


# DevOps Checklist

01/10/2018 • 14 minutes to read • Contributors 

## In this article

[Culture](#)

[Development](#)

[Testing](#)

[Release](#)

[Monitoring](#)

[Management](#)

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 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 helps 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 reliable Azure applications](#) . 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.

For more about DevOps, see [What is DevOps?](#) on the Visual Studio site.