is some guidance that we apply in all DevOps projects:

- Ensure that top management understands the challenge and is willing to support. Software is not just another item to manage—software is of a different nature. “Try harder, work faster” won't work to push the transition.
- Set up a powerful, empowered, and competent central software organization to define and drive appropriate methods such as agile development, DevOps, feature-based development, safety,

A warning here: optimizing for velocity is dangerous, as it increases technical debt.

- Break complex architectures and feature sets toward small chunks that can be produced and deployed independently.
- Maintain a configuration and build environment that always provides visibility about what is currently deployed with which versions and dependencies.
- Introduce a purpose-built development and production environment from legacy ALM/PLM environments.
- Bridge the traditional silo-type cultures of development (perceived by operations, in its thoroughness, as cumbersome and expensive) and operations (perceived by developers as quick and dirty).

security, and tools such as workflow management, configuration management, and continuous X.

- Identify standards and existing software as building blocks you want to use. Reuse is the key. Concentrate on the glue and differentiating features.
- Identify partners to support your tour to the summit. Choose your partners carefully—you will have to accept a certain level of dependence in binding to long-term partners.

We recommend an agile approach to the DevOps transformation.¹ Define a stepwise approach and an appropriate road map. Go for an evolution rather than a revolution.

Start with pilot projects that allow failures and delays. Learn from failures and move on. For Vector, this is an endless journey, and that is also what we see with all other successful software companies.

From an operations perspective, a DevOps practice has a great impact on culture and discipline. With DevOps, an operations team must continuously connect to other functions without losing control. Therefore, a close collaboration between IT operations and development is needed to achieve the continuous process promoted by the DevOps team. Moreover, infrastructure monitoring and application performance management are more important in this approach, and a fluid communication with other team members is needed.

DevOps is a paradigm shift impacting the entire software and IT industry. Building upon lean and agile practices, DevOps means end-to-end automation in software development and delivery. Hardly anybody will be able to approach it with a cookbook-style approach, but most will benefit from better connecting the previously isolated silos of development and operations. Close collaboration from requirements onward to maintenance, service, and product evolution will yield, typically, a cycle time improvement of up to 50% (with centralized IT systems having higher savings than distributed embedded product IT) and a cost reduction of up to 20% due to less effort and reduced rework. Major

ABOUT THE AUTHORS




CHRISTOF EBERT is the managing director of Vector Consulting Services, Stuttgart, 70499, Germany. Contact him at christof.ebert@vector.com or <https://twitter.com/christofebert>.



LORIN HOCHSTEIN is a senior software engineer at Netflix in Los Gatos, CA 95032 USA, where he works on continuous delivery tooling. Contact him at lhochstein@netflix.com or <https://lorinhochstein.org>.

drivers are less impact of requirements changes, focused testing and quality assurance, and much faster delivery cycle with feature-driven teams.

Albert Einstein, the ingenuous mastermind, once remarked, “All means prove but a blunt instrument, if they have not behind them a living spirit.” This certainly applies to DevOps, as there is no out-of-the-box solution. DevOps needs experience and tailoring to orchestrate the instruments toward value in frequent software deliveries without accumulating the problems of tomorrow, such as technical debt. As products and lifecycle processes vary, each company needs its own approach toward a DevOps environment, from architecture to tools and culture. 

References

1. C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, “DevOps,” *IEEE Softw.*, vol. 33, no. 3, pp. 94–100, May/Jun. 2016, doi: 10.1109/MS.2016.68.
2. G. Kim, P. Debois, J. Willis, and J. Humble, *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. Portland, OR, USA: IT Revolution Press, 2021.
3. M. Gokarna and R. Singh, “DevOps: A historical review and future works,” in *Proc. 2021 Int. Conf. Comput., Commun., Intell. Syst. (ICCCIS)*, pp. 366–371, doi: 10.1109/ICCCIS51004.2021.9397235.