# A Cool Collection of DevOps Checklists

We have compiled DevOps checklists that you need to do what a DevOps guy does in a day or should do depending on your use case. This list covers different scenarios that you might be in.

- This checklist is comprised of 48 items you can use to gauge the maturity of your software delivery competency and form a baseline to measure your future improvements.
- Granular checklist around DevOps culture and it's unique goals.
- Perfect for starting out and implementing the DevOps culture.
- Checklists and Templates for DevOps catering to the following
  - ✓ Requirements Checklist
  - ✓ Runbook Template
  - ✓ SLA Templates
  - ✓ Deployment Checklist
  - ✓ Application Level Monitoring
- A DevOps checklist for successful app deployment.
- This list caters particularly for DevOps on Windows.
- It particularly covers high level do's for DevOps security, development and performance monitoring.
- AWS Well-Architected Framework Checklist by AWS

# DevOps Checklist

DevOps is the integration of development, quality assurance, and IT operations into a unified culture and set of processes for delivering software. Use this checklist as a starting point to assess your DevOps culture and process.

**Culture**

**Ensure business alignment across organizations and teams.** Conflicts over resources, purpose, goals, and priorities within an organization can be a risk to successful operations. Ensure that the business, development, and operations teams are all aligned.

**Ensure the entire team understands the software lifecycle.** Your team needs to understand the overall lifecycle of the application, and which part of the lifecycle the application is currently in. This helps all team members know what they should be doing now, and what they should be planning and preparing for in the future.

**Reduce cycle time.** Aim to minimize the time it takes to move from ideas to usable developed software. Limit the size and scope of individual releases to keep the test burden low. Automate the build, test, configuration, and deployment processes whenever possible. Clear any obstacles to communication among developers, and between developers and operations.

**Review and improve processes.** Your processes and procedures, both automated and manual, are never final. Set up regular reviews of current workflows, procedures, and documentation, with a goal of continual improvement.

**Do proactive planning.** Proactively plan for failure. Have processes in place to quickly identify issues when they occur, escalate to the correct team members to fix, and confirm resolution.

**Learn from failures.** Failures are inevitable, but it's important to learn from failures to avoid repeating them. If an operational failure occurs, triage the issue, document the cause and solution, and share any lessons that were learned. Whenever possible, update your build processes to automatically detect that kind of failure in the future.

**Optimize for speed and collect data.** Every planned improvement is a hypothesis. Work in the smallest increments possible. Treat new ideas as experiments. Instrument the experiments so that you can collect production data to assess their effectiveness. Be prepared to fail fast if the hypothesis is wrong.

**Allow time for learning.** Both failures and successes provide good opportunities for learning. Before moving on to new projects, allow enough time to gather the important lessons, and make sure those lessons are absorbed by your team. Also give the team the time to build skills, experiment, and learn about new tools and techniques.

**Document operations.** Document all tools, processes, and automated tasks with the same level of quality as your product code. Document the current design and architecture of any systems you support, along with recovery processes and other maintenance procedures. Focus on the steps you actually perform, not theoretically optimal processes. Regularly review and update the documentation. For code, make sure that meaningful comments are included, especially in public APIs, and use tools to automatically generate code documentation whenever possible.

**Share knowledge.** Documentation is only useful if people know that it exists and can find it. Ensure the documentation is organized and easily discoverable. Be creative: Use brown bags (informal presentations), videos, or newsletters to share knowledge.

## Development

**Provide developers with production-like environments.** If development and test environments don't match the production environment, it is hard to test and diagnose problems. Therefore, keep development and test environments as close to the production environment as possible. Make sure that test data is consistent with the data used in production, even if it's sample data and not real production data (for privacy or compliance reasons). Plan to generate and anonymize sample test data.

**Ensure that all authorized team members can provision infrastructure and deploy the application.** Setting up production-like resources and deploying the application should not involve complicated manual tasks or detailed technical knowledge of the system. Anyone with the right permissions should be able to create or deploy production-like resources without going to the operations team.

This recommendation doesn't imply that anyone can push live updates to the production deployment. It's about reducing friction for the development and QA teams to create production-like environments.

**Instrument the application for insight.** To understand the health of your application, you need to know how it's performing and whether it's experiencing any errors or problems. Always include instrumentation as a design requirement, and build the instrumentation into the application from the start. Instrumentation must include event logging for root cause analysis, but also telemetry and metrics to monitor the overall health and usage of the application.

**Track your technical debt.** In many projects, release schedules can get prioritized over code quality to one degree or another. Always keep track when this occurs. Document any shortcuts or other nonoptimal implementations, and schedule time in the future to revisit these issues.

**Consider pushing updates directly to production.** To reduce the overall release cycle time, consider pushing properly tested code commits directly to production. Use feature toggles to control which features are enabled. This allows you to move from development to release quickly, using the toggles to enable or disable features. Toggles are also useful when performing tests such as canary releases, where a particular feature is deployed to a subset of the production environment.

## Testing

**Automate testing.** Manually testing software is tedious and susceptible to error. Automate common testing tasks and integrate the tests into your build processes. Automated testing ensures consistent test coverage and reproducibility. Integrated UI tests should also be performed by an automated tool. Azure offers development and test resources that can help you configure and execute testing. For more information, see Development and test.

**Test for failures.** If a system can't connect to a service, how does it respond? Can it recover once the service is available again? Make fault injection testing a standard part of review on test and staging environments. When your test process and practices are mature, consider running these tests in production.

**Test in production.** The release process doesn't end with deployment to production. Have tests in place to ensure that deployed code works as expected. For deployments that are infrequently updated, schedule production testing as a regular part of maintenance.

**Automate performance testing to identify performance issues early.** The impact of a serious performance issue can be just as severe as a bug in the code. While automated functional tests can prevent application bugs, they might not detect performance problems. Define acceptable performance goals for metrics like latency, load times, and resource usage. Include automated performance tests in your release pipeline, to make sure the application meets those goals.

**Perform capacity testing.** An application might work fine under test conditions, and then have problems in production due to scale or resource limitations. Always define the maximum expected capacity and usage limits. Test to make sure the application can handle those limits, but also test what happens when those limits are exceeded. Capacity testing should be performed at regular intervals.

After the initial release, you should run performance and capacity tests whenever updates are made to production code. Use historical data to fine tune tests and to determine what types of tests need to be performed.

**Perform automated security penetration testing.** Ensuring your application is secure is as important as testing any other functionality. Make automated penetration testing a standard part of the build and deployment process. Schedule regular security tests and vulnerability scanning on deployed applications, monitoring for open ports, endpoints, and attacks. Automated testing does not remove the need for in-depth security reviews at regular intervals.

**Perform automated business continuity testing.** Develop tests for large scale business continuity, including backup recovery and failover. Set up automated processes to perform these tests regularly.
**Release**

**Automate deployments.** Automate deploying the application to test, staging, and production environments. Automation enables faster and more reliable deployments, and ensures consistent deployments to any supported environment. It removes the risk of human error caused by manual deployments. It also makes it easy to schedule releases for convenient times, to minimize any effects of potential downtime.

**Use continuous integration.** Continuous integration (CI) is the practice of merging all developer code into a central codebase on a regular schedule, and then automatically performing standard build and test processes. CI ensures that an entire team can work on a codebase at the same time without having conflicts. It also ensures that code defects are found as early as possible. Preferably, the CI process should run every time that code is committed or checked in. At the very least, it should run once per day.

Consider adopting a trunk based development model. In this model, developers commit to a single branch (the trunk). There is a requirement that commits never break the build. This model facilitates CI, because all feature work is done in the trunk, and any merge conflicts are resolved when the commit happens.

**Consider using continuous delivery.** Continuous delivery (CD) is the practice of ensuring that code is always ready to deploy, by automatically building, testing, and deploying code to production-like environments. Adding continuous delivery to create a full CI/CD pipeline will help you detect code defects as soon as possible, and ensures that properly tested updates can be released in a very short time.

Continuous *deployment* is an additional process that automatically takes any updates that have passed through the CI/CD pipeline and deploys them into production. Continuous deployment requires robust automatic testing and advanced process planning, and may not be appropriate for all teams.

**Make small incremental changes.** Large code changes have a greater potential to introduce bugs. Whenever possible, keep changes small. This limits the potential effects of each change, and makes it easier to understand and debug any issues.

**Control exposure to changes.** Make sure you're in control of when updates are visible to your end users. Consider using feature toggles to control when features are enabled for end users.

**Implement release management strategies to reduce deployment risk.** Deploying an application update to production always entails some risk. To minimize this risk, use strategies such as canary releases or blue-green deployments to deploy updates to a subset of users. Confirm the update works as expected, and then roll the update out to the rest of the system.

**Document all changes.** Minor updates and configuration changes can be a source of confusion and versioning conflict. Always keep a clear record of any changes, no matter how small. Log everything that changes, including patches applied, policy changes, and configuration changes. (Don't include sensitive data in these logs. For example, log that a credential was updated, and who made the change, but don't record the updated credentials.) The record of the changes should be visible to the entire team.

**Automate Deployments.** Automate all deployments, and have systems in place to detect any problems during rollout. Have a mitigation process for preserving the existing code and data in production, before the update replaces them in all production instances. Have an automated way to roll forward fixes or roll back changes.

**Consider making infrastructure immutable.** Immutable infrastructure is the principle that you shouldn't modify infrastructure after it's deployed to production. Otherwise, you can get into a state where ad hoc changes have been applied, making it hard to know exactly what changed. Immutable infrastructure works by replacing entire servers as part of any new deployment. This allows the code and the hosting environment to be tested and deployed as a block. Once deployed, infrastructure components aren't modified until the next build and deploy cycle.

**Monitoring**

**Make systems observable.** The operations team should always have clear visibility into the health and status of a system or service. Set up external health endpoints to monitor status, and ensure that applications are coded to instrument the operations metrics. Use a common and consistent schema that lets you correlate events across systems. Azure Diagnostics and Application Insights are the standard method of tracking the health and status of Azure resources. Microsoft Operation Management Suite also provides centralized monitoring and management for cloud or hybrid solutions.

**Aggregate and correlate logs and metrics**. A properly instrumented telemetry system will provide a large amount of raw performance data and event logs. Make sure that telemetry and log data is processed and correlated in a short period of time, so that operations staff always have an up-to-date picture of system health. Organize and display data in ways that give a cohesive view of any issues, so that whenever possible it's clear when events are related to one another.

Consult your corporate retention policy for requirements on how data is processed and how long it should be stored.

**Implement automated alerts and notifications.** Set up monitoring tools like Azure Monitor to detect patterns or conditions that indicate potential or current issues, and send alerts to the team members who can address the issues. Tune the alerts to avoid false positives.

**Monitor assets and resources for expirations.** Some resources and assets, such as certificates, expire after a given amount of time. Make sure to track which assets expire, when they expire, and what services or features depend on them. Use automated processes to monitor these assets. Notify the operations team before an asset expires, and escalate if expiration threatens to disrupt the application.

**Management**

**Automate operations tasks.** Manually handling repetitive operations processes is error-prone. Automate these tasks whenever possible to ensure consistent execution and quality. Code that implements the automation should be versioned in source control. As with any other code, automation tools must be tested.

**Take an infrastructure-as-code approach to provisioning.** Minimize the amount of manual configuration needed to provision resources. Instead, use scripts and Azure Resource Manager templates. Keep the scripts and templates in source control, like any other code you maintain.

**Consider using containers.** Containers provide a standard package-based interface for deploying applications. Using containers, an application is deployed using self-contained packages that include any software, dependencies, and files needed to run the application, which greatly simplifies the deployment process.

Containers also create an abstraction layer between the application and the underlying operating system, which provides consistency across environments. This abstraction can also isolate a container from other processes or applications running on a host.

**Implement resiliency and self-healing.** Resiliency is the ability of an application to recover from failures. Strategies for resiliency include retrying transient failures, and failing over to a secondary instance or even another region. For more information, see Designing resilient applications for Azure. Instrument your applications so that issues are reported immediately and you can manage outages or other system failures.

**Have an operations manual.** An operations manual or *runbook* documents the procedures and management information needed for operations staff to maintain a system. Also document any operations scenarios and mitigation plans that might come into play during a failure or other disruption to your service. Create this documentation during the development process, and keep it up to date afterwards. This is a living document, and should be reviewed, tested, and improved regularly.

Shared documentation is critical. Encourage team members to contribute and share knowledge. The entire team should have access to documents. Make it easy for anyone on the team to help keep documents updated.

**Document on-call procedures.** Make sure on-call duties, schedules, and procedures are documented and shared to all team members. Keep this information up-to-date at all times.

**Document escalation procedures for third-party dependencies.** If your application depends on external third-party services that you don't directly control, you must have a plan to deal with outages. Create documentation for your planned mitigation processes. Include support contacts and escalation paths.

**Use configuration management.** Configuration changes should be planned, visible to operations, and recorded. This could take the form of a configuration management database, or a configuration-as-code approach. Configuration should be audited regularly to ensure that what's expected is actually in place.

**Get an Azure support plan and understand the process.** Azure offers a number of support plans. Determine the right plan for your needs, and make sure the entire team knows how to use it. Team members should understand the details of the plan, how the support process works, and how to open a support ticket with Azure. If you are anticipating a high-scale event, Azure support can assist you with increasing your service limits. For more information, see the Azure Support FAQs.

**Follow least-privilege principles when granting access to resources.** Carefully manage access to resources. Access should be denied by default, unless a user is explicitly given access to a resource. Only grant a user access to what they need to complete their tasks. Track user permissions and perform regular security audits.

**Use role-based access control.** Assigning user accounts and access to resources should not be a manual process. Use Role-Based Access Control(RBAC) grant access based on Azure Active Directory identities and groups.

**Use a bug tracking system to track issues.** Without a good way to track issues, it's easy to miss items, duplicate work, or introduce additional problems. Don't rely on informal person-to-person communication to track the status of bugs. Use a bug tracking tool to record details about problems, assign resources to address them, and provide an audit trail of progress and status.

**Manage all resources in a change management system.** All aspects of your DevOps process should be included in a management and versioning system, so that changes can be easily tracked and audited. This includes code, infrastructure, configuration, documentation, and scripts. Treat all these types of resources as code throughout the test/build/review process.

**Use checklists.** Create operations checklists to ensure processes are followed. It’s common to miss something in a large manual, and following a checklist can force attention to details that might otherwise be overlooked. Maintain the checklists, and continually look for ways to automate tasks and streamline processes.

# Ultimate Checklist for Successful App Deployment

Deploying software releases is a mixture of planning, testing, late hours, and celebratory beers. This checklist is intended to be a guide to help improve how your team achieves this complicated and sometimes hairy task.

**Part 1: Planning and Deployment Preparation**

### *Collaboration Between Development and Operations*

It is important that operations be in the loop when it comes to planning any software deployment. They need to know when the project starts, not at the last minute about your new deployment. Ensure that operations is aware of any infrastructure changes needed well in advance. If the operations team is your front-line defense for monitoring and troubleshooting your app, help them help you by collaborating on how best to monitor the app and discuss early warning signs of trouble to be alerted to. If you're thinking of setting up DevOp team you should probably read this.

### *Build & Release Automation*

Removing the human factor is important to duplicating software deployment success. How you deploy the code doesn't have to be state of the art. Utilize a basic build server like Maven, Bamboo, or TFS to automate the process or even a simple script that copies the files across the network can be sometimes sufficient. The key here is to make it easily repeatable by anyone on the team and eliminate the possibility of skipping any manual steps.

### *Minimize the Amount of Change*

When things break in IT, typically it is because  something changed. When you introduce small changes in each software update it is easier to roll back those changes, or know exactly where to look for the source of the problem. As each release takes a certain amount of work in planning and testing, the key is finding the right balance of frequency and size of your releases that is optimal for your team to handle.

### *Create and Test SQL Change Scripts*

Prior to your app deployment make sure you use tools like Red Gate SQL Compareto know what SQL schema and data changes have to be moved to production. to ensure there are no environmental differences that will cause potential headaches, I recommend never to forget to take a backup of production and run the test scripts on the backup copy before the final release. Production data differences or different database software versions can commonly cause hiccups. Also, have a plan to roll back to a previous state if the release has issues. Code is easy to roll back, but SQL requires some planning.

### *Know your KPIs*

Every app should have some form of key performance indicators or metrics that can be used to know if everything is running smoothly. These will vary wildly from one app to another. They can be as simple as server CPU, page load times, or database performance metrics. The best metrics are more custom though. Knowing how many orders are happening per minute, or how many messages are being processed off a queue and how long it takes, or how long a certain piece of code takes to execute is a critical part of your app. Make your KPIs the heartbeat of your application and monitor them at all times.

### *Setup Synthetic Transactions Tests*

Create a few tests that test things like your login page, key pages within your apps, APIs, etc. This will allow you to feel very confident that all systems are operational after the app deployment. I highly recommend Selenium, the leading open source framework for web automation.

### *Notify Your Users and Colleagues*

Don't forget to notify your users and others within your company about the upcoming changes. Make sure you have time to update documentation, train internal staff about new features, and coordinate with other departments in your company.

## Part 2: After the Deployment

### *Monitor Server Utilization*

Make sure servers are all online, in the load balancer, and still receiving traffic. Monitor server and app CPU, memory, network, and disk usage. *N*outilization can be as bad as sudden *high* utilization.

### *Monitor Exception Rate*

After a new release your software is likely to throw some new exceptions you have never seen. Utilize an error tracking program so you can immediately identify new errors introduced in the release and to ensure previously resolved errors haven't been reintroduced. A comparison of your overall error rates before and after the deployment will give you good indication of the release health.

### *Check your KPIs*

Keep an eye on your performance indicators that let you know if your application is running properly.

### *Watch your logs*

After an app deployment make sure logging is still working and the volume of logging hasn't changed dramatically. No logging at all can be as bad as sudden high logging rates. A centralized log management tool makes this easy to monitor. You might also want to check the logs of new features that were released to ensure they are behaving properly.

***Monitor Page Load Times and HTTP Error Rates***

Watch the overall page load times of your application and keep a closer eye on 2-3 requests that are mission critical to your app. Also watch out for the rate of 400 and 500 level HTTP requests to ensure they haven't dramatically changed.

***Monitor Database Performance***

After your deployment make sure database CPU, IO, and overall traffic look normal. Nothing is scarier than a sudden drop in database traffic after a release… unless it's a sudden spike in database traffic.

***Monitor Key Database Queries***

Setup monitors in your monitoring system to test key queries to ensure they are loading quickly and returning proper results. For example, if some important background processes work based on SQL queries, run similar queries to make sure it isn't missing work to be done and is working properly.

***Monitor Application Queues***

Monitor how many messages are in your application queues to ensure everything is flowing properly and not getting backed up.

***Cache Server Changes***

Don't forget about caching. Depending on how you use caching, you may need to clear your caches or make other configuration changes. Data type changes can commonly cause some weird serialization type issues in cache.

.

***Post Mortem Meeting***

It can be very valuable to stop for a few minutes and discuss how the last release went and how things potentially can be improved. These kinds of conversations can help identify team weaknesses, allow people to vent, and take note of issues that can try to be prevented in future development cycles.

# DevOps Checklist

## What is DevOps?

### The Dilemma

- Operations is the longest and most important phase in an applications life cycle.
- But requirements for this phase get the least attention.
- Operational Requirements are the step children of requirements engineering and development.

### The Answer

**DevOps** brings **operational competency** into the development team to facilitate the delivery of easy to operate, reliable and resilient applications.

### Examples

| The usual Situation without DevOps | The Benefit - How DevOps addresses those issues: |
|---|---|
| Complex, manual installation process requires high deployment effort on each test and production system. | **Fully scripted installation**. DevOps staff is highly skilled in using professional tools to facilitate fully automated deployment.<br>This eliminates human errors, reduces planned downtime for upgrades and facilitates Continuous Integration and Continuous Delivery. |
| The system is very difficult to operate | The DevOps member takes care that within<br>• Requirements specification phase (Waterfall) or<br>• Sprint Planning Meeting #2 (Agile)<br>the **non-functional requirements for operability** get sufficient attention in the specifications. |
| Troubleshooting is difficult, because of insufficient log file information and trace functionalities | DevOps member will support developers in implement sufficient looging- and tracing functionalities, e.g. by using logging frameworks. |
| At overload or other disturbances application crashes or hangs up, requiring restart | DevOps will help to **build resilience into the applications code**. |
| It is impossible to meet SLA's desired by business with application as delivered. | DevOps member will take care that SLA's are known before, and that operational requirements are specified and implemented such that the SLA's can be met. |
| No or incomplete **Operations Manual (Runbook)** | The Operations Manual (Runbook) is written during development. Advanced DevOps level will facilitate Runbook driven Development. |

| | |
|---|---|
| Only Server, Filesystems and Database are being monitored, but no application specific monitoring. | DevOps member will take care that **application specific monitors** are being developed as part of normal delivery process. |
| A significant amount of money is spent on redundant hardware for failover, but application does not achieve transparent failover. | DevOps members understand HA / DR Technologies, will participate in the selection process and ensure that application code will utilize the selected HA / DR Technology. *"Building Resilience into the application code."* |

## DevOps Checklists and Templates

IT-Checklists > DevOps > DevOps Checklists

The following Checklists and Templates support DevOps Experts and increase the efficiency of their daily activities:

1. Requirements Checklist
2. Runbook Template
3. SLA Templates
4. Deployment Checklist
5. Application Level Monitoring

## Operability Requirements Checklist

**Operability Requirements** are the major group within the group of **Non-Functional Requirements**, often referred as **“Quality Attributes”**.

The purpose of those requirements is to ensure that an application / an IT system is easy and efficient to operate with high availability and reliability.

Types of **Operability Requirements** are e.g.

- Monitoring Requirements
- Robustness and Resilience Requirements
- Fast Troubleshooting
- Performance
- Scalability
- Redundant Hardware / Cluster

## Operations Manual / Runbook Template

This template for an ***IT Operations Manual / Systems-Handbook*** will help you

1. that no important item is forgotten and
2. a common handbook structure is used throughout all systems. This will support easy cross-system troubleshooting and documentation.

The resulting Manual / Runbook is an important deliverable of the overall IT system for

- **compliance with documentation-requirements** for systems and processes required by internal QA-department or internal auditing department or external auditors or other organizations and laws

- There are no doubts that an Application Runbook needs to be delivered.
- The only question is: When to start writing it?.

**DevOps** recommends that the operations manual is written during development.

The concept of "Runbook Driven Development" recommends to write it even before writing the test cases and implementation:
Within an agile development team that includes an DevOps expert,

- the architect has the clear vision how the functionalities being developed should work, and how to operate those
- the developers and testers gain deep understanding during the analysis and specification of backlog items being worked on
- the tester, participating in analysis and specifications gains the knowledge how to test the backlog item

Conclusion: The team knows all details to be documented in the Operations Manual!
Question: Why to delay writing that chapter of the Operations Manual, resulting later in additional work to explain an document writer what he should write?

Recommendation: Go one step further! - Use the principle of TDD (Test Driven Development) for writing the Operations Manual!

## SLA Templates

**SLA's** can be that tight, that the application needs to be optimized to get a chance to meet those SLA's.

For that reason the SLA's need be known during requirements anlysis!

- Requirements specification phase (Waterfall) or
- Sprint Planning Meeting #2 (Agile)

## Application Level Monitoring

Experienced DevOps Experts already know that application specific monitoring doesn't come "out of the box" when purchasing a standard monitoring framework.

Those monitoring frameworks do offer modules for

- server,
- operating systems and
- databases

Special **application level** monitoring modules are being offered for only a very limited number of very popular applications.

For most applications monitoring scripts / API's - which **CAN** be utilized from monitoring frameworks - need to be implemented within the project.

# The 15-point DevOps Check List

DevOps is a culture that requires some practices and a new vision, its common goal is unifying people and organizations around unique goals.

The 15-point DevOps Check List is taken from a technical book (The Jumpstart Up) that I am actually writing to provide a practical handbook and experience-based guide focused on the application to lead DevOps and Agile transformation.

The DevOps Checklist is neither static nor unique, there is no manifesto that describes DevOps, but it should be adapted to the organization need, human interactions and other specific criteria. In other words, the checklist could help you proceed with setting up a DevOps culture but don't consider it as a unique way to proceed with your organization transformation.

These points are cultural, process-related or technical. In all cases, something I had never or rarely found in other checklists which is reliability not just your live environments but also processes and the first thing reliability requires is the simplicity. Simplicity is not an element in the following checklist because it must demonstrate in each of these points, so keep thing simple.

## A Cross-Functional Team

A cross-functional team is a group of people with different functional expertise (marketing, operations, development, QA, account managers ..etc) working for the same goals and projects.

A group of individuals of various backgrounds and expertises are assembled to collaborate in better manners and solve problems faster.

As said in Wikipedia: The growth of self-directed cross-functional teams has influenced decision-making processes and organizational structures. Although management theory likes to propound that every type of organizational structure needs to make strategic, tactical, and operational decisions, new procedures have started to emerge that work best with teams.

In DevOps context, the dev and ops teams should not live in separate silos. Each team should provide support and advices in order to take advantage of the skills of everyone.

According to some management studies, like Peter Drucker's on management by objectives, cross-functional teams are less goal dominated and less unidirectional which stimulates the productivity and the capability of dealing with fuzzy logic.

## Communication Culture & Global Thinking

When working together on the same products, communication is bound in achieve better results and reach valued goals. DevOps is mainly a culture of communication and cross-functional collaboration. Individuals and departments could speak in different professional languages, which creates different types of communication types, like it is the case between developers and operation teams. Every team has its own goals, operation engineers seek stability, while developers tend to make changes that may affect the stability in some cases. This is not just the case for developers and operation engineers, every department in a non-DevOps environment has its own goals and give almost 100% of the effort and the time to achieve it.

Obviously with different goals, different responsibilities and different professional languages, communication becomes difficult. From that point departments will throw responsibilities of problems that they either create or encounter.

Being rigid in setting goals for each department is not a good idea in many cases. Having common goals, goals shared between departments and make managers, executives and all members of a team aware about the fact that there are common goals can reduce this gap between two or more teams.

Setting up departmental and local goals brings up the “not-my-job” problem where each department throw responsibilities on another one, while establishing common and global goals encourage people to work together.

The following hacks could help your organization to ameliorate communication:

### *Motivate through Gamification*

Gamification is a good way to keep your team motivated while playing.

In my work we are using Hipchat and I integrated several chat bots but the one that I like the most is the Karma bot: every one in my team instead of saying “Thank you”, can give one or several karma points to another colleague.

This is somehow a kind of building a symbolic “meritocracy”. The GNOME Foundation, Apache Software Foundation, Mozilla Foundation, and The Document Foundation are examples of (open source) organizations that officially claim to be meritocracies.

You can find other tools that could help in the gamification of work spaces and the daily professional life like Game Effective to gamify sales, customer service and employee training.

### *The Smiley Board*

At the end of each work day, everyone is required to draw a picture of their face in one of three modes:

- Happy face
- Blah face

- Sad face

The Smiley Board is also called the Niko-niko Calendar (or Smiley Calendar). It is a Japanese creation where each member shall put a smiley on his own schedule at the end of the day, before leaving the office. This gives a view about the well-being and motivation of each member of the project.

### *Open & shared spaces*

Open spaces done right are a key to valuable communications and collaboration.

### *Chat Rooms*

Internal chat applications are widely used in many organizations, generally chat rooms are specific to a team or a project. Adding chatbots in tools like Hipchat, Slack or any of their alternatives, helps teams to communicate on several recurring events and being transparent and aware about:

- Deployments
- Incidents
- Builds
- Developers' commit to the versioning server

In my company I have even set up bots to post daily motivational quotes, we have also some chat rooms where we talk about non-work related stuff.

### *Hackathons*

A hackathon, hackfest, codefest or hackday is an event in which individuals involved in software or hardware development, design, UX, project management, collaborate intensively on software/hardware projects. Hackathons tend to have a specific focus on a technology, a theme or a project but some hackathons are open and participants have the full freedom. Hackathons are a great way to build communications and allow people from the same organization to work together and have the same goal. Internal Hackathons are also a way that some companies like Netflix, Facebook, Google, Microsoft, Hewlett Packard organize to promote new product innovation by the engineering staff. For example, Facebook's Like button was conceived during a hackathon.

### *Team building*

Or helping a team to make it more efficient in terms of operability, more cohesive, consistent in its results and competent. This type of accompaniment is particularly relevant when the professional situation is internally complex during a process of reorganization or profound change in the business, or when the situation is externally complex during a competitive pressure or a changing market. It helps the team to co-build the solution and develop their collective intelligence and autonomy.

### *Playing Games*

Game playing is a good way to create a good communication behavior. A good example is the Kanban simulation game simulates variable work flow for a SaaS company The getKanban Board Game is a physical game designed to teach the concepts and mechanics of Kanban for software development in a class or workshop setting.

### *Outlets, group outing, Friday drinks*

---

## Customer-Oriented Culture

Product-centric culture will not work in most cases, especially if you are a startup with an MVP or even more and trying to be competitive in the market. No one know how a product should be better than the customer himself.

To build a customer-oriented culture, you should:

– Not build the best product but create the best solution for your customer.
– Stop worrying about creating new products or features and instead of that search for your customers' new needs to fill.
– Hire people who fit and reward people having deep insights into customers instead of rewarding new features or product development.
– Share your customers needs explicitly with your developers and operation engineers. Being transparent about business needs and customers needs is the first step to create a customer-oriented culture.

Actually, being focused on customers is the best way to align teams towards the same valuable goals without creating an inter-department war.
The DevOps feedback loop is also a good way to keep your systems stable and scalable in the same time, customers need this: stability and scalability.

---

## Source Control & Revisions

According to Wikipedia
*A component of software configuration management, version control, also known as revision control or source control, is the management of changes to documents, computer programs, large web sites, and other collections of information.*

Source control can be critical to your success.

When developing a digital product, it is important to give developers a technical tool equivalent to project management tools but from a pure technical view. Source control was created to resolve real problems that developers encountered during the coding process. It allows them to keep a manage:

- The code modification history, so that change management and getting back to older versions of a working code could be easier.
- Concurrent file editing, in the case when multiple developers works on the same code.
- Tagging
- Branching
- Merging

Version control is not just for developers since one of the best practices in startups is versioning documentations. Documentation versioning will help you:

- Track incremental backups and recover
- Record any change and revert a documentation to an earlier version
- Track co-authoring and collaboration and individual contributions

Most of the modern documentation softwares use versioning like Atlassian Confluence or other open source softwares like wiki softwares (MediaWiki, DocuWiki ..etc).

---

## Infrastructure As Code

Infrastructure management and provisioning is moving to the next big thing: Infrastructure As Code.

Infrastructure as Code (IaC) is the usage of definition and configuration files to create, start, stop, delete, terminate and restart virtual or bare-metal machines. When mastering IaC organizations can reduce costs and time of infrastructure management in order to focus more on the product development.

With the rise of DevOps movement the fact of enabling the Continuous Configuration Automation approach is becoming a key step in the life cycle of a product.

I published a post on medium where I explain how Infrastructure As Code can work using a configuration management tool.

## Routine Automation

The DevOps philosophy could be described in different manners, but I saw once on Twitter a good point of view about defining DevOps from an automation perspective: “Using Things You Can Program, and Programing the Things You Use”.

Automation in the DevOps philosophy is about making faster tasks in order to interface things and create automated pipelines. Almost everthing could be done manually, but in order to focus on product development and to create a continuous pipelines (continuous integration, delivery, testing, deployment ..etc) and feedback loops, everything starts with automation:

- Automate infrastructure
- Automate integration
- Automate delivery
- Automate feedbacks
- Automate scalability
- Automate bugs hunting
- ..etc

The continuous processes relies on already automated tasks.

---

## Self-Service Configuration

Using cloud technologies with configuration management software allows automated provisioning of infrastructure and services. Generally the operation engineers after listening to developers need for product development, install and configure software on production-like servers. Using tools like Chef, Puppet, Ansible or SaltStack, cloud infrastructure like AWS and Digital Ocean, versioning systems like Github or Bitbucket, containers technologies like Docker, continuous integration and deployment servers like Jenkins or Rundeck.

- You system configuration should be always into a source control service/servers
- Developers should be able to create systems with production-like configurations and data.
- Developers should have access to continuous integration tasks to build softwares and test artifacts in a short period: I prefer working with git hooks, where an automatic build is launched right when a developer push a change to development branch.

- In a stage of maturity, Developers could be able to deploy a change to production: This can be complex to achieve, in some cases it is preferred to keep operation engineers deploy to production.

---

## Automated Builds

With the increasing number of Jenkins plugin, automating builds is becoming easier. An example For web applications is using automated build tools like Ant with package managers like npm and dependencies managers like php composer.

The main types of automated builds are:

- On-demand automated builds where the user run a script to launch the build if it is needed: This is not really fully automated and it is used in the cases where scheduled and triggered builds are complex or useless.
- Scheduled automated builds like it is the case with the continuous integration servers (such as Jenkins) running nightly builds
- Triggered automated builds where builds in a continuous integration server are launched just after commit to a git repository.

---

## Continuous Integration

Continuous integration use automated build to create a process where the integration server build a project if any changes were made or periodically. The process could also include automated tests and automated delivery.

Continuous integration is an important brick in the DevOps settlement and the weak link in the automation process since it is positioned between development and operations in order to automate the flow and fluidify the passage of an application from development to post-development operations.

---

## Continuous Delivery

Continuous delivery is a use both the continuous integration and automated builds to deliver software to other teams as fast as possible, ideally after a code change. The product is then delivered in an artifact server or at least a FTP server. As an example, QA teams cloud access to the last delivery to do their tests.

If the transition from the test phase to the production deployment phase is automatic, we call this continuous deployment. The big difference between continuous is the automated deployment.

A stable continuous delivery is a sign of success for a great part of a DevOps journey. If you already implemented this in your DevOps pipeline, you are then mastering the DevOps art.

---

## Incremental Testing & Test-Driven Development

Testing relies on the concepts of integration testing and incremental testing relies on the continuous integration and delivery.

Incremental testing is continuous and repetitive as new and fresh functionality is added.

The incremental approach has the advantage of detecting fresh defect. Because finding bugs in an early stage will help your organization spend less money and stabilize production environments, incremental testing is one the best practices in DevOps.

Different testing approaches could be adopted:

- Top down: Testing takes place from top to bottom, following the control flow. Example: starting from the GUI to the program core.
- Bottom up: Testing takes place from the bottom to top following . Example: starting from the system components to the GUI.
- Functional incremental: Testing functions and functionalities like described in the functional specification.

Test-driven development is one of the concepts of the extreme programming( a software development methodology which is intended to improve software quality and responsiveness and one of the agile software development practices).

TDD is one of the best practices to adopt in a DevOps culture, it is based on incremental testing and the repetition of a very short development cycle. Kent Beck, who is credited with having developed or rediscovered the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.

Test-driven development is a test-first concept that developers can apply in order to improve and debug their code or even legacy code developed with older techniques and methodologies.

---

## Automated release management

Product development is not just code. The product has a whole life cycle, from development, version control, builds, repositories and artifact delivery, tests and acceptance, server provisioning, application configuration to production deployment. This cycle describes the release management, while automation is one of the pillars of the DevOps culture, the release management should be automated for better business results.
The automation of release management relies on automating all of its stages, that is why automating releases require setting up a continuous delivery strategy.

---

## Shorter Development Cycles & Time-To-Market

The motivation of being customer-oriented organization while using the techniques mentioned above will induce development cycle length and in a result the time-to-market.

Time-to-market (TTM) is the length of time it takes from the conception stage to the product being deployed and running in production servers. TTM is important in all industries where products are outmoded quickly as it happens in organization working on web applications, SaaS or PaaS products. In order to know the existent and improve it, measuring the time-to-market is a good practice, one of the simplest approaches to measure it is counting days from the conception of a feature to the the deployment of the stable release containing the feature to the production.

Agile methodologies and DevOps culture advocates working in developments sprint, while routine tasks are automated, integration and tests are continuous, a sprint may take from one to two weeks and so the time-to-market.

---

## Key Performance Indicators

In my actual job, I am leading a DevOps transformation. It could be painful sometimes, but measuring your success is motivating and rewarding. Since my first month, I identified major problems and picked out some indicators, the first thing I have done was measuring, after ten month (almost a year), I started gathering the same indicators to measure. Well things evolved :

- Uptime vs downtime
- Errors ratio
- Responsiveness
- Reject ratio
- Load capacity
- Many other specific indicators

And of course the time-to-market.

As said above, measuring the time-to-market is a good practice—it is one of the business-oriented indicators. In order to reduce the length of a product TTM, we should measure its actual one and this is similar for other improvements.

Well, key performance indicators (KP) are the key-indicators of the efficiency, performance and the good result. It can be collective or personal and to promote sharing global global goals, departmental KPIs are less important than global KPIs.

A key performance indicator can meet the following objectives:

- evaluation
- diagnostic
- communication
- information
- motivation
- continuous progress

Indicators are a key-concept in the continuous improvements and it relies on:

- Choosing your indicators, rather choosing the good indicators—well it simply depends on your organization specificities, priorities and goals.
- Automating measurements: Working on sprints of improvements is one of the best practices. Improvements are not a stage or a list of tasks to achieve after deploying a product, but a continuous work that require a continuous measurement that could be automated. Having the number of errors and bugs happening in your live or in your integration environments is a good example. I always considered having a screen with live KPI measurements in an open space is genius idea.

## The DevOps Feedback Loop

To achieve an incremental work flow, a constant, feedback loop between operation and software development teams is important. The feedback is the description of the current state of the software across its life cycle.

The feedback loops, is one of the most important DevOps principles, it can describe the DevOps and the Lean methodology goals: Continuous improvements while having a fail-fast workflow between development and IT.

This feedback could be a real time flow in its most advanced forms.

Monitoring, log gathering and analysing is a practice among others forming this feedback loop and if you are used to this kind of practices, you will understand that this is about communication and transparency between two teams : dev and ops.

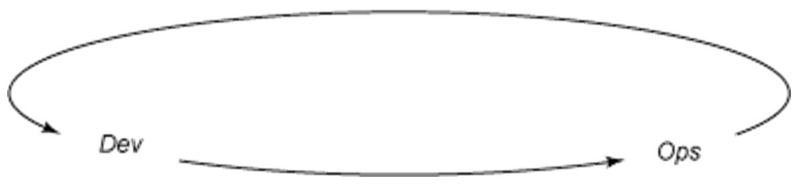
In startups, change is a law so the high deployment rates will often result a supercharge for IT operations. Clyde Logue, founder of StreamStep, said: “Agile was instrumental in development regaining the trust in the business, but it unintentionally left IT operations behind. DevOps is a way for the business to regain trust in the entire IT organization as a whole.”

“The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win” book describes the fundamental principles of DevOps, describes “The Three Ways” from which are the principles that all of the DevOps patterns can be derived from.

### *Systems Thinking*

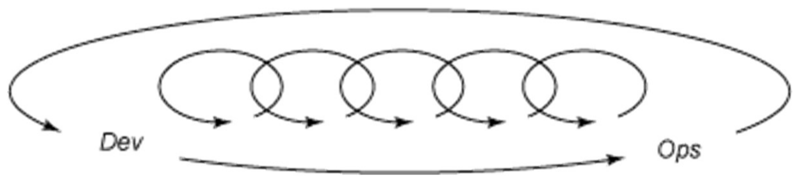
System thinking highlights in the first place the performance of the entire system so that no more departmental goals will be considered as more important as the global goals. The focus is on all business value streams that are enabled by IT.

## *Amplifying Feedback Loops*



This way is about creating a feedback loop in order to achieve improvements while amplifying the loops and this helps understanding customers needs and responding to all the the internal needs (both developers and system administrators ..etc). Amplifying the feedback loop is a way to increase the global degradation caused by local changes. Some local changes are just optimizations intended to achieve an individual or a local goal and putting the amplified feedback in work will never pass a defect to downstream work centers.

## *Culture Of Continual Experimentation And Learning*



This way is about continuous experimentation, taking risks and continuous learning. The repetition in this way is the prerequisite to mastery and the experimentation is a path towards improvements which is daily work. Feedback could be done in a shorter time.