The DevOps Handbook Feb 24, 2021

"In previous economic eras, business created value by moving atoms. Now they create value by moving bits."
- Jeffrey Snover, Technical Fellow at Microsoft

Modern society requires all forms of engineering to work in a multidisciplinary way. Cross-functional teams with diverse viewpoints and broad expertise collaborate. They test their hypotheses in continuous experimentation and let changes flow into production smoothly, without any disruptions for any internal or external customers. Quality Assurance, Operations and InfoSec work on ways to reduce friction for teams. This results in a safe system of work, where teams quickly and independently develop, test and deploy value in a safe, secure and reliable way. It maximizes productivity and enables organizational learning, leads to a higher level of employee satisfaction and makes the company win in the marketplace. This is the target state of an organization applying DevOps.

This target state is achieved only by few organizations in reality. Because there is an inherent goal conflict between development - responding to changes in the competitive landscape rapidly - and operations - providing reliable, stable and secure services, local optimizations within those "silos" occur. This does not contribute to the overall optimization of the organization. It may lead the organization into a downward spiral that in the wort case might even put the existance of the organization at stake.

The goal of DevOps - counteracting this downward spiral - is to enable teams towards bringing planned changes into production quickly. Fast feedback loops present everyone with the immediate effect of their actions. Automated testing ensures the quality of the products. And pervasive production telemetry supports relentless experimentation in the field. Dark launches and feature toggles help decoupling deployment and release for controlled, predictable, reversible and low-stress releases.

"Because we value everyone's time, we do not spend years building features our customers don't want to use, deploying code that does not work or fixing something that isn't actually the cause of our problem."

DevOps culture rewards people for taking risks and keeps successful product teams together. To improve resilience of the systems in the field, teams inject faults into production environments. Everyone has ownership of their work.

There are several metrics that can be used to measure DevOps culture, including throughput, code changes and deployments, lead time or reliability. These metrics show that high performers are more agile and exceed profitability and market share. And they produce more reliable products at the same time.

The Foundations of DevOps

Feb 25, 2021

The DevOps movement has been influenced by various independent developments from manufacturing and from IT. Among those are Lean, Agile, Continuous Delivery and Toyota Kata. The foundations of DevOps are built around one fundamental concept of Lean – the value stream. In manufacturing, a value stream flows from a request to its delivery. In IT, this concept can be applied to the flow from Development to Operations to the customer.

Principles already known from manufacturing can be applied for optimizing the IT value stream as well. Metrics can be defined such as lead time – the time from request to delivery – processing time – the time from taking up work to delivery o- or %CA – the percentage of work that is directly processable by the next downstream process. For DevOps, the deployment lead time is of particular interest. This is the time from a check-in of a change to its delivery into production.

To optimize the value stream, DevOps suggests three ways. The first way is about getting to fast flow from left to right, i.e. from Dev to Ops to the customer. Techniques such as small batch sizes, reduced work intervals, built-in quality and global optimization support this. The result is improved quality and throughput enabling experimentation.

The second way is about fast and constant flow of feedback from right to left. It leads to safer systems of work where errors are detected long before they result in catastrophic consequences. Problems are swarmed until effective countermeasures are in place

The third way creates a generative, high trust culture with dynamic disciplined approach to experimentation and risk taking. Organizations learn from both success and failure, multiplying the effects of the gained knowledge from local discovery to global application. Everyone working in the organization has all the background from all the experimentation and learning to do their work.

DevOps - The First Way Feb 27, 2021

The First Way of DevOps is about establishing fast and smooth flow through the technology value stream - from Development to Operations to the customer. This reduces the lead time from request to delivery and at the same time improves product quality. In turn, winning in the marketplace becomes more likely, by the ability to respond to new requests quickly.

Optimizing the flow through the technology value stream follows a similar way to how Lean principles have been applied to manufacturing. A prerequisite to applying such principles to IT is making the initially invisible work visible. For this, techniques like Kanban boards or Sprint boards can be used.

Based on the work being visualized on a board, several optimizations can be applied. First, the work in progress (WIP) can be limited for a particular work station. This helps discovering problems in the flow, e.g. when a work station is not able to start any work due to a delay in another work station. It also helps avoiding efficiency losses due to extensive multitasking.

Another key optimization is to perform work in small batch sizes, ideally single piece flow, i.e. all the steps require to complete within the value stream are performed in sequence for a single piece. This reduces the time until the result of the work is available to the customer.

Reorganizing teams so that they can deliver value themselves without needing too many interactions or hand-offs reduces waiting time throughout the value stream. This goes hand in hand with a modular architecture of the product, without tight coupling of the different architectural elements.

For any optimization to be effective, it is key to elevate the constraint. As the constraint limits the overall throughput of the value stream, any optimization not happening at the constraint is not materializing into an immediate advantage. Usually in IT organizations, the constraint first lies in environment creation, which can be addressed by automation making self-serviced environments available on demand. After this, the constraint usually moves to code deployment and thereafter to test setup and execution. Both can as well be automated, resulting in a pipeline building, testing and deploying changes from Development automatically.

The Technical Practices of Flow Mar 12,2021

The First Way of DevOps can be supported by the following practices:

(1) Deployment pipelines
(2) Fast, reliable automated testing
(3) Continuous integration and testing
(4) Automating, enabling and architecting for low risk releases

These practices reduce lead time to production-like environments to enable fast feedback to developers, making deployments a routine part of their daily work. In addition, they reduce hardship, firefighting and toil. People become productive with an increased joy in their work.

While an Agile transformation improves planning and may also contribute to a higher degree of autonomy and joy in the developer's work, there are cases, where it provides only little improvement in productivity. This is addressed by the First Way, as it targets automating the environment creation process in order to make it repeatable. This enables every developer to run production-like environments on their own workstations.

Knowledge is captured in automated build mechanisms instead of just creating Wiki pages:

> "All our requirements are embedded, not in documents or as knowledge in someone's head, but codified in our automated build process"

This principle can be applied to production code in a similar way along with test-driven development, where the requirements and the detailed design get "encoded" in unit tests.

All artifacts belonging to the software product are kept in version control, creating a single repostitory of truth, containing (1) all application code and dependencies, (2) any script needed to create DB schemas or reference data, (3) all environment creation tools, (4) any file to create containers, (5) all automation tests and manual test scripts, (6) any deployment, migration and other scripts, (7) all project artifacts, e.g. requirements, release notes and (8) any other script or configuration. It is <u>not</u> sufficient that the production environment is recreatable, all of the "pre-production" environment, especially the build process, must be recreatable from the artifacts under version control.

This leads to infrastructure that is easier to rebuild than to repair so that production environments are routinely killed. Servers are treated as "cattle", not as "pets".

Finally, the Definition of Done should include "Running in production-like environment". As the automation enables developers to easily start and deploy into production-like environments, this step needs to be part of their routine process of completing their work.

Automated Testing - Fast and Reliable March 16, 2021

Fast and reliable automated testing helps establishing the link between cause and effect of a bug, as developers receive quick feedback. If code is brought into production without testing, teams check in large and infrequent changes conflicting with those from other teams. Automation testing pushes down the maintenance cost of newly added code, as with manual testing the maintenance cost grows more and more over time, as the code base is being extended.

When run in a pipeline, automation tests can be executed on every commit, ensuring that each change is immediately tested in a production-like environment. All build, test and integration errors are discovered quickly, enabling early fixing. This way, the pipeline becomes a prerequisite to commit code, so it must be very easy to roll back in case of errors. Everyone needs to be committed to do what it takes to keep the pipeline running, prioritizing it over development work. This is established by a high-trust culture and engineering professionalism.

The pipeline stores the history of each build, which tests were done on which build, which builds were deployed where and what the test results were of a particular build.

Three central capabilities are required for "continuous integration":

(1) Comprehensive and reliable automated tests, validating that the product is in a deployable state

(2) A culture to stop the production line on failure

(3) Working on small batches on trunk rather than on long-lived feature branches

These capabilities enable fast and immediate feedback. In contrast to this, slow and periodic feedback - e.g. through builds running over night - kills productivity. For instance, if ten developers work on a project with one change breaking the nightly build, it takes minutes or even hours on the next day to figure out what broke the build and how to fix it. Developers might think the issue is fixed once all unit tests pass and only the next nightly build will again fail. Until then, more changes have been introduced, possibly adding more errors.

The idea of executing all tests in a pipeline gives rise to the question how to prioritize tests in the pipeline. The "test pyramid" is a model aiming at detecting errors early. At the bottom of the pyramid lay unit tests, that are high in number and quick in their execution time. They reflect the developer's perspective and are created by developers. On top of this, there is a layer of slightly less and slightly slower acceptance tests, reflecting the business or customer perspective. A third layer of tests are the integration tests, which are smoke tests against a production-like environment. Those are usually brittle and their number should be reduced, covering as much as possible already in the acceptance test stage. Finally, there are also manual exploratory tests and UI tests at the top of the pyramid, which should be executed only after all other tests passed and which should be lowest in number, as they take longest to run.

By covering large portions of the code base and the requirements with lower levels of tests, the build duration of the pipeline can be reduced and errors can be found earlier in the process. As a general rule, fast tests should be run before slow tests. Errors should always be detected with the fastest possible test. For example it can be beneficial to "shift left" the detection of errors by creating a lower level test for an error found in a higher level test. For instance, if an error is found in an acceptance test, it might be worth writing a unit test for the detected error.

Additional practices contributing to fast feedback include parallel execution of tests and test-driven development. In addition to functional aspects, the automation test suite should to some extent cover performance and other non-functional properties, e.g. security.

Crucial to the success of automation testing is the reliability of the test suite, as an unreliable test suite will not be trusted by developers, leading to it being bypassed with all the undesired effects of untested code being checked in. For this, as much as possible should be automated with a particular focus on reliability of the automation. In this context, the concept of the Andon cord can help improving reliability of the builds and the tests being executed within its pipeline. Whenever an error is detected, the cord is being pulled in order to stop all work and just focus on fixing the issue. This Andon cord can have a physical representation such as a red light in the office indicating a build failure. However, the essential part of it is the cultural aspect of everyone being ready to interrupt their work to focus on fixing the build.

Continuous Integration March 17, 2021

With the number of branches and the number of code changes, the complexity of integration grows exponentially. Integration problems can then result in significant rework needed. The result is an even lower frequency of integration, resulting in a "downward spiral" of less frequent merges.

An industrial case study at HP showed that before adapting continuous integration practices, developers spent about 5 % of their time creating new features, while most of their time was spent on unproductive activities, such as detailed estimation, porting and integrating code or manual testing. The measures taken include continuous integration, trunk-based development, heavy investments in test automation, a hardware simulator, the reproduction of test failures in developer's environments and re-architecture. Especially trunk-based development requires a huge shift in mindset. After its introduction, all 24 product lines were developing on trunk with XML configurations enabling their individual features at runtime. The result of these changes amounted to an increase of the time spent on creating new features from the initial 5 % up to 40 % in addition to significant savings in costs.

Along with the consistent use of version control, continuous integration ensures fast flow throughout the value stream. One of the enablers of continuous integration is the branching strategy used in a project. Branching strategies can generally be found on a spectrum between two extremes:

(1) Optimizing for <u>individual</u> productivity. Developers work on their private branches. Noone disrupts anyone else's work. Merging becomes a nightmare.

(2) Optimizing for <u>team</u> productivity. Everyone works in a common area. Commits are simple to understand. Each commit can break the entire project.

As the "merge hell" resulting from the branching strategies of the first kind results in delayed feedback, rework required by others due to refactorings and thus higher integration costs and payoffs not being realized, trunk-based development has been created. It optimizes for team productivity over individual productivity, as commits to trunk happen at least once per day for every developer. This eliminates the need of a separate test and stabilization phase at the end of the project and instead enables one-click deployments

from trunk into production.

Automation for Low-Risk Releases Mar 20, 2021

Experiences from industrial case studies show there is a fixed number of changes that can be handled in a deployment. This implies that for handling a higher number of changes, the frequency of deployments needs to be increased. A higher release frequency can be supported with automation, by removing steps that take long and by removing handoffs between teams. Automation will become part of the deployment pipeline, providing the following core features:

- Packaging software for deployment
- Showing readiness of the product for deployment
- A button to push for the deployment
- Traceability of all actions taken for a deployment (audit)
- Smoke testing of the deployed product
- Fast feedback to developers

This leads to three essential requirements to the deployment pipeline: (1) Deploy to every environment in the same way, (2) Smoke test the deployment and (3) Ensure consistent environments by synchronizing them in the right way.

In detail, the pipeline may cover packaging of the source code in a deployable format, the creation of virtual machines or images, the deployment or configuration of middleware, copying files to servers, restarting servers, applications and services, generating configuration files from templates, smoke testing, testing procedures and database migration.

Experience from industry has shown that daily deployments and a higher release frequency lead to more reliable products with less production incidents and a lower mean time to recovery (MTTR).

To optimize the support of developers, deployments should be available as automated self-service to Dev, Ops and possibly other stakeholders based on their needs. In practice, the ability of developers to deploy their own code into production has been reduced due to security and compliance requirements. With DevOps, these requirements need to be covered by other - equally or even more effective - control mechanisms, such as automated testing, automated deployment and peer review.

One more important principle for low-risk releases is decoupling releases from deployments. While a deployment is the installation of a product in an environment and falls into the responsibility

of Dev and Ops, a release is the point in time where the product becomes available for customers. Releases should be in the responsibility of product owners.

To achieve this segregation between deployment and release, several patterns are available, falling into one of the following two categories:

(1) Environment-based patterns, such as canary releases, blue-green deployments or cluster-immune systems

(2) Application-based patterns, e.g. dark launches supported by feature toggles; these also facilitate rollbacks, support the graceful degradation of performance by providing the ability to deliver a lower quality of service to avoid interruptions in high-load scenarios and contribute to a higher resilience by encouraging service-oriented architectures.

The principle of decoupling deployments from releases also lays the foundations for defining "Continuous Deployment" and "Continuous Delivery". There, continuous delivery refers to continually delivering changes into trunk and keeping trunk always in a releasable state so that releases are reduced to the push of a button. Continuous deployment refers to actually deploying into a production system without the actual need to release on this deployment.

Architecting for Low-Risk Releases Mar 22, 2021

Tightly coupled architecture, which is created due to IT project owners not being held responsible for the "overall system entropy" is one of the reasons for entering the "downward spiral" of deploying less frequently. The solution to this problem is creating loosely coupled architectures with well-defined interfaces. This contributes to productivity and safety of the organization. Such loosely coupled - service-oriented - architectures enable small teams to make small changes, work on simpler units and deploy independently, quickly and safely.

Conversely, monolithic architectures do not scale to this extent, they do however provide some other advantages, especially to organizations early in their product lifecycle, such as start-ups. Accordingly, the architectural needs of an organization change over time. To deal with such changes, the evolution of architecture needs to be considered. Many large companies re-wrote their software architecture several times from scratch. Such re-engineering must be conducted while still supporting the existing rolled-out products. One approach to such evolution is the piloting of new architecture within some teams and the roll-out of re-engineerings based on their expected value they create for the business. From a technical perspective, the evolution of architecture is supported by the clear definition of interfaces, e.g. APIs. The strangler application pattern is an example how such APIs can be introduced in an organization. The existing system is placed behind an API and not changed as frequently as newly introduced services. New services will follow the newly defined architecture. Several evolution steps are then clearly separated by producing separate versions of the API. The versioning of API services is one core concept behind this pattern and architectural evolution in general.

DevOps - The Second Way

Mar 3, 2021

The Second Way of DevOps is about the fast and constant feedback from right to left through the technology value stream, i.e. from the customer to Operations and Development. It contributes to a safer and more resilient system of work.

General prerequisites to a safer work system are (1) the management of complex work, (2) swarming and solving problems, (3) exploiting new knowledge at a global level and (4) leaders creating other leaders.

The feedback and feedforward loops in the value stream provide fast, frequent, high quality information that leads to high performance outcomes. In the technology value stream, the mechanisms to ensure such fast and constant feedback are automated builds, integration, testing and deployment. Whenever one of the steps in the pipeline fails, the feedback is there for immediate action. Moreover, pervasive telemetry in the product allows to receive feedback directly from the production use of the system.

Whenever a problem occurs affecting the value stream, swarming and jointly solving the problem helps in creating new knowledge from it. A famous example for swarming is Toyota's Andon cord. This is a cord that every work center has and that every employee is trained to pull when observing a problem. Once pulled, the team lead of the work center is informed and the team has limited time to work the problem to its resolution. If after the time limit (e.g. after 55 seconds) the problem is still unsolved, the entire organization will be summoned to jointly solve the problem. This way, every member of the organization will be participating in the learning process and everyone will get the viewpoint of the entire value stream, stepping out of his or her local viewpoint from his or her respective work center. Swarming also prevents defects from being passed downstream and it prevents new work from being taken up when the problem might prevent that new work from being successfully processed. The equivalent to the Andon cord in technology is given by test and build failures.

In addition, quality needs to be brought close to the source, avoiding inferior quality products to be passed down through the value stream.

This includes avoiding ineffective quality controls, such as:

(1) Depending on tedious and error prone manual work to be done by other teams

(2) Depending on approvals from busy people wo are distant from the work and have no adequate knowledge of it.

(3) Creating large volumes of documentation of questionable detail that is obsolete shortly after it is written

(4) Pushing large batches of work onto teams or special committees for approval or processing and then waiting for a response

Finding and fixing problems in the overall work system needs to become part of everyone's daily work, so that decision making is done where the work is being performed. For this, peer reviews and automated tests should be used; quality needs to become everyone's responsibility.

This way, the work should be optimized for downstream work centers, e.g. by designing software for Operations by prioritizing operational non-functional requirements as highly as user features.

Establishing and Using Telemetry for Fast Feedback Mar 23, 2021

Sufficient telemetry needs to be in place to confirm that our services are operating correctly. For this, data collection is required at business logic, application and environment layer. The data collection produces events, logs and metrics. For storing events and metrics, an event router enables visualization, trending, alerting, anomaly detection etc. Metrics need to cover all levels of the application stack and they should be generated from production and pre-production environments, including the build pipeline.

Logging on different levels supports every member of the value stream. Log events should be generated by authentication and authorization (including logoff), system and data access, system and application changes, invalid inputs, resources, health and availability of the system, startup, shutdown, circuit breaker trips, delays and backup success and failure.

Using telemetry enables a scientific approach to problem solving, driven by hypotheses. The creation of production metrics should become part of the daily work. Self-service access to telemetry and information radiators promote openness, as they show that teams have nothing to hide from visitors and that they are open to confront problems.

Metrics are required on various levels. Starting on business level (e.g. number of sales transactions) over application level (e.g. application faults), infrastructure level (e.g. CPU load), client level (e.g. application errors) down to the deployment pipeline (e.g. build status), metrics are collected. Security related events can be detected by checking application errors and faults. Comprehensive telemetry enables to detect problems early, when fixing them is still cheap.

Application and business level metrics provide fast feedback to development teams. This helps them to see if their changes achieve the business goals. Graphing infrastructure metrics next to business and application metrics helps finding causes for events. In addition, relevant information, such as deployments, can be overlayed to metrics in order to better understand how quickly performance returns to normal after such an event.

Analyzing Telemetry

To analyze telemetry, statistical methods need to be applied and the required competencies for this must be made available to the development teams.

A basic method for discovering abnormal running conditions, from which significant performance degradation may result, is outlier detection. This method comprises three steps. First, the current normal is computed. Second, nodes are identified that do not match the current normal. Third, these nodes are removed from the population.

To detect variations, mean and standard deviation provide good guidance for Gaussian distributions. Setting a static threshold value based on mean and standard deviation allows to track only those values with significant variations, focusing only on significant incidents. Alerts can be set up on significant deviations from the mean.

For improving telemetry, the most severe incidents in the recent past serve as examples that can be investigated. Creating a list of telemetry that would have enabled early and fast detection of these problems is likely to also provide value for detecting future problems.

For non-Gaussian distributions, standard deviation does not apply and in practice often Chi-squared distributions will be observed. In such cases, anomaly detection requires other statistical methods such as smoothing or Fast Fourier Transformations. This again supports the idea of including experts on statistics into product development teams.

Conway's Law

Mar 24, 2021

> "Organizations, which design systems (...) are constrained to produce designs, which are copies of the communication structures of these organizations."
>
> - Conway's Law

Due to their increasing inflexibility, Conway's Law in particular applies to large organizations. In more concrete terms, the law states that the way teams are organized impacts the architecture of the software products being developed.

There are three well-known organizational archetypes: functional, matrix and market-based organizations. Functional organizations are the type of which classical operations are - characterized by tall hierarchies and an optimization for costs. They avoid any redundancies across the organization. On the downside, they tend to create constraints due to the competition for resources. In contrast, market-oriented organizations have flat hierarchies and allow for lots of redundancies across the organization. This is because they are optimized for speed instead of being cost-optimized. The main benefit is a quick response to changes on the market. Matrix organizations are a combination of the functional and the market oriented approach. While they aim at combining the advantages out of the two worlds, they usually fail to achieve their goals due to a tendency of becoming overly complex.

Keeping in mind the architectural ideal of loosely coupled, service-oriented architectures with bounded contexts and a design driven by the domain, an ideal, in which it is not required to know anything about the internals of other services for developing one, the organization producing such a design follows from Conway's Law. As the design would be a mirror of that organization's communication structures, the organization must be set up accordingly. This means small teams work independently of each other, each team being responsible for a service. Amazon applied this and coined the term of a two-pizza-team, which is a team of the size that it can be well-fed by two large pizzas.

Experiences from Toyota - a functionally organized organization - have shown that it is even more decisive how people behave inside an organization than how the organization is structured. Toyota is an example for an efficient functional organization, which is due

to the way people act and react. They avoid creating constraints, as every member of the organization takes a global viewpoint. The goals of testing, operations and security should be in everyone's mind so that everyone strives for quality, availability and security during their everyday work.

Team members taking such a global viewpoint are rather generalists than specialists. They remove bottlenecks and avoid downstream waste in the value stream. In addition, they make planning more flexible and absorb variability. In order to evolve specialists into generalists, they need to get a broader knowledge over several areas in addition to their specialization - which remains important. They also need to collect experiences across several areas. For this, the organization needs to encourage learning, developing a "growth mindset".

Deciding to organize around products and services rather than projects supports the idea of optimizing on a global level. This is as product and service teams are driven by more long-term goals, such as revenue and customer lifetime value. Project teams target to deliver a defined scope on time and within budget.

Products and services operated and developed by Two-Pizza-Teams are small enough so that the team has a clear, shared understanding of what they deliver. The limited size of the teams also limits the growth rate so that the shared understanding evolves along with the product without growing to high levels of complexity too quickly. Decentralized power distribution in the organization also enables a higher degree of autonomy, minimizing risks. And the organization into small teams creates opportunities for employees to grow by taking the lead within their team without the risk of a catastrophic failure.

Safe Deployments

Mar 27, 2021

Deployments into complex systems without telemetry or quality mechanisms lead to failure!

Use telemetry to monitor a service and ensure it operates as desired without breaking anything else. Turn off broken features or fix forward. Fixing forward requires automated tests and fast deployment processes.

For globally optimizing at the value stream level, Development takes over some of the classic Operations duties. Development shares pager rotation duties with Operations, leading to a better understanding of operational problems and a faster delivery of related fixes by developers. Development also follows their work downstream, e.g. through UX observations. This is known to generate significant learnings for the developers. Such learnings can be integrated into the backlog as non-functional requirements. Development even initially self-manages their production services before they become eligible for central operations groups to manage. Operations engineers serve as consultants to developers during this phase. Clearly defined launch requirements and guidelines additionally support the development teams. These may include the number of defects and their severity, the type and frequency of alerts, the monitoring coverage, the system architecture, the presence of a predictable and deterministic automated deployment process, as well as good production habits that would allow the service to be managed by anyone else.

From this involvement into operational questions, developers will learn about compliance issues, e.g.. the handling of personal data, which helps considering such issues early in the product development process.

Based on the readiness for operation by a central Operations team, hand-offs and hand-backs from and to Development can be planned. The hand-back into self-management by Development is an option when new developments with strategic importance are planned or when a service becomes too troublesome to operate for the central Operations group.

Hypothesis-Driven Development Mar 27, 2021

In order to minimize waste and maximize customer focus, rigorously ask the question "should we build this new feature, and why?". Then perform the cheapest and fastest experiments to validate whether the feature can achieve the desired outcome.

These experiments are conducted through A/B-testing, e.g. through randomly showing one of two variants of a website to users in order to determine if there is a significant difference between the two regarding business outcome. With A/B-testing, startling results have been observed at Microsoft, where only 1/3 of experiments lead to improvements. This means that 2/3 of all features that would be built without A/B-testing have no or even negative impact on business. Additionally, building them would come at the price of not building other features with positive impact on business, i.e. at an opportunity cost.

Technically, A/B-testing is supported by feature toggles. There are also some frameworks, e.g. the Etsy A/B API, which is open source.

Already during planning, product owners should think about hypotheses for the features to be included, with releases being chances to prove or disprove those hypotheses. The plan needs to be broken down into small requirements and stories, which are validated through A/B testing. Whenever needed, adapt the roadmap. An example for a hypothesis as it can be used for planning is given below:

> **We believe** that increasing the size of hotel images on the booking page
>
> **will result** in improved customer engagement and conversion.
>
> **We will have confidence to proceed when** we see a 5% increase in customers who review hotel images who then proceed to book in fourty-eight hours.

Performing experiments daily or weekly supports shifting focus from feature work to customer outcomes!

Failed deployments are typically explained through one of the following two counterfactual narratives:

(1) Change control failure: The error could have been prevented if better change control mechanisms were in place.

(2) Testing failure: Better testing could have prevented the error.

However, especially in environments with a low-trust command and control culture, measures addressing those narratives often increase the likelihood of problems, often worsening outcomes. This is as often measures chosen - such as approvals for change control or the decision towards more testing - increase lead times and batch sizes. This results in the known downward spiral to less frequent deployments, often with catastrophic outcomes. While in testing, this downward spiral is usually entered through manual testing and can be effectively avoided with automation, the issue is more complex for change control. Change control in practice comprises the review and approval as well as the coordinated integration of changes.

As far as the review and approval is concerned, outcomes worsen with an increasing distance between the person doing the work and the person deciding about the work. Relying on review boards distant from the developer's work, such as requiring a CCB to manually evaluate and approve every change, is not effective. Control practices need to be close to the developer's work, more resembling peer reviews.

Some general guidelines for reviews a project may apply, are:

- Have at least one fellow engineer review every change
- In some cases, where things are more complex, involve subject matter experts or several reviewers based on the risk of a change
- Large changes produce a large burden on the reviewer; thus apply the principle of small batch sizes ("trunk-based development") also to code reviews

Reviews should be done before integrating a change into trunk. There are several forms of reviews, from pair programming over online reviews to offline reviews using mail or specific tools. Pair programming is a good way to establish a review culture and

it also significantly improves code quality. In case changes are reviewed offline, they must be small enough to reason about. For large changes, splitting them into several smaller changes with every change understanable at a glance enables an effective review.

The effectiveness of reviews can be monitored using review statistics (e.g. approved vs. rejected changes) and by sampling and inspecting single reviews.

In addition to reviews, changes need coordination to avoid side effects and collisions. A loosely coupled architecture is a good means to reduce such. Still, additional coordination is needed, as even a good architecture will not completely remove dependencies. A basic approach to such coordination is using a chat room to announce changes and have teams proactively find possible collisions, e.g. through monitoring the commit stream of others. In more complex environments, changes can be deliberately scheduled and technical countermeasures can be taken during rollout, such as redundancy or fallover. This further reduces the risk of deploying the changes.

The rollout of changes without bureaucracy ensures their efficient deployment. A good metric for this is the number of meetings and tickets needed to release changes. To initially remove bureaucracy, teams can proactively take over responsibility for integration.

A successful workflow for integrating changes is the GitHub Flow, consisting of five steps:

(1) Create a branch
(2) Commit to the branch
(3) Open a pull request
(4) Merge into master
(5) Deploy into production

The core concept for reviews there is the pull request, which enables the developer to request for a review. Good pull requests provide the reviewers with sufficient context information and they state what expectations the developer has to the review (e.g. the number of reviewers).

DevOps - The Third Way

Mar 3, 2021

The Third Way of DevOps focuses on creating a culture of continual learning and experimentation. High performance organizations require and actively promote learning. In such organizations, the system of work is dynamic. For instance, in manufacturing, line workers are performing experiments in their daily work to generate new improvements.

Such a dynamic system of work depends on a high trust culture, in which everyone is a life-long learner and everyone is taking risks as part of their daily work. There is a scientific approach to process and product development. The organization learns from both success and failure. This kind of culture is also called a generative culture, as opposed to pathological and bureaucratic cultures. Culture is an important prerequisite to continual learning.

To establish the cultural prerequisites, two steps can be performed:

(1) Remove blame and replace it with institutionalized learning, e.g. through establishing "blameless post-mortems"

(2) Remove fear to enable honesty, which in turn is a prerequisite for prevention.

Due to chaos and entropy, processes generally degrade over time. Thus, it is even more important than our daily work to improve our daily work. For this, some time needs to be explicitly reserved. One option is to reserve some development cycles explicitly for improvement, e.g. as it is done in SAFe with the IP Sprints. Another option is to schedule kaizen blitzes, during which all members of the organization get the chance to work on improvements of their choice. By continued exposure to the improvement process, the members of the organization will learn to listen for weaker failure signals and improve situations that would otherwise even not be detected. Generally, this will replace coping, firefighting and make-do with a dynamic of identifying opportunities for process and product improvements.

In addition to creating improvements, it is important that the whole organization can benefit from the improvements throughout their daily work. For this, local improvements must be made available at a global - organizational - level

For this, collective knowledge can be made available by providing a searchable documentation base that can be used when solving further problems. Additionally, shared organization-wide code repositories help sharing solutions and thus sharing knowledge.

Resilience patterns can be introduced to reduce deployment lead time, to increase test coverage, to decrease test execution time or even to re-architect for improved developer productivity.

The role of leaders in such a setting is to create conditions in the team that the team members can discover greatness in their daily work. Leaders must elevate the value of learning and disciplined problem solving. Leaders help by defining strategic goals and refining them into further short-term goals that provide the target conditions for the experiments conducted by the teams.

Leaders support in coaching people in conducting experiments with questions. Such questions target at establishing a scientific approach guiding all internal improvement. This includes how experiments are performed to ensure products achieve customer goals.

One good example for this is Toyota, which is known for unique behavior routines taught to all its members continually, where leaders help workers and solve problems in their daily work.

Practices of the Third Way April 2nd, 2021

The Third Way of DevOps is all about creating opportunities for learning as quick, as frequent, as cheap and as soon as possible from accidents and failures. Such learning results in a higher resilience and collective knowledge. In turn, the organization better achieves its goals.

The following prerequisites support the Third Way:

- Make safety possible through a just culture
- Inject failures into production for improved resilience
- Create global improvement from local discovery
- Reserve time for organizational improvement and learning

Learning needs to be part of our daily work. This way, resilience is designed into the architecture. A good example is the "Chaos Monkey" at Netflix, a service designed to kill production servers in order to learn from the resulting failures.

A just culture is based on the assumption that not human error is the cause of trouble, it is rather the consequence of the tools we gave them. Thus, removing humans who produce errors will not address the root cause. Instead, maximize the opportunities for organizational learning and continually reinforce the value of actions exposing problems more widely. Punishment of errors is counter-productive to this, as it dis-incentivizes engineers to share knowledge on errors.

A core technique supporting a just culture are blameless post-mortems. The first action in such a post-mortem is to reconstruct the timeline leading to the accident, gathering details from multiple perspectives on failures. It is crucial to the success of the post-mortem to not punish people for mistakes. Everyone gives detailed account of their contribution to failures. People should be encouraged to educate the organization on how to avoid mistakes they have made. The organization also accepts discretionary space where people decide to take action or not. Judgement is always in hindsight. People propose countermeasures and ensure for each one of these there is an owner and a target date set.

The attendants of a blameless post-mortem include people involved in decisions contributing to the problem, people identified, responded to, diagnosed, or were affected by the problem. Anyone else who is interested, should be invited as well.

During the post-mortem, do not "would-have" and "could-have", stay with facts. A good question is "Why did it make sense to me when I took that action?". Brainstorm on countermeasures. Prepare for a future where we are as stupid as we are today, i.e. do not assume to become smarter or "just be more careful".

The outcome of the post-mortem needs to be published to reach a wide range in the organization. This way, local discovery of a problem and its countermeasures is made available at a global level. Once learning improves quality, decrease the tolerance to find weaker failure signals. For this, also the attitude towards failure needs to be checked. More failures need to be considered a good thing, as they provide more opportunities to learn. Go away from "mean time between failures" towards "mean time till recovery". Encourage calculated risk-taking by the teams.

To improve system resiliance, failures injected into production systems provide another opportunity to learn. Game days are events for defining and executing drills for failures. These are held before a scheduled failure. When the scheduled failure date arrives, execute the failure without backing out of it. This way, the organization will learn any side effects of the failure, e.g. if the systems for resolving the failure work as expected under the failure condition.

Regular failures in production followed by Game days strengthen the relationship between people from different departments who need to work together to resolve the failure. In addition, resolving failures becomes a routine for employees, improving their reaction under failure conditions.

Turning Local Discovery into Global Learning April 3rd, 2021

Elevating the state of the practice of an organization using local discoveries and turning them into global - organizational - learnings is a key enabler to the Third Way of DevOps. For this, several practices exist:

- chat rooms and chatbots
- integrating procedural knowledge and NFRs into the code base
- a single-source repository at organization level
- sharing Ops knowledge in user stories
- optimizing technology choices towards organizational goals

Chat rooms and chatbots help distributing the knowledge generated out of local learnings. A chatbot can even be used as an alternative to the command line. Engineers pass commands, e.g. for deployments, to the chatbot, which in turn executes the command. This way, the operation history and the outcome of the commands is visible to a broader audience. Transparency is one of the key advantages of this. Onboarding of new team members becomes easier, as information is more available. The use of chatbots also facilitates members of the organization helping each other and results in overall organizational learning. Chat rooms also record the communication, keeping a history of it and making this publicly available throughout the organization. This can be useful for events that need to be known to other members of the organization to work efficiently, such as the pipeline status, deployments and code commits of other developers.

Procedural knowledge in documentation is not as easily spread as the code base. Thus, such knowledge needs to become part of the code base in order to be spread across the organization. This will also help resolving ambiguities between different interpretations of the documents and end in a higer quality of procedural knowledge. Additionally, non-functional requirements need to be codified in order to get a precise understanding of them. Specifications are most easily shared in the form of automated tests created using the test-driven development approach.

Keeping knowledge at the organizational level is easiest through a single-source repository. This repository contains configuration standards for libraries, infrastructure and environments, deployment tools, testing standards and tools, including security, deployment pipeline tools, monitoring and analysis tools and

tutorials and standards.

Every library used in the organization - internal or third-party - has an owner, who acts very much like a real-world librarian. The owner ensures that the library passes all tests for all projects depending on it. For each library and also for services, a community of practice helps, as users can ask other users in discussion groups and get answers quickly.

For Ops activities, user stories keep the knowledge of which steps need to be taken and what effort is required. The user stories can be re-used for several instances of the same action. Planning of Ops activities is then done in the same way as for Dev activities, facilitating the creation of an overall plan view.

The alignment of technologies with the goals of an organization is one more important contribution to the Third Way. Technologies need to support the organization's goals. Thus, technologies that slow down the flow, create unplanned work or support requests in a disproportionally high volume or do not fit into the overall architecture must be eliminated as part of the learning process.

Institutionalized time for organizational learning and improvement is part of the Third Way of DevOps. For this, two key requirements need to be met:

(1) Reserve time for individuals, teams or even the whole organization to work on improvements. This is easiest through scheduled events, such as improvement blitzes, hackathons, etc.

(2) Create a culture for everyone to teach and learn in various ways. Learning something new from other members of the organization should not be embarassing, and everyone considers himself a lifelong learner.

Good examples for the first of the two points include Toyota's kaizen blitzes, Facebook's hackathons and Google's Fixits. These are all scheduled events during which members of the organization focus on improvement. Kaizen blitzes usually last a few days with a group of people focusing on process improvement. People usually within the process to improve take the support of people outside the process to generate improvements. A hackathon at Facebook is a dedicated time - usually a few days every couple of months - in which everyone builds prototypes for new ideas they have. At the end, everyone shares the results. Within Google, Fixit events are for the entire organization to contribute. They usually span a large number of participants from different offices across the globe. As focused missions at critical points in time, they provide excitement and energy in addition to generating and spreading knowledge.

The culture of teaching and learning inside an organization is best supported by members of the organization who act as internal coaches and consultants. Giving employees time to focus on creating and spreading knowledge has led to the formation of grouplets at Google, where people with similar interests use their time to improve the organization. In one example, a grouplet started publishing newsletters on testing topics in the restrooms of Google offices worldwide - reaching far more members of Google than any online publication could. Another starting point towards learning culture is to share knowledge from external conferences within the organization. Target is one very advanced example of an organization with a learning culture. They established a Dojo

with full-time members and 18,000 square feet of office space focusing on the improvement of the organization throughout the entire work day. Development teams visit the Dojo to improve their daily work, conducting improvement blitzes with the Dojo's coaches and engineers for up to 30 days in a row.

Information Security as Everyone's Daily Job April 6th, 2021

Compliance checking is the opposite of security engineering and makes everyone hate information security. Instead of retroactive compliance checking, InfoSec needs to become part of the technology value stream along with Dev and Ops. A starting point to this is inviting InfoSec to all product demos for guidance and feedback. This way, InfoSec is involved in the product development process. Through this involvement, InfoSec is exposed to more context information on the product development, which leads to better risk-based decisions.

As the way of working is concerned, InfoSec needs to be part of the technology value stream by using the same tools for tracking their issues, as Dev and Ops are using, and by placing their artefacts in a version control system. Also the means of test automation, the deployment pipeline and production telemetry need to become a channel for InfoSec to enable Dev and Ops to deliver secure and compliant products.

For example, encryption and authentication libraries cleared by InfoSec become part of the version control repository. Based on this, a shared security-relevant platform is then built and operated within the organization. As InfoSec was involved in the definition of this platform, teams throughout the organization make use of it without the need of special security reviews or additional clearance.

Additional means of ensuring information security in the deployment pipeline include static analysis, i.e. checking the code base for vulnerabilities, dynamic analysis, i.e. checking the running system, dependency scanning and the signing and integrity-checking of source code. The dependency scanning supports the security of the software supply chain, another important aspect InfoSec needs to cover.

To check the security status of production services, telemetry is established for InfoSec in a similar way as it is done for Dev and Ops. The data to be observed may include user logins, password resets, e-mail resets or changes of credit cards. The information gained from telemetry radiates how services are attacked in production and helps the organization to get knowledge on this. To optimize the overall value stream also considering InfoSec, the metric "compliance response time" measures how the organization responds to quantitative and qualitative information requests from

audits.

In addition to the use of tools for securing services, their dependencies and their environment, InfoSec needs to care for protecting the deployment pipeline. This is achieved through the following means:

- hardening the continuous build and integration servers and being able to reproduce them in an automated manner
- review all changes introduced into version control
- instrument repositories to detect suspicious API calls
- ensure the continuous integration processes run on their own container or VM
- ensure version control credentials from the continuous integration system are read-only

To even further strengthen the security of the deployment pipeline, re-classify changes to not require approval of change control boards and instead rely on the automated deployment of routine changes wherever possible. Changes that need approval need to provide enough context information on the change request to enable the approving board in their decision. Moreover, reduce the reliance on separation of duty as a control mechanism in favor of other controls, such as pair programming, the continuous inspection of check-ins and code reviews. Furthermore, the pipeline ensures documentation for auditors and compliance officers is in place when needed.

Starting a DevOps Transformation Mar 6, 2021

For starting a DevOps transformation, it should be first checked, which types of projects participate in the transformation. Generally, there are two types of projects, which need to be handled differently in a transformation: Greenfield projects and brownfield projects. While greenfield projects are newly started software products without any history, brownfield projects operate in an established product landscape.

Greenfield projects ideally serve as a demonstrator of the feasibility of applying DevOps in an organization. This allows piloting the new development approach before a broader rollout in the organization. The team of the pilot project can operate outside the boundaries of established processes that would usually apply.

Brownfield projects need to consider the constraints from products that are in the field for many years or even decades. These products might have accumulated significant amounts of technical debt, such as lacking test automation, unsupported platforms, etc. About 60 % of all projects are brownfield projects. These projects sometimes show a large performance gap between customer need and delivery, due to which they highly benefit from introducing DevOps. However, these projects are also usually the most challenging ones, which is less due to the age of the software products and more due to shortcomings in their architecture and otherwise accumulated technical debt. Key to successfully transforming such a project is that they either already have an architecture supporting testability and deployability or that they can be successfully re-architected to support these quality goals.

In addition to the classification above, IT products can be distinguished into "systems of record" and "systems of engagement". Systems of record are usually changing at a slow pace and the organizations goal with those is doing it right, as they are constrained by regulatory requirements. One example is an ERP system. Systems of engagement are those with customer-facing UIs. Those need to change at a higher pace, needing experimentation to find out the customer needs. Organizations aim at doing it fast on those systems. As DevOps closes the gap between "doing it right" and "doing it fast", there is no need to restrict its application to one of these two types of systems, both will benefit from applying DevOps in a different way.

Executing a DevOps Transformation Mar 6, 2021

When starting a DevOps transformation, the question arises, how to actually bring it into the organization. The answer to this is to start with innovators and early adopters. These will be the ones that believe in the need for DevOps principles and practices. Ideally, these are also respected and have a high degree of influence over the organization. After the first step of gaining sufficient support from this group, build a critical mass to have a stable base of support. This silent majority creates a bandwagon effect and pulls more supporters on your side. During this phase, it is crucial to avoid dangerous political battles. These are better bypassed than fought out. Only after having established a stable base of support, identify and tackle the "holdouts" in the third phase of the transformation. Those are the influential detractors that need to be convinced to complete the transformation.

During the transformation, a value stream analysis supports finding out how value is delivered to the customer. Members of the value stream need to be identified and considered, such as the product owner - defining the next set of functionality - or the development team responsible for delivering the functionality. Further members are QA, Ops, InfoSec, the release manager and the value stream manager or executive responsible for the value stream. The result of the analysis can be documented in a value stream map that visualizes the work in the value stream. It helps in particular to find places where work waits and places where work is generated. For each work center in the value stream, metrics such as lead time, value added time and percent correct and accurate ("%CA"). This way, problems in the value stream become visible.

Another prerequisite to a successful transformation is the setup of a DevOps transformation team. Its role can be compared to the one of the change team in change management. It should consist of members dedicating their full capacity to the transformation. Ideally, the members should also be generalists with a longstanding relationship towards the organization that needs to be mutually respectful. This DevOps team should be physically separated from other teams and it should have the liberty to operate outside the organizations regular processes. Communication within the DevOps team should be intensified, while the team is at the same time to some degree detached from the rest of the organization.

A high-level goal needs to be set for the DevOps team, e.g. reducing lead time below one week. From this, the work of the DevOps team should be planned in short cycles, enabling quick learning and limiting risk. The team should invest 20 % of their capacity to pay down technical debt, in order to avoid it being accumulated.

This '20 % tax' on technical debt can be applied to all teams in the organization, as it is a general concept to avoid the accumulation of technical debt. In some cases the percentage of capacity spent on paying down technical debt might be higher, however, if a team claims to get along with significantly less than 20 %, this might be a reason to have another look.

One more supporting factor of the DevOps transformation is the use of tools in the organization. Ideally, common tools are in place throughout the entire organization in order to allow everyone to take the overall system's viewpoint that is needed for global - rather than local - optimization. Such tools include the engineering and management tools used when developing products and supporting tools, e.g. chat rooms facilitating communication between people in the organization. While establisning direct communication between members of the organization from different teams helps towards global organization, there is also a risk of the team members being distracted from their work by taking requests through these informal communication channels. This risk needs to be mitigated by introducing appropriate planning procedures in the team and having all requests requiring work from the team flow through these channels.

Operations as Part of Development's Daily Work Mar 11, 2021

DevOps targets at small teams quickly and independently delivering value. With a centralized and functionally oriented Operations organization, this can be a challenge. Competition for the scarce resources in Ops can cause long lead times for Ops work followed by constant reprioritization and escalation, poor deployment outcomes.

To improve this, operations engineers and architects should be included in the Dev teams where possible. Where a dedicated operations engineer for a team is not an option, business relations between Dev and Ops can be improved by "Ops liasons". This means defining specific roles - one on managerial and one on technical level - for Ops work in each Dev team. These roles ensure that (1) a direct interface into the Ops organization is available to the Dev team, preventing work from being queued for long and (2) the technical viewpoint of Ops is considered in the product design and rollout.

Technically, services from Operations should be available to Development on demand, as a self-service. A set of central capabilities needs to be covered by the services provided by Ops, such as environment creation, telemetry and others. In order to avoid waiting times, all services are ideally fully automated.

The internal development of these services needs to follow the same principles as the development of tools for the external market does. This includes focusing on the needs of the (internal) customers. The customer focus can be strengthened by not mandating such platforms and allowing teams to chose whatever - internal or external - tooling they see fit in. This way, an internal market is created with competition that drives platform teams towards better products. All teams in such a setting will continually look for internal platforms that work well, as expanding the usage of something already working is easier than creating something newly from scratch.

The integration of Operations work and viewpoints in the Dev teams by dedicated engineers or Ops liasons helps Operations to better understand the functionality of the system and it helps Development considering the non-functional requirements from operational perspective as well as maintaining the link to rolled out products through production telemetry.

Including Ops in the Dev rituals helps the operations engineers to understand the culture in Dev. Two main rituals, where Ops should participate - in form of the dedicated engineer or the person assigned to the team through an Ops liason - are the Daily and the Retrospective. During the Daily, Ops will gain awareness of activities in order to support Dev more early and proactively. In Retrospectives, Ops can bring in another viewpoint for improvement and learning. Additionally, shared Kanban boards between Dev and Ops help to make dependencies on Ops work visible.