Defining DevOps

Build Your DevOps Practice on GitHub

WRITTEN BY GITHUB WITH ♥

What's inside

WRITTEN BY GITHUB WITH ♥

Introduction to DevOps

From headlines to job descriptions, DevOps has emerged as an outsized buzzword over the past decade–and for good reason.

Organizations that successfully adopt DevOps often see big gains in software development speeds, improved reliability, faster product iterations, and have an easier time scaling their services.

But despite its roots in software development, DevOps is a holistic business practice that combines people, processes, cultural practices, and technologies to bring previously siloed teams together to deliver speed, value, and quality across the **software development life cycle** (**SDLC**).

DevOps is a holistic business practice that combines people, processes, cultural practices, and technologies.

That means there's often no one-size-fits-all approach. But there is a common set of practices and principles in any successful DevOps implementation. This e-book outlines core DevOps fundamentals and principles, as well as how to implement a DevOps pipeline in an organization to deliver increased value to customers. Guidance is then provided for the tools that best complement an organization's DevOps practice.

GitHub approaches DevOps as a philosophy and set of practices that bring development, IT operations, and security teams together to build, test, iterate, and provide regular feedback throughout the SDLC.

DevOps fundamentals

DevOps helps teams ship high-quality products faster by reducing the friction between writing, testing, and deploying code.

GitHub offers a holistic platform designed to help organizations successfully adopt DevOps, making it easier to continuously ship and improve software.

People often ask what a DevOps model is—but this misses the point of DevOps. DevOps is an approach to building software that touches the entire development life cycle. It's a mix of practices, cultures, and technologies intended to continuously deliver value to end users.

In short, there is no one-size-fits-all approach to DevOps. Its implementation varies from organization to organization. Despite this, DevOps does have a framework of practices that all organizations leverage in varying forms.

At the core of DevOps is the idea that everyone responsible for a product should collaborate as a unified team. Rather than work in separate development, quality assurance, security, and operations silos, DevOps brings people together to take end-to-end responsibility for planning, building, delivering, and improving software.

Compared to traditional development methods where programming teams write code, testing teams find bugs, and operations teams take care of the infrastructure, DevOps can seem like a radical change. As a practice, DevOps fundamentally seeks to transform organizations by bringing traditionally siloed teams together across every part of the SDLC.

Even though every organization's DevOps adoption journey is unique, these are key principles that indicate success. If you do these things you're doing DevOps well—but depending on your industry, you'll have things that are particular and necessary to your DevOps practice.

DevOps principles defined

The implementation of DevOps will look different in every organization. GitHub believes it's best to understand DevOps as a framework for thinking about how to deliver value through software. It's more than a single methodology or collection of processes. It's fundamentally a set of principles—both cultural and technological. Let's break that down:

- **Collaboration:** To succeed, DevOps requires a close working relationship between operations and development—two teams that were historically siloed from one another. By having these teams collaborate closely under a DevOps model, you seek to encourage communication and partnership between these teams to improve your ability to develop, test, operate, deploy, monitor, and iterate upon your application and software stack.

- **Version control:** Version control is an integral part of DevOps—and most software development these days, too. A version control system is designed to automatically record file changes and preserve records of previous file versions.

- **Automation:** Automation in DevOps commonly means leveraging technology and scripts to create feedback loops between those responsible for maintaining and scaling the underlying infrastructure and those responsible for building the core software. From helping to scale environments to creating software builds and orchestrating tests, automation in DevOps can take on a variety of different forms.

- **Incremental releases:** Incremental releases are a mainstay of successful DevOps practices and are defined by rapidly shipping small changes and updates based on the previous functionality. Instead of updating a whole application across the board, incremental releases mean development teams can quickly integrate smaller changes into the main branch, test them for quality and security, and then ship them to end users.

- **Orchestration:** Orchestration refers to a set of automated tasks that are built into a single workflow to solve a group of functions such as managing containers, launching a new web server, changing a database entry, and integrating a web application. More simply, orchestration helps configure, manage, and coordinate the infrastructure requirements an application needs to run effectively.

- **Pipeline:** In any conversation about DevOps, you're likely to hear the term pipeline thrown around fairly regularly. In the simplest terms, a DevOps pipeline is a process that leverages automation and a number of tools to enable developers to quickly ship their code to a testing environment. The operations and development teams will then test that code to detect any security issues or bugs before deploying it to production.

- **Feedback sharing (or feedback loops):** Feedback sharing—or feedback loops—is a common DevOps term first defined in the seminal book The Phoenix Project by Gene Kim. Kim explains it this way: "The goal of almost any process improvement initiative is to shorten and amplify feedback loops so necessary corrections can be continually made." In simple terms, a feedback loop is a process for monitoring application and infrastructure performance for potential issues or bugs and tracking end-user activity within the application itself.

These DevOps principles are paramount for building a successful DevOps practice. Successful, high-functioning DevOps practices exhibit the following characteristics and benefits:

- Faster delivery and release cycles
- More automation and increased productivity
- Increased quality through collaboration
- More scalable products
- Improved process scalability, where continuous measurement drives continuous improvement
- Greater resilience in applications, infrastructure, and teams

These benefits translate to the ultimate goal of DevOps, which is to bring increased value to customers and more productivity, as well as collaboration between teams.

Key stages in the DevOps adoption journey

When building your organization's DevOps practice, understand that it's a journey, not a destination. Each journey will vary depending on what the organization needs. Organizations often start by implementing DevOps practices at a small scale while evolving and becoming more proficient over time. The stages in the table below show a typical evolution of an organization—from Experimental DevOps where only a few teams are practicing DevOps to Native DevOps where DevOps is adopted across the whole organization, silos are broken down and there's an easy flow of information between teams.

Our philosophy is to build automation and great DevOps for the company you will be tomorrow."

Todd O'Connor
Senior SCM Engineer at Adobe

Experimental DevOps	Learned DevOps	Proactive DevOps	Native DevOps
One or two teams are exploring DevOps. Role-based silos still largely in place. Some experimentation with automation but manual intervention needed for each step. No formal process.	Some parts of the organization have adopted collaborative product teams. Those teams are using DevOps tooling to good effect, but each team has its own approach. Process is forming and largely learned from what other organizations are doing.	All new products start out under the DevOps model. Measurements are in place to monitor process effectiveness and feed into improvements. Process is starting to become adapted to the needs of the organization. Most of the organization is using DevOps tooling.	DevOps is adopted across the organization. The DevOps process is tuned precisely to the organization's needs, with regular updates as circumstances change. Tests, builds, and deployments are automated using DevOps tooling. All teams are product-focused, with an easy flow of communication and collaboration across the entire organization.

For a DevOps practice to reach its full potential, it requires buy-in from everyone in the organization. However, changes can still be affected at a smaller scale within individual product teams. This is reflected in the Experimental DevOps stage. Often, other areas of the organization notice the successes of the teams practicing DevOps and want to replicate what they are doing. This is reflected in the Learned DevOps stage. Organizations may then reach a point where all new product development is following the DevOps model. In this stage, the development and operations teams are in sync to form product-focused teams. This is reflected in the Proactive DevOps stage. The Native DevOps stage is the objective. Here, the process is well defined, and communication and collaboration flow easily across the organization. In the Native DevOps stage, everyone in the organization is working together to deliver increased value to customers.

Foundational practices

How DevOps works varies from one company to the next. But there are three core themes you'll find in every organization that successfully adopts DevOps.

Everyone is responsible for quality

DevOps reduces the barriers between the disciplines found in software development teams. Practitioners tend to focus on building end-to-end products instead of completing siloed, incremental projects. This means the same individual will collaborate across the full SDLC, from planning to building to testing and deploying a product.

Code ships when it's ready

Traditional software development practices often bundle many changes into large releases. This means customers typically wait longer for software updates. It also makes it harder to predict the knock-on effect big code changes will have, putting greater pressure on development and operations teams. In contrast, DevOps favors incremental code changes that are easier to build and test—and to ship as soon as they are ready. Once a developer commits code changes to a project, **continuous integration and deployment** (**CI/CD**) tools facilitate automated tests, application builds, and code integration or issue reporting. Many DevOps practitioners extend the concept of continuous improvement to their own work, measuring and adjusting their processes over time.

Automation improves quality and predictability

In a successful DevOps practice, anything that can be automated should be automated. This reduces the risk of human error and makes products easier to scale. Tools are used to automate the configuration and deployment of infrastructure, while static analysis tools find and highlight security vulnerabilities. DevOps practitioners strive to automate repetitive tasks at every stage.

DevOps maturity

Introducing DevOps to your organization is an ongoing journey with different levels of adoption across the many different stages of product delivery. DevOps is as dynamic as a business needs it to be, and its implementation varies from organization to organization.

This means there isn't one defined DevOps maturity model. At GitHub, we shy away from talking about DevOps maturity models because it implies there's a checklist any organization can use to achieve "DevOps." This isn't true. At its core, DevOps is an ongoing practice. However, there are common steps and markers of success businesses can work toward, as shown in the following figure.

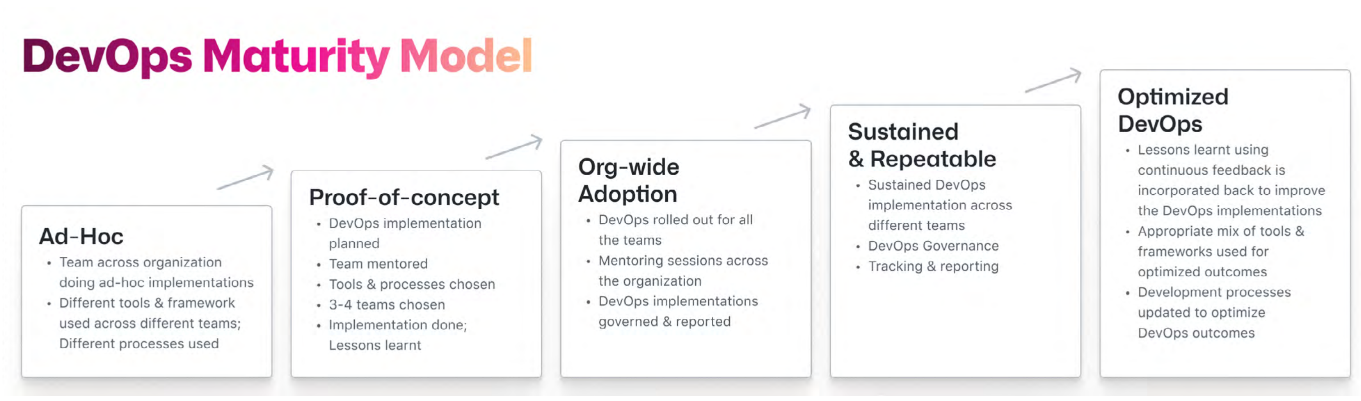
DevOps Maturity Model
Ad-Hoc
Team across organization doing ad-hoc implementations
Different tools & framework used across different teams; Different processes used
Proof-of-concept
DevOps implementation planned
Team mentored
Tools & processes chosen
3-4 teams chosen
Implementation done; Lessons learnt
Org-wide Adoption
DevOps rolled out for all the teams
Mentoring sessions across the organization
DevOps implementations governed & reported
Sustained & Repeatable
Sustained DevOps implementation across different teams
DevOps Governance
Tracking & reporting
Optimized DevOps
Lessons learnt using continuous feedback is incorporated back to improve the DevOps implementations
Appropriate mix of tools & frameworks used for optimized outcomes
Development processes updated to optimize DevOps outcomes

Figure 1: The DevOps maturity model

This DevOps maturity model diagram should be used to roughly plot where an organization is currently at and what it takes to reach the next level. Whichever stage of the maturity model an organization is currently at, the key concept is promoting an environment that fosters collaboration, continuous learning, and iterative improvement.

The next section introduces the idea of the DevOps pipeline and the tools and practices that make up each stage. Keep in mind the definition of DevOps—bringing together people, processes, cultural practices, and technologies in software development. It's important to note that technology is mentioned last and is only one piece of the overall picture. Technology can certainly help influence and optimize your DevOps practices, but people and culture are at the heart. An organization could have state-of-the-art DevOps tools, but a collaborative culture is required for the benefits of DevOps to be fully realized.

"The mindset we carry within our team is that we always want to automate ourselves into a better job."

Andrew Mulholland // Director of Engineering at Buzzfeed

The DevOps pipeline

A DevOps pipeline is a combination of automation, tools, and practices across the SDLC to facilitate the development and deployment of software into the hands of end users.

Critically, there is no one-size-fits-all approach to building a DevOps pipeline and they often vary in design and implementation from one organization to another. Most DevOps pipelines, however, involve automation, CI/CD, automated testing, reporting, and monitoring.

Important concepts in any successful DevOps pipeline are that it's repeatable, continuous, and always on. Nothing in a DevOps pipeline should be an isolated event, but instead comprise a larger system where each step is defined by its repeatability.

Importantly, building a DevOps pipeline is often one of the most tangible elements for organizations looking to adopt DevOps, which is defined as much by its cultural dimension that favors deep collaboration as it is by automation and specific tooling.

With the right technology and investments in people and processes, any organization can build an always-on DevOps pipeline—even if it's a simple one to start with. But without fully adopting a DevOps culture that prioritizes incremental development work and deep, cross-functional collaboration across the SDLC, organizations are unlikely to realize the full value of a DevOps pipeline.

“We have a slogan on our team: don't let a human do a machine's job. GitHub helps us achieve that.”

Gabriel Kohen
Principal Software Engineer at Blue Yonder

Stages of the DevOps pipeline

A common way people often explain a DevOps pipeline is by comparison to an assembly line. Each part of the SDLC is analyzed to establish a consistent set of automated and manual processes. The result is improved efficiency and consistency in terms of the overall output.

But unlike an assembly line, DevOps isn't an end-to-end process with a definite beginning and end. Instead, DevOps is a cycle of continuous improvement where even after software is shipped, improvement continues.

In practice, that means that even as a new software feature might take a linear path through stages of development, the overall system (and even that feature) goes through a continuous cycle of iteration.

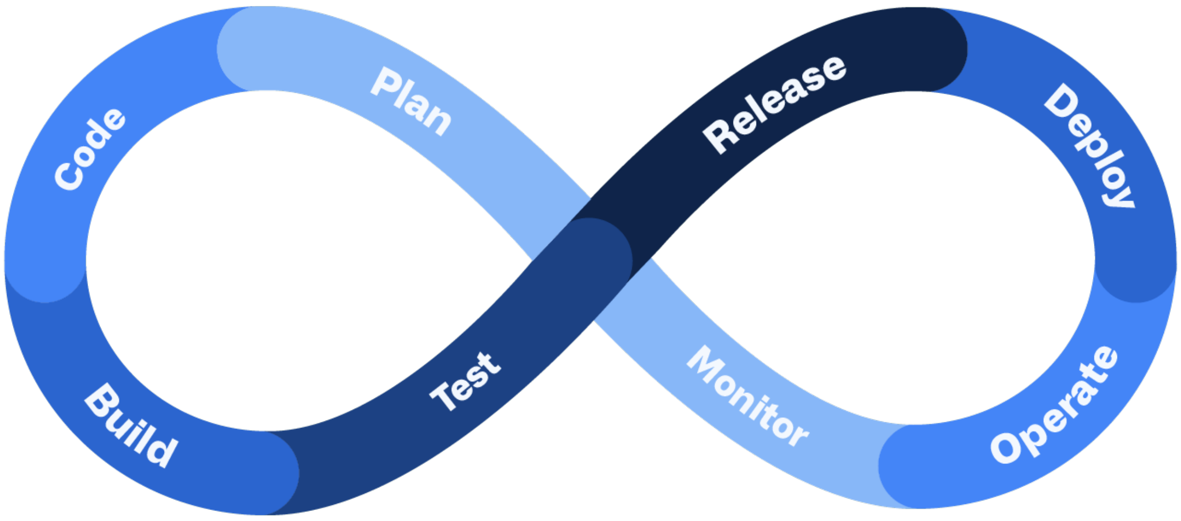

Figure 2: A DevOps pipeline

To understand this, it helps to break down the stages of a DevOps pipeline and how they feed back into one another:

1. **Plan:** Every DevOps pipeline starts in the planning stage, where new features or fixes are introduced and scheduled. At this stage, the primary goal is to ensure people who play different roles within the larger DevOps practice are collaborating from the start—and that means working together to understand user needs, design a solution, understand the implications of the change, and ensure it fits smoothly into the existing system. Learn how to use GitHub for project planning with GitHub Issues.

2. **Code:** In the coding stage, organizations begin writing code according to the plan and track their work via a version control system such as Git. At this point in a DevOps pipeline, developers may use a number of tools in their development environment to introduce consistency in code styling and identify any potential security flaws. This might include utilizing tools such as a cloud-hosted **integrated development environment** (**IDE**), which are often used to introduce consistency across development workflows and increase the speed at which coding environments can be spun up. Find out how developers code on GitHub.

3. **Build:** The build stage is when a DevOps pipeline fully kicks into gear and begins once a developer commits code changes to a shared repository. At this point, a developer may submit a pull request to merge their code changes with the codebase. This will alert someone else on the team to review their code before approving the merge. At the same time, a typical DevOps pipeline will initiate an automated build process that merges the codebase and begins a series of integration and unit tests. If any of these tests or the build itself fails, the pull request will also fail, and the developer will get a notification about the issue. This level of workflow automation in a DevOps pipeline helps organizations mitigate any potential build integration problems and identify any bugs or security issues at an earlier point in the SDLC. See how GitHub enables CI/CD for code integration and application building.

4. **Test:** After a build is approved, the testing stage in a DevOps pipeline begins and the build will be deployed to a testing environment that closely mirrors the production environment. Some organizations may elect to adopt **infrastructure-as-code** (**IaC**) in their DevOps pipeline to automate the provisioning of a testing environment for staging. Others may have dedicated,

pre-built testing environments ready for any new build—the choice largely depends on an organization's needs and computing resources. Once the build is deployed to the test environment, it will be subject to a number of automated and manual tests. This may include automated security tests such as **dynamic application security testing** (**DAST**) and **interactive application security testing** (**IAST**) to identify any vulnerabilities or risk areas. It can also include manual **user acceptance testing** (**UAT**) where team members will use the application and note any potential problems or bugs a customer may encounter. Every organization will have its own unique automated and manual testing suite and strategy during the test stage in its DevOps pipeline. But this stage, critically, provides a space for organizations to apply their tests without disrupting the development workflow. Learn how to use GitHub to build a continuous testing pipeline.

5. **Release:** The release stage marks the point in a DevOps pipeline where a new build has been fully tested and is ready to be deployed. In addition to the code itself having been tested, its operational performance has also been cleared, leaving organizations confident that it will successfully run in production without being affected by any undiscovered bugs or issues. At this stage, some organizations will elect to automatically deploy code when it reaches this stage in a practice commonly called continuous deployment. This is how some software teams deploy several code changes a day. Others may instead decide to manually release a new build into production and include a final approval stage, and still others will schedule automated releases to happen on certain days or at certain times. CI/CD platforms and other DevOps tools enable organizations to build a release cadence that best works for them—and apply automation throughout the release stage in their DevOps pipeline. See how companies release software on GitHub.

6. **Deploy:** Once a build has been released, it should be ready to deploy into production. At this stage in a DevOps pipeline, organizations will leverage a number of tools to automate the deployment process by provisioning new production environments via IaC or orchestrating a blue-green deployment (this is where code changes are slowly rolled out to a percentage of users in a new environment while the old codebase remains operational for other users in a separate environment). A blue-green deployment strategy also enables organizations to quickly migrate users back to an old build in the event that anything goes wrong. Learn how to deploy your code with GitHub.

7. **Operate:** A DevOps pipeline doesn't end once an application is deployed—that's when the operational stage begins, and organizations need to make sure everything is running smoothly. This stage includes infrastructure orchestration and configuration settings that will enforce rules to automatically scale resources to meet real-time demand. It also will often include mechanisms to capture user activity within the application such as behavioral logging and customer feedback forms. The goal in the operations stage is implied by the name of the stage: to successfully operate the application and underlying infrastructure and seek out ways to improve the operational profile of the software itself. Learn how GitHub enables software operations.

8. **Monitor:** Building upon the operational stage of a DevOps pipeline, organizations will set up automated monitoring tools to identify potential performance bottlenecks, application issues, and user behavior. This stage requires implementing tooling to collect data on application and infrastructure performance and then pass actionable items back to the product teams to either resolve outstanding issues or develop new features to support existing user behaviors in the application. Even though this is the last stage of a DevOps pipeline, it's important to understand that the process itself is continuous—i.e., monitoring tools help organizations identify areas for additional planning and iteration to feed back through the DevOps pipeline. Each stage feeds into the next in an infinite loop. Find out how GitHub gives organizations advanced monitoring capabilities.

Each of these stages makes up a part of the overall DevOps picture. As the diagram depicts, each of these stages flows to the next in an infinite loop. It requires diligent effort in each of the stages for an organization to transform itself into a well-oiled DevOps machine. As the stages in the DevOps pipeline become more natural for an organization, they will bring increased value to customers faster with improved quality.

Automating as much of the process as possible, from code to production, is fundamental to DevOps. The next section discusses the concepts of CI and CD, and how they fit into the DevOps pipeline.

Continuous integration and continuous deployment concepts

Continuous integration (CI) is a foundational DevOps practice where development teams integrate code changes from multiple contributors into a shared repository. **Continuous deployment (CD)** is an automated software release practice where code changes are deployed to different stages as they pass predefined tests.

CI enables organizations to quickly identify defects and ship higher-quality software faster, and the goal of CD is to facilitate faster releases by using automation to help remove the need for human intervention as much as possible during the deployment process. Together they are often referred to as CI/CD.

The next few sections discuss the concepts of CI/CD as well as guidelines organizations should follow to yield the best results. CI and CD make up a large part of the overarching DevOps pipeline, encompassing code from when is first written, built, and tested, all the way through release. Additionally, the topics of containerization and security in DevOps (DevSecOps) are introduced, and we'll see how they can play a crucial role in a DevOps organization.

“

CI/CD with GitHub Actions allows us to build, test, and deploy right from GitHub. We've reduced build time from 80 to 10 minutes.”

Engineering Architect at Pinterest

Continuous integration

CI seeks to encourage faster and more efficient development cycles by solving a key problem in software development: managing code integration challenges in a shared repository with multiple contributors.

When a developer begins working on a software update or fixing a bug, they make a copy of the codebase to work from. This is done via a version control system such as Git, which enables developers to create a copy of, or "fork," the codebase.

As more developers create codebase copies, integrating the changes from multiple contributors can become challenging—especially when the codebase that one developer started working from becomes dated and no longer matches the main repository.

In a worst-case scenario, it can take longer to successfully integrate code changes than to make the changes themselves as each developer tries to untangle where their code is not matching up. Developers often call this "integration hell."

CI seeks to prevent this by encouraging developers to integrate changes as they make them. CI also leverages automation to increase the speed at which code is integrated and tested to ensure no additional changes are needed, reducing the burden for a developer. This combination of more frequent code integrations and automated builds and testing helps speed up the software development process.

Every company will define its CI practice per its unique needs. Some companies may introduce more rigorous automated security tests; others may prioritize fast code merges and reserve more time-consuming automated tests for later in the SDLC.

Despite this, effective CI pipelines share a set of common tools and best practices. These include:

- **A shared code repository:** A shared code repository in a version control system is foundational to creating an effective CI practice. Beyond serving as a place to store code, scripts, automated tests, and everything in between, version control systems also enable developers to create multiple branches from which to work.

- **Regular code commits:** Automation, testing, and tooling are important for creating an effective pipeline—but without a team cultural shift that prioritizes committing code changes often, you're unlikely to get very far. There are no hard and fast rules for how often developers should be committing code. A good rule of thumb, however, is that the more often individuals commit changes, the more productive the development environment will be.

- **Build automation:** Build automation is a critical component of a CI pipeline and enables teams to standardize their software builds. A typical build process includes compiling source code, generating software installers, and ensuring that all the necessary items are in place to support a successful deployment. In a CI practice, this process is automated to help integrate incremental code commits into the codebase.

- **Automated testing:** You can make lots of code commits and have a fully automated build process. But just because a program runs, it doesn't mean it's running correctly. That's where testing comes in. Automated testing is a key part of CI pipelines. Each commit triggers a set of tests to identify bugs, security flaws, and commit issues. These tests are meant to keep the main code branch operational, or "green," and give rapid feedback to developers about the efficacy of their code changes. No two CI models are alike. Every organization will implement CI according to its unique needs and team requirements.

There are, however, some common steps every organization needs to take to implement CI successfully. These break down into five practices:

1. **Create a testing strategy:** Every CI practice starts with a cogent and clear testing strategy. You'll need to consider what types of tests you're running, what triggers you use to build your automated testing sequences, and which tests you apply to each coding branch your development teams will work on.

2. **Choose a CI tool:** Choosing a CI platform is a critical part of building a CI model in any organization. This is the tool that will trigger your automated builds, tests, packages, and releases.

You'll want to ask several questions when selecting a CI platform. These include:

How well does it integrate with your current technology stack?

From your programming languages to your version control system to your third-party tools, a CI platform should easily integrate with everything in your stack. It's also worth considering any future technologies you might adopt and looking for a platform that can grow with you.

Does it offer native support for containers?

Containers are a critical part of a DevOps and CI practice and making sure your CI platform has native support for container applications such as Docker is critical. You might not leverage containers today, but as you grow your DevOps practice there's a good chance you will end up using containers in some capacity.

Does it enable matrix build testing capabilities?

Matrix builds enable you to simultaneously test builds across multiple operating systems and runtime versions. Look for a CI tool that has native support for matrix builds, which help streamline your testing and ensure your application will work for all of your end users.

Does it offer built-in code coverage and testing visualization?

Code coverage and testing visualization give you a simple way to see how much of your codebase is currently being tested and how existing tests are running in real time and have run historically.

How does it map to your security requirements?

Security is a critical consideration with any technology investment—especially if that investment will end up deeply integrated with your codebase and core services.

3. **Integrate code as soon as possible:** Successfully adopting CI starts with making sure your developers are integrating their code as soon as possible to a shared repository.

There are two benefits to this:

- You avoid larger integration conflicts that can arise when merging older branches back to the main repository.
- You end up regularly integrating smaller code changes, which helps with knowledge transfer between your teams and simplifies your testing regimen.

You should also consider what your existing SDLC looks like today, and what changes you might want to make procedurally moving forward as you implement a CI pipeline. This isn't a conversation about whether feature branching or trunk-based development is better either. Instead, it's about making sure you have an organized development workflow that facilitates a steady stream of coding, testing, merging, and reviewing.

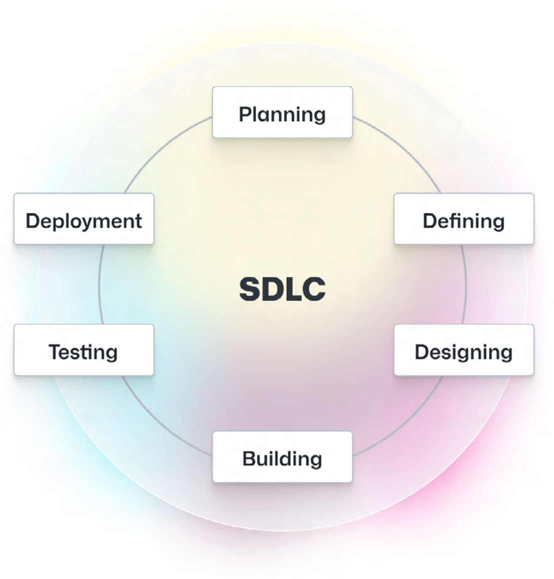

Figure 3: The stages of the SDLC

4. **Fix your main branch as soon as it breaks:** As a rule of thumb, you should fix your main branch as soon as it breaks. In a CI model, your developers should be integrating code changes as soon as possible. That's a good thing. But if a code change breaks your main branch and your developers keep adding more changes, it becomes difficult to identify what caused the initial failure. To do this, write tests that immediately notify developers when one of their code changes breaks the main branch. This helps create a feedback loop, which is an important DevOps practice. Make sure to balance testing speed with testing coverage when it comes to keeping your builds green, or operational. If your tests take too long to run, it becomes harder to pinpoint what code change led to a failure. The best testing suites start with simple tests such as build and integration tests before advancing to more time-consuming tests.

5. **Build new tests for every new feature you introduce:** Under a CI model, your testing suite should grow with your software or application. That means that as you build new features and prepare larger updates, you should also be building tests to validate these features. Consider writing tests as you build new features and fix bugs. This might feel time consuming—but going back after the fact will almost certainly take longer than writing tests as you build code.

The idea behind CI is to integrate code as often as possible, in small chunks. It's easier for developers to merge a code branch that contains a single new feature or hotfix than it is to merge several branches that have weeks' worth of work. When CI concepts are being followed, it is easier for teams to pinpoint which bit of code broke the build or caused the tests to fail. This allows developers to spend more time creating value instead of troubleshooting build errors.

With CI, code is integrated early and often to minimize merge conflicts that occur when multiple developers are working on the same codebase. The next section introduces the next step in the process: CD. With CI and CD working together, organizations can build, test, and deploy code often and with confidence.

Continuous deployment

Continuous deployment, or CD, is one of the more advanced examples of automation in a DevOps practice. It requires a mixture of rigorous testing, deep cross-team collaboration, advanced tooling, and workflow processes across the application design and development process.

And when it's successfully implemented, it works. DevOps organizations that adopt CD have been found to ship code faster and outperform other companies by 4-5x.

DevOps seeks to increase the speed of innovation and value delivery by applying automation to every stage of the SDLC. With that view, CD stands as the ultimate goal of DevOps: a completely automated SDLC where every code change is pushed to production if it passes a predefined set of tests.

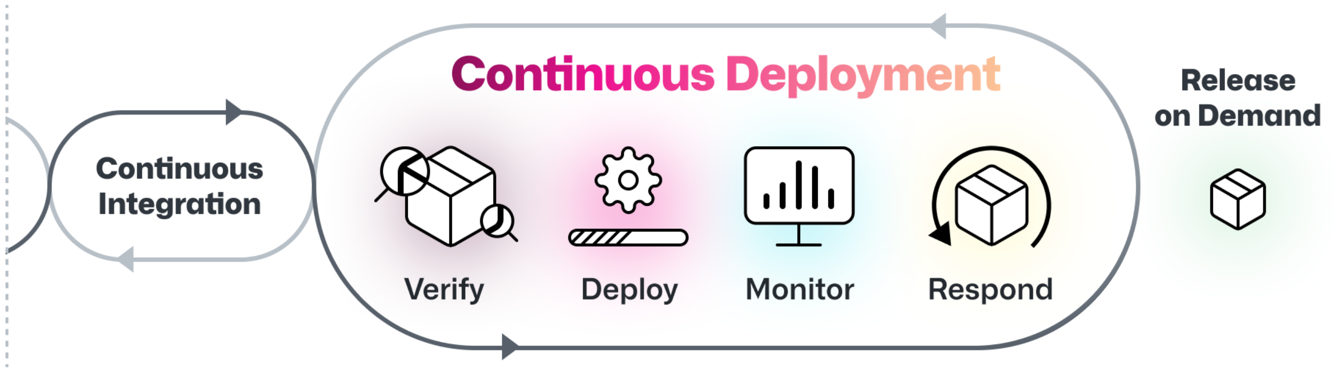

Figure 4: Continuous deployment in practice

In some ways, building an automated pipeline is one of the easiest parts of adopting a CD model. But very few organizations start their DevOps journey by building a CD practice due to the cultural change it signifies, and the maturity of the testing suite it requires.

In that light, it's best to understand the process and journey of achieving a fully functioning CD practice.

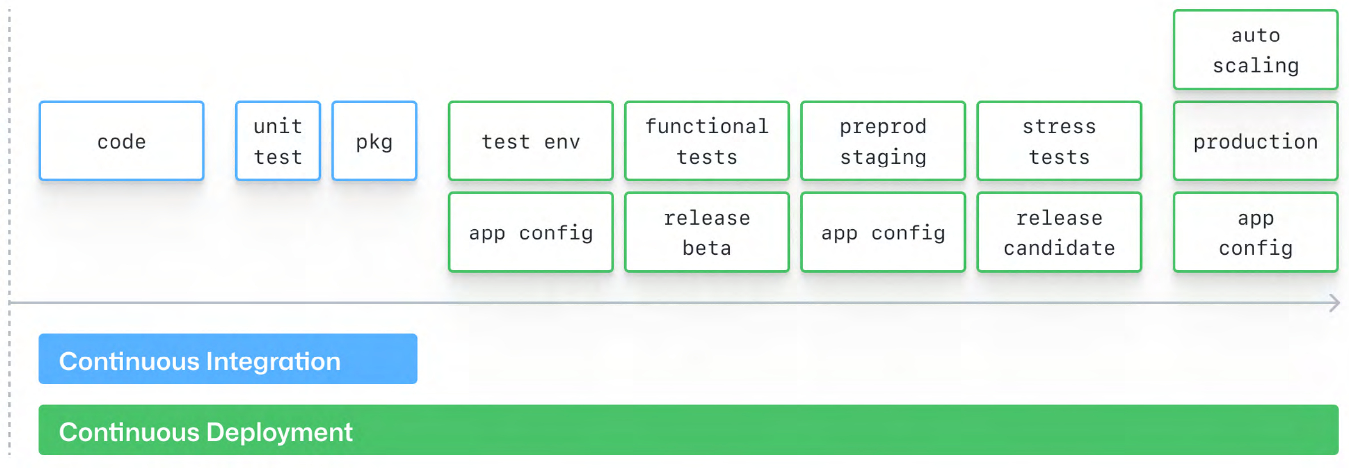

Figure 5: Steps to automation in CI/CD

Figure 5 shows a high-level journey map for how organizations typically start thinking about automating the SDLC.

To start, organizations need to build a CI practice. The foundational elements of a strong CI practice—regular code commits, a testing strategy, version control tooling, and a CI platform—set the stage for organizations to begin developing a CD practice.

At its most basic, CD brings automated builds, tests, and deployments together in a single release workflow. The goal is to automate the deployment of software builds into production. Each company needs to identify the right combination of unit, functional, and stress tests that comprise its testing suite. It's also critical to mirror production environment pressures in a pre-production environment to effectively stage and test builds and release candidates.

Getting all of this right leads to a significant payoff: faster and more stable releases. It also positions organizations to achieve CD with a fully automated CI/CD pipeline.

Ideally, a DevOps practice becomes so fine-tuned across its testing regimen, automation triggers, workflow composition, and CI/CD platforms that it naturally leads to CD. In effect, the need for human intervention to orchestrate a software release dissipates over time.

In practice, however, achieving a durable and scalable CD model takes significant investments in engineering resources and tooling. And while CI/CD platforms and associated tooling go a long way to standing up a CD practice, a cultural change that emphasizes cross-team collaboration and regular code commits is critical.

Stages in the CD pipeline

There is no singular "model" for CD. Every organization will build a CD pipeline unique to their needs, software development practices, and customer demands.

Despite this, there are four commonly accepted stages in any CD pipeline that every organization should build into their engineering plans. These include:

1. **Verification:** CD builds upon CI–and it's at this stage that CI stops, and CD begins. After a new piece of code is committed and integrated into the codebase, this triggers the automated verification process that runs a series of tests on a release candidate build. This can include functional, integration, security, and production-level testing to ensure a release candidate will work following deployment.

2. **Deployment:** Once code is verified via testing, the automated deployment process begins. More advanced implementations will typically create automation workflows that move code to deployment immediately after it is committed (of course, this assumes it passes all predefined tests in the CI stage).

3. **Monitoring:** Continuous monitoring is a critical element that organizations need to invest in to support CD. Monitoring should take place across the SDLC. But the ability to see what is and is not working and receive real-time alerts before, during, and after deployments is key. Tooling that helps teams visualize performance metrics and show system strains is one helpful investment.

4. **Response:** Whether it's addressing a production-level system error or identifying a security incident or a potential new feature for development, being able to respond to events is a critical element of a CD pipeline. A benefit of CD is that code is immediately released into production. This also means that organizations need to be prepared to respond to and address any issues that emerge after deployment. Common metrics used to evaluate response times include **mean time to resolution** (**MTTR**), which organizations will track to evaluate improvement over time.

Gone are the days when software was predominately distributed by floppy disks or CDs. With the advent of cloud-based hosting solutions and technologies, practicing CD concepts allows organizations to actively deliver value to customers through software updates. This doesn't necessarily mean that every DevOps organization needs to deploy code to production dozens of times a day. Instead, the idea is that organizations can deploy as soon as the code is ready. Deployment to production is a non-event in organizations with a successful DevOps practice. The code that is being deployed is relatively small in function, has already been rigorously tested, and deployment automation leaves no room for human error in the deployment process.

Containerization

It's not just code where problems can arise. What works on one person's machine might behave differently on a colleague's laptop—or worse, on a production server. Containerization is one type of technology that can be used in DevOps practices to ensure that the software environment is consistent from one machine to another during development, testing, and on into production.

Containerization packages software code with dependencies and an operating system in the form of a standalone application that can be run on top of another computer. These virtualized environments are lightweight by design and require comparatively little computing power. They can also be run on any underlying infrastructure and are portable or are able to be consistently run on any platform.

By bundling application code, configuration files, operating system libraries, and all dependencies together, containers help solve a common problem in software development: code that is developed in one environment often exhibits bugs and errors messages when transferred to another environment. A developer may, for instance, build code in a Linux environment and then transfer it to a virtual machine or Windows computer and find their code no longer works as expected. In contrast, containers stand alone from the host infrastructure and provide consistent development environments.

But what makes containers particularly useful is that they are easy to share. By using container images—files that act as a snapshot of the container's code, configuration, and other data—you can quickly spin up consistent environments across each stage of the SDLC. This helps organizations create reproducible environments that are fast and easy to work with from development through testing and on into production.

Now that we have a base understanding of what containers are, the next section highlights the benefits containerization brings to DevOps. Afterward, we discuss the specific aspects of CI/CD where containers can have the most effect.

The benefits of containerization in DevOps

At the heart of DevOps are lightweight, repeatable processes that automate the software development process. However, modern applications are increasingly complex, particularly as they grow to include many different services.

Containers help simplify that complexity through greater standardization and repeatability—and that translates to a faster, higher-quality, more efficient SDLC. GitHub provides tools that help companies adopt and manage containers in their DevOps practice. Through this experience, GitHub has identified key areas organizations need to consider to successfully integrate containers into their SDLC.

The benefits of containerization include:

- **Portability:** Even seemingly small differences in the underlying environment can impact how code runs. That's why the saying "It works on my machine" is rarely meaningful—and is often somewhat of a joke. It's also why the saying "Write once, run anywhere" has been a recurring goal for people looking to improve software development practices. Containers help organizations accomplish this by bundling up everything an application needs into consistent and portable environments that make it easier to standardize application performance.

- **Scalability:** Containers can be deployed and configured to work with one another in a larger system architecture through the use of orchestration management tools such as Kubernetes. These tools can also be used to automate the provisioning of new containerized environments to scale with real-time demand. That means properly configured containerized environments can be rapidly scaled up—or scaled down—with little to no human intervention.

- **Cloud-agnostic:** When configured for portability, containers can run anywhere—whether that's a laptop, bare metal server, or cloud provider platform. And because containers abstract away underlying platform differences, they mitigate the risk of platform lock in. You can also use containers to run applications across multiple cloud platforms and switch from one provider to another.

- **Integration into the DevOps pipeline:** Containerization platforms are often designed to be inserted into larger automation workflows. That makes them ideally suited to DevOps where CI/CD tools can create and destroy containers automatically for tasks such as testing or even deployment into production.

- **Efficient use of system resources:** Unlike virtual machines, containers are often more efficient and require less overhead. There's typically no hypervisor or additional operating system that's native to a container. Instead, container tools provide just enough structure to make each container a standalone environment that leverages the shared resources of the host system wherever possible—and that includes the underlying operating system, too.

- **Facilitate faster software releases:** Containers can be used to simplify larger and more complex applications by dividing their underlying codebases into smaller run-time processes that work together. This helps organizations accelerate each step of the SDLC because it enables practitioners to narrow their focus to a specific part of an application rather than working with the entire, wider codebase.

- **Flexibility:** Containers bring inherent flexibility to the SDLC by enabling organizations to quickly provision more computing resources to meet real-time demand. They are also often used to create redundancies to support greater application reliability and uptime.

- **Improved application reliability and security:** By making the application environment part of the DevOps pipeline, containers face the same quality assurance as the rest of the application. And although containers work together, the isolated environment provided by a container makes it easier to prevent issues in one part of the application from impacting the wider system.

Containers provide a modern way to develop software efficiently at scale. They provide flexibility and repeatability that pair well with fundamental DevOps practices. In the next section, we will explore how containers can enhance the CI/CD process.

How containers work in CI/CD

A CI/CD pipeline can be thought of as the conveyor belt that drives the DevOps workflow. To be effective, a CI/CD pipeline must balance speed with thoroughness. Without speed, a CI/CD flow risks backlogs as commits occur faster than they can make it through the pipeline. Without thoroughness, people will lose faith in the CI/CD pipeline as problems slip into production.

Here's how containerization boosts both aspects of CI/CD at key stages:

1. **Integration:** By using containers, you don't have to start from scratch when integrating code changes to the larger codebase. You can create a base container that already holds the application's dependencies and modify that during the integration phase.
2. **Test:** Containers can be quickly provisioned and retired as necessary. Rather than needing to manually maintain explicit test environments or wait for configuration scripts to build an environment, a container can be provisioned and deployed automatically at scale. That way, tests run faster and with less need for human intervention to build test environments.
3. **Release:** Once all the tests pass, a CI/CD pipeline's build phase results in a container image that is then stored in a container registry. Once that image exists, much of the work that would usually take place in the release and deploy phases is already complete. Orchestration tools such as Kubernetes then take care of managing where the containers are deployed and how they interact.

The usage of containers in a DevOps CI/CD workflow for their predictability and scalability reflects foundational DevOps principles. While using containers is not an absolute requirement of DevOps, organizations that use containers effectively tend to be higher functioning on the DevOps maturity model. Containers are seen as a natural fit to reduce the friction of modern application development by allowing increased consistency and repeatability. It is this consistency and repeatability, along with reliability, that is essential for any successful DevOps practice.

Security in DevOps (DevSecOps)

DevOps has transformed how many organizations build and ship software. But until recently one aspect of the SDLC has remained outside DevOps: security. DevSecOps seeks to correct that by baking security into the SDLC in the same way that DevOps prioritizes quality, speed, and deep collaboration throughout all stages of software development.

DevSecOps seeks to build security into every step of the SDLC. This ideally means that security-related tests (automated and not) take place at each stage from coding to merging branches to builds, deployments, and on into operation of production software. Moreover, DevSecOps advances the idea that everyone working on a product is accountable for its security. This helps teams catch vulnerabilities before they make it to production and reduces the need for late-stage, manual security reviews, which can slow down software releases.

Organizations that adopt DevSecOps typically see advantages that include:

- **Reduced risk of data breaches:** DevSecOps seeks to make code secure by design. A combination of secure coding cultural practices, secure developer environments, and automated security tests throughout the SDLC help reduce the chances of security vulnerabilities or flaws making it into production software.
- **Improved compliance:** DevSecOps practitioners often use automation to enforce code compliance and integrate policy enforcement tooling directly into the CI/CD pipeline.
- **Greater confidence in dependencies:** The modern technology stack depends heavily on third-party code, often from public package repositories. DevSecOps practitioners frequently leverage tooling and automated tests to identify potential issues before a software release.
- **Value gets to end users faster:** By creating a security-first culture and applying automated checks, DevSecOps reduces the need for distinct security reviews that slow down code deployments.

Building a successful DevSecOps practice requires building security in every stage of the SDLC. This varies from one organization to another, and often different industries will have different regulations that must be followed. With DevSecOps, building security into each phase of the SDLC doesn't mean implementing onerous controls, which can slow down software development. Quite the opposite. Security in effective DevSecOps practices becomes a part of the release itself, leading to faster and more secure deployments.

Best practices

DevSecOps argues that security needs to be embedded across the SDLC. Whether your organization already practices DevOps or you're looking at how to adopt a DevOps culture, here are the foundational best practices you need to establish a DevSecOps practice:

- **Create a DevSecOps culture:** Success in DevSecOps relies on everyone taking responsibility for security. That means each person in the SDLC codes, builds, tests, and configures application and infrastructure settings defensively. Just like DevOps, DevSecOps thrives in an open culture where each individual works together to build the best and most secure product possible.

- **Design security into the product:** DevSecOps seeks to design security into products from the initial planning stages to deployed production-level code. This means security work is planned alongside feature work, and practitioners are provided security knowledge and testing throughout each stage of their development work. The goal is to make security an everyday part of your team's work.

- **Build a threat modeling practice:** The seeds of security vulnerabilities are often sown before a line of code is written. Model potential threats during the planning phase and design your infrastructure and the application's architecture to mitigate those issues.

  And periodic penetration testing, where a trusted person attempts to break into your system, can help unveil weaknesses you may miss in your threat models.

- **Automate for speed and security:** Automated testing is used throughout the SDLC to ensure the right security checks happen at the right time. That gives people more time to focus on building the core product while ensuring security requirements are met.

- **Plan security checkpoints in your product development:** Identify transition points in your SDLC where the risk profile changes. That could be the point at which a developer merges their code into the main branch, which might increase the potential for that code to be run on the machines of colleagues and eventually reach production. In that case, opening a pull request might be a good trigger event for automated security checks, along with the appropriate manual escalations.

- **Approach security failures as learning opportunities:** Building on DevOps' culture of continuous improvement, a successful DevSecOps practice strives to turn security incidents into learning opportunities. This can be accomplished by leveraging audit logs, building incident reports, and modeling malicious behavior to improve tooling, testing, and processes to further secure your applications and systems.

- **Stay on top of dependencies:** Understanding and mitigating the potential threats from dependencies is critical to your product's security. Apply the same threat modeling and automated testing to your dependencies as to your in-house code. GitHub has identified and shared details of tens of millions of threats in open-source software, helping organizations and developers be more aware of and avoid vulnerabilities.

- **Build your analytics and reporting capabilities:** Continuous monitoring is a critical part of a DevSecOps practice—and that includes real-time alerts, system analytics, and proactive threat monitoring. By measuring every aspect of your application and your DevSecOps pipeline, you can create a common point for understanding application health. Reporting dashboards and alerts highlight problems early. When a problem does occur, the telemetry you've set up—such as application-level logging—provides insight for incident resolution and root cause analysis.

Creating a DevSecOps culture begins by making security everyone's responsibility. This can be a big change for many organizations. Traditionally, security was something developers left in the hands of security professionals. There was often friction since engineering teams looked at security practices as an impediment to shipping software fast. In more extreme cases, security was merely a rubber stamp in the process.

DevSecOps fundamentally seeks to change this perception by making security as core to the SDLC as writing code, running tests, and configuring services. Just like how the DevOps model brought developers and operation teams together, DevSecOps brings security to the forefront. Each new feature or fix begins with considering its security implications. Security and compliance policies are enforced through automated tests. For modern organizations, DevSecOps becomes just "DevOps": security is baked into all aspects of the SDLC workflow.

DevOps planning, tools, and capabilities

Automation, continuous monitoring, and continuous feedback are essential parts of the DevOps model.

As an umbrella term, DevOps tools include any number of applications that automate processes within the SDLC, improve organizational collaboration, and implement monitoring and alerts. Organizations will often invest in building out a "DevOps toolchain," or collection of tools to use in its DevOps practice, to address each stage of the SDLC.

A DevOps toolchain is a core tenant of any DevOps practice, helping organizations apply automation to the SDLC and improve their ability to deliver higher-quality software faster. It's also one of the more tangible aspects of DevOps.

Some organizations will invest in an "all-in-one" platform to build their DevOps toolchain. Others will integrate different best-of-breed solutions to create a toolchain. But critically, there is no one-size-fits-all approach to DevOps or building a DevOps toolchain.

Our philosophy is to build automation and great DevOps for the company you will be tomorrow."

Todd O'Connor
Senior SCM Engineer at Adobe

Guide to tools

Each stage of the DevOps pipeline has unique considerations that one or many tools can help solve. Over the next few sections, all eight stages of the DevOps pipeline are reintroduced along with various considerations to keep in mind when selecting tools. A single tool might account for several stages of the DevOps pipeline. Conversely, a single stage in the DevOps pipeline can be represented by multiple tools. Regardless of the tools being used, the most successful organizations are going to have cohesive flow and integration between each of the stages in the DevOps pipeline.

In this section, we'll look at how DevOps tools can shape your strategic planning, communication, and roadmap for the future.

DevOps planning and collaboration tools

In large part, DevOps seeks to bring previously siloed teams together across all stages of the SDLC—and that starts at the planning stage. From chat applications to project management tools, there are a number of tools organizations can implement in their DevOps toolchains to better align and encourage collaboration in an organization during its planning stages.

DevOps planning and collaboration tools generally fall into two buckets:

- **Product and roadmap planning:** Having a centralized place to plan, track, and manage work is a foundational capability for any modern development team—and DevOps organizations, too. The best tools help organizations build plans, sprints, and roadmaps while being able to assign and track work from the initial plans to the delivered end product. Need an example? Try looking at our own public product roadmap plans, which we build using projects on GitHub.

- **Team communication:** Maintaining communication throughout the planning process is key to spurring collaboration—and having a preserved record of conversations that led to a given decision can be incredibly helpful. Tools such as GitHub Discussions, chat applications, and issue trackers that enable team conversations are key here. GitHub provides apps to help your team integrate with Slack or Microsoft Teams. The best tools will integrate with your project planning, too. That means you can turn a discussion into an executable piece of work or turn an idea into a discussion if more conversation is needed before work can start.

DevOps build tools

Once developers commit code changes to a central repository, the build stage begins—and that means using version control to create shared repositories, provisioning development environments, and integrating code, among other things.

At this stage, organizations can typically take advantage of the following DevOps tools:

- **Version and source control:** A version control system is designed to automatically record file changes and preserve records of previous file versions, which enables code rollbacks, historical references, and multiple code branches allowing developers to collaboratively code and work in parallel. Platforms such as GitHub offer version control and source control with features such as pull requests, which enable individual developers to get reviews on proposed code changes before they are integrated into the main code branch. The best version and source control platforms integrate with your broader DevOps toolchain and enable product teams to collaborate across the SDLC.

- **Pre-production development environments:** In a DevOps practice, developers need to leverage virtual environments that mirror production as closely as possible. These environments are identical to one another and easy to provision, so that all developers can quickly build and test code changes in consistent environments. Organizations will often leverage containerization platforms and registries such as GitHub Packages to build standardized, pre-production environments for development teams. Ideally, these platforms should integrate into the source control solution so that when a team member commits new code, it triggers the automated provisioning of a pre-production environment.

- **Cloud-based IDEs:** Cloud-based IDEs offer comprehensive development environments that are pre-configured and can be quickly provisioned. These are an increasingly popular tool in DevSecOps (and development circles more broadly, for that matter) since they help standardize developer environments, including security configurations across machines. And since they're centrally managed, cloud-based IDEs also keep code off an individual developer's computer, which can improve overall development security. Tools such as GitHub Codespaces also feature deep integrations into core DevOps platforms. This can improve development speeds by cutting down the amount of time it takes to spin up a developer environment—and reducing the need to wait for running builds and tests locally.

- **IaC:** The rise in cloud infrastructure, or **Infrastructure as a Service** (**IaaS**), has made it simpler to quickly provision resources to meet real-time demand. It's also introduced a need among organizations to manage complex cloud-based infrastructure at scale. IaC draws on DevOps best practices to provision and manage cloud infrastructure resources from a version control system such as GitHub via YAML files. These files specify a CI/CD workflow automation that is triggered by an event such as a pull request, code commit, or code merge. Once this event happens, the workflow automates the provisioning and management of cloud infrastructure resources. Tools such as GitHub Actions offer this type of integration, which makes it easier to manage infrastructure from your repository with CI/CD.

DevOps CI tools

CI is a mainstay of any DevOps practice and combines the cultural practice of frequent code commits with automation to integrate that code successfully and create builds.

- **CI:** As a practice, CI often involves committing multiple code changes each day to a shared repository and using automation to integrate these changes, applying a series of automated tests to the merged codebase to ensure its stability, and preparing the codebase for deployment. This level of automation requires deep integration between a version control solution and the larger CI/CD platform, which enables DevOps organizations to build CI/CD pipelines

that are triggered by a code commit. When you're looking for a good CI solution, you'll want to make sure it easily integrates with your version control solution. This integration is key to making sure you can build an automated pipeline that starts as soon as your development teams commit code changes.
A good example of this level of integration comes with the GitHub platform, which features platform-native CI/CD via GitHub Actions and also features a number of pre-built integrations for third-party CI/CD services. You'll also want to make sure that whatever CI/CD platform you choose can automatically apply tests at all stages of the SDLC and includes native support for containerization platforms.

- **Automated testing:** Automated testing tools are a core part of any DevOps toolchain. Most platforms will offer automated testing as a capability, making it simple to incorporate automated tests into key parts of the pipeline—for instance, after a code change is merged to the main branch. The goal is to have a comprehensive testing strategy with basic unit tests, integration tests, and acceptance tests that are applied at key points in the SDLC. The best testing tools integrate seamlessly with—or are part of—your CI/CD platform and offer built-in code coverage and testing visualization. You'll also want to look for testing platforms that enable matrix build testing capabilities or allow you to simultaneously test builds across multiple operating systems and runtime versions. It's also a good practice to ensure that your automated test solution of choice comes with monitoring and alerts that integrate with your chat application of choice. This means that if something breaks, you can quickly get a notification and work to fix whatever the underlying problem is. Tools such as GitHub Actions, for instance, can be used to send alerts to chat applications once a test fails for quicker remediation.

- **Packaging:** Once code changes clear all tests in a CI/CD pipeline, they are packaged into independent units of code and prepared for deployment. DevOps organizations will typically leverage a package manager such as GitHub Packages to facilitate the delivery of software packages to a shared repository in preparation for a release. Package managers help remove the need for manual installations and help bundle code dependencies within a given project. There are different package managers for different code libraries—but you should ideally look for a solution that integrates with your version control system and your CI/CD platform.

DevOps CD tools

CD builds upon CI/CD by removing the need for human intervention when releasing software. Instead, a CD practice applies automation to every stage of the SDLC. That means if a code change clears all automated tests, it is deployed to production. These tools support CD:

- **Automated deployment:** Automated deployments are a core part of CD and having a toolchain that supports automated deployment. These capabilities are typically present in most CI/CD platforms. However, there is no one-size-fits-all approach to building out a CD pipeline—and it won't work with every application or environment. If you decide to invest in CD, look for platforms that readily support the development and management of multiple environments. Importantly, you'll want a solution that helps protect you from "server drift," or differences between development, pre-production, and production environments. You'll also want to consider a platform that supports blue-green deployments, which enables you to slowly migrate traffic from an old version of an application to a new release to ensure its stability in production. At GitHub, we provide deployment dashboards and CI/CD visualization displays as part of our native CI/CD tool GitHub Actions—and we consider these core features for any CD toolchain. This is meant to give DevOps organizations full visibility into different code branches, automated test results, audit logs, and ongoing deployments as they happen.
- **Configuration management:** Configuration management is a process where technology teams manage the different environmental configurations necessary in the core infrastructure and application systems across the life of the product.It's also something that is frequently paired with CI/CD and versioning control via automation. Just as a CI/CD pipeline applies automation across the SDLC, configuration management tools automatically apply configuration changes in response to trigger-based events. These automated workflows can be used to orchestrate and manage container clusters with platforms. GitHub repositories and issues make it easy for IT professionals to work with systems that produce text-based configuration files for both IaC and **Configuration as Code** (**CaC**).

Continuous testing tools

In a DevOps practice, testing doesn't stop at CI/CD—it's an ongoing practice that extends throughout the SDLC. And more importantly, DevOps seeks to replace siloed QA teams with a continuous testing practice that leverages automation and holistic testing strategies across the SDLC.

Each DevOps organization will design its own continuous testing strategy in accordance with its needs. GitHub Actions provides workflow automation related to testing and supports a rich set of open source and commercial testing tools. Every continuous testing strategy will leverage a combination of the following test types across the SDLC:

- **Unit testing:** Unit tests are a way of testing small units of code to verify that they are structured correctly with isolated components. They are also the easiest tests to build and the fastest to execute, making them a foundational test to automate in any continuous testing practice.

- **Integration testing:** Once you commit code changes to a repository, integration tests ensure build stability, and that the codebase continues to work successfully. These tests are used to identify defects that emerge when different application processes and code units are merged together. Integration tests are commonly automated to begin as soon as code changes are committed to a codebase and test the interplay of multiple parts of an application.

- **End-to-end and regression testing:** Building on integration testing, end-to-end and regression tests are applied after a codebase is packaged and staged in a pre-production environment. These tests are used to check if any old defects, bugs, or issues are reintroduced by code changes. Regression testing is commonly used before and after deployments to ensure that an application works as expected and does not contain any previously identified issues.

- **Production testing:** After an application is deployed, production-level tests monitor application health and stability—and identify any issues before they cause problems for end users. Importantly, these tests help organizations identify any potential problems in a production environment with live user traffic that can't be fully replicated in a pre-production environment.

DevOps operations and continuous monitoring tools

A successful DevOps practice touches every stage of the SDLC—and that includes production-level software, too. This means companies need to invest in core operations and continuous monitoring tools to evaluate application and infrastructure performance. If used correctly, these tools can help continuously identify potential issues across the SDLC:

- **Application and infrastructure monitoring:** Application and infrastructure monitoring are core components of a successful continuous monitoring practice. The best tools offer 24/7 automated monitoring of the application and infrastructure health and give DevOps practitioners alerts when something goes wrong—and visibility into what the underlying problem might be. Ideally, you'll want to monitor application health in pre-production and production environments to track any process issues or areas to improve overall performance. This is also true for your underlying infrastructure where monitoring can lead to insights on how to improve your IaC and configuration management policies. Try looking for a tool that integrates with your version control tool and chat applications so you can immediately send alerts to the right people and create issues to outline the scope of work for a solution.

- **Audit logs:** Auditing is a central part of an effective operations and continuous monitoring practice—and resolving any incidents if and when they happen. They give DevOps practitioners a record of what happened, where it happened, and when it happened, and can be critical to build behavioral models that led to an issue and improve application and infrastructure health. Look for DevOps tools that have live logs and auditing retention periods to equip your teams with the information they need to improve core services and application performance.

- **Incident and change tracking:** The primary goal of DevOps is to help organizations ship higher-quality software faster through deep collaboration and automation. And that means tracking incidents and changes as they arise and sharing them with the right people is critical. To build a successful DevOps toolchain, you'll want to incorporate tools that surface incidents and changes on your core DevOps platform and shared repositories. The more centralized you can keep all reports on incidents and changes, the better. The goal is to create a single source of truth that makes it easier to identify and fix issues.

- **Continuous feedback:** A core tenet of DevOps, continuous feedback is a practice that focuses on tracking user behavior and customer feedback about your core products and building actionable data to inform future investments in new features and system updates. This can include NPS survey data about how users are navigating your product. It can also include tracking and modeling user behavior in the product itself. To build a continuous feedback practice, you'll want to identify core areas in your product and even outside it in places like social media and reviews where you can identify unexpected user behavior and customer pain points. Look for tools that enable you to model and analyze user behavior. You also might consider social listening tools, which you can use to track historical patterns on social media and review sites.

Security and DevSecOps tools

As DevOps has evolved as practice, it has underscored the need to move past more traditional approaches to security, which was often siloed from the core SDLC. To ensure you're shipping high-quality code, making security a core part of the DevOps practice is important. This practice is commonly called DevSecOps, which seeks to integrate security into every stage of the SDLC and make it a core part of CI/CD pipelines.

Companies that invest in DevOps often find the need to invest in also building a DevSecOps practice to ensure software security. This typically involves several tools that help organizations model potential threats and apply automated security testing at key stages of the SDLC. While organizations often try to grab individual tools to create a solution, integrated products such as GitHub Advanced Security can reduce the friction of bringing DevSecOps to your teams. By complementing their DevOps toolchain with DevSecOps tools, companies will often look for the following solutions:

- **Threat modeling:** Here's a truism: it's a lot easier to find security vulnerabilities and potential weak points when you're developing software instead of after you've released it. Threat modeling is a practice that DevSecOps practitioners will engage in from the early planning stages of the SDLC to anticipate any issues and develop plans to solve them. DevSecOps organizations today will also invest in threat modeling tools that leverage automation and monitoring to proactively identify threats and mitigation efforts. The best tools survey application and infrastructure threats and will automatically track changes in the underlying codebase and infrastructure architecture. Look for solutions that can integrate with your core DevOps toolchain to provide updates to relevant people on your team and show risk evaluation scores throughout the SDLC.

- **Security dashboards:** Having a single view of your security profile including potential risks, testing coverage, alerts, and more is critical for any DevSecOps practice. Security dashboards are often used to collate and break down all relevant security information and provide a quick way to triage issues and assign tasks. At GitHub, we include a security overview page with GitHub Advanced Security to help showcase risk categories across projects and repositories and alert details, too. Ideally, you should look for tools that integrate with your wider DevSecOps security toolchain and offer a single view of your security profile.

- **Static application security testing (SAST):** SAST tools are used to evaluate code before it is run to identify any potential security risks or vulnerabilities. Importantly, these tools do not need a running system to execute but can be performed on a static codebase. The best tools will integrate directly into a shared repository and seek out any security vulnerabilities, conduct dependency reviews, scan for any confidential passwords or secrets, and identify coding errors before they make it into production. These tools will also make it simple to find, triage, and prioritize fixes for any problems in your codebase. You'll ideally want to look for a solution that integrates with your repository and can be automated to build out issues based on analysis. At GitHub, for instance, we have a SAST tool called Dependabot that analyzes all dependencies for any known security vulnerabilities–and it's directly integrated into every repository on the platform.

- **Dynamic application security testing (DAST):** DAST is used to imitate malicious attacks on an application to find any potential vulnerabilities that might risk its real-world security. DAST tools typically analyze applications in pre-production environments to help DevSecOps practitioners identify any possible security flaws before they make it into production. These flaws typically include underlying issues attackers can exploit to run SQL injection attacks and **cross-site scripting** (**XSS**) attacks, among other things. The best DAST tools will integrate with your CI/CD platform of choice so you can automate their deployment within the wider SDLC.

- **Interactive application security testing (IAST):** IAST solutions are used to identify and profile risks and vulnerabilities in running applications—most often earlier in the SDLC before a release is made. These solutions leverage software instrumentation to monitor and collect information in pre-production environments through manual and automated tests. The best IAST solutions will include **software composition analysis** (**SCA**) tools to identify any open-source component vulnerabilities.

- **Container image scanning:** Due to their lightweight architectures, containers have made it simpler for DevOps organizations to build, test, deploy, and update applications in a fast and flexible manner. But large-scale container environments also introduce security risks due to the number of surface areas and potential for vulnerabilities. To mitigate against any risks, DevSecOps practitioners will leverage container scanning tools to identify issues in the container registry, scan container clusters at runtime, and prevent vulnerabilities from making it into production. Look for tools that can be integrated into your CI/CD pipeline and automated to run at specific points in your SDLC before a deployment—including the build, integration, and packaging stages.

Monitoring tools

Monitoring is a core part of a successful DevOps practice and a critical way to both understand and detect any potential issues before they make it to production—and surface any issues that may show up in production.

Continuous monitoring

Not so long ago, monitoring was costly. Tools would take up precious system resources and require manual intervention. Moreover, the data these tools provided would often take time to parse through and act upon. As a result, organizations typically only monitored mission-critical processes such as coding issues and production-level performance.

Today, collecting data is much easier due to more advanced tooling—but the amount of data has also vastly increased. That means organizations now need to determine how best to manage, interpret, and act upon much larger volumes of data.

Continuous monitoring is a practice that seeks to solve this issue by building monitoring into every part of the SDLC. Its primary goal is to enable the rapid detection of any potential issues and provide real-time feedback.

A continuous monitoring practice will leverage a series of tools and an automated series of tests to evaluate new code and the production performance of an application, as well as its underlying infrastructure. The primary goal is to provide an automated, 360-degree view of all systems and ensure the right people know when and where to intervene.

The best continuous monitoring practices often prioritize collecting as much data as possible to audit systems in their entirety and analyze potential operational issues as well as compliance and security risks.

Implementing monitoring tools in your DevOps practice

Just like the move to DevOps itself, establishing a successful DevOps monitoring strategy requires a mix of culture, process, and tooling. And while you can take inspiration from how other organizations manage monitoring, the precise model you adopt will be driven by the unique needs of your organizations and your SDLC.

There are plenty of frameworks that offer guidance as to what data to capture. But knowing where to implement monitoring is a question of optimization. What questions do you need to answer? What data do you need to get those answers? How will you act on that data? Who should be involved?

Capabilities you should look for in DevOps monitoring tools

There is a rich choice of tooling to help you build monitoring into your DevOps practice. The precise products you choose will depend on the shape of your SDLC and your application's infrastructure. But there are two core initial questions you should ask when evaluating monitoring tooling:

Is it actionable? Does the tool integrate back into your DevOps pipeline and with your other tooling to enable you to automate actions and alerts based on its data?

Does it tell you something new? Generating more data is easy but more data demands attention, fills up storage, and needs to be maintained. Choose tools that open up new avenues of monitoring, rather than those that offer marginal gains.

There are several areas where monitoring should be implemented in a DevOps practice, and some are more obvious than others. Tools to monitor the infrastructure and network are used to understand how constraints such as memory and CPU are impacting your application. At the application layer, **application performance monitoring** (**APM**) tools are used to showcase signals about your application's performance. These tools provide insights into how to better optimize your application. GitHub has features for monitoring CI/CD workflows with GitHub Actions, aggregating security findings with GitHub Advanced Security, and providing developer velocity metrics with organizational insights.

Tooling is often the most visible aspect of a DevOps practice. It is a practical manifestation of the DevOps culture and processes that influence every stage of the SDLC. In DevOps, tools are often used to apply automation wherever possible, create feedback loops, and free up organizational resources. By encouraging feedback loops through automated monitoring and reporting tools, DevOps helps teams build more resilient software. When problems arise, DevOps automation and tooling help deploy fixes into production faster than traditional software development practices.

DevOps isn't implemented simply by purchasing a set of tools, but tools that are open and collaborative in nature can foster DevOps principles. State-of-the-art DevOps tools, along with a high-functioning DevOps culture, can provide organizations with an immense competitive advantage.

Conclusion: DevOps as a framework to deliver value

If you ask 10 people to define DevOps, you're likely to get at least five different answers.

Some people might focus on the practical implementation of DevOps—CI/CD, test automation, and so on—and they'll call it a process. Others might call DevOps a methodology with a set of processes that work together under a coherent philosophy. But both definitions miss the larger point: DevOps consists of a set of practices that are adaptable to each business that adopts them.

It's better to understand DevOps as a framework for thinking about how to deliver value through software. It's more than a single methodology or collection of processes. It's fundamentally a set of practices—both cultural and technological.

Tooling is often the most visible aspect of DevOps, but DevOps is not a single tool. A DevOps transformation starts with a cultural change intended to shift how we think about expertise and responsibility. Information is the currency of a DevOps pipeline. The key consideration is whether that information can flow freely between stages, which is facilitated by culture and process as much as it is by technology.

Once the foundation of a high-functioning DevOps culture is in place, tooling can absolutely influence the overall success of a DevOps practice in an organization. The best DevOps culture, with continuous learning feedback loops, requires tooling capable of allowing cohesive flow from one DevOps stage to the next. The combination of people, processes, cultural practices, and technologies all working together is the indicator of a successful DevOps practice. Only when each pillar is working in unison can DevOps' benefits of faster delivery, increased quality, and more scalable products be achieved.

Resources

- What is the DevOps Model? Exploring foundational practices in DevOps
- DevOps fundamentals: Defining DevOps principles
- Should we think of DevOps as a methodology?
- What is a DevOps pipeline? A complete guide
- The fundamentals of continuous integration in DevOps
- The fundamentals of continuous deployment in DevOps
- What is containerization?
- DevSecOps explained
- A guide to DevOps tools and DevOps automation toolchains
- DevOps monitoring tools: Automating your DevOps monitoring processes

Build your DevOps practice on GitHub

GitHub is an integrated platform that takes companies from idea to planning to production, combining a focused developer experience with powerful, fully managed development, automation, and test infrastructure.

GitHub's comprehensive suite of tools brings the entire DevOps pipeline into a single toolset. To assist in planning, GitHub Issues and Projects provide an innovative, developer-first approach to work management. Once the idea has been planned, developers can start working on code in an isolated development container that is identical to their co-workers with GitHub Codespaces. When the feature is ready to be reviewed, pull requests within GitHub allow developers to collaborate and receive real-time feedback. GitHub Actions is the automation platform used for CI, CD, and automating anything and everything in between. GitHub Packages is used to store, manage, and distribute software packages. To keep your code secure and secrets out of source control without disrupting developers' flow, leverage the GitHub Advanced Security toolset. By using GitHub and its features, every stage of the DevOps pipeline can be enhanced.

As the world's largest and most advanced development platform, GitHub helps millions of developers and companies collaborate, build, and deliver faster. And with thousands of DevOps integrations, you can build with the tools you know from day one—or discover new ones. Get the complete developer platform today and join over 83 million developers and 4 million organizations developing software on GitHub.

Got questions about GitHub Enterprise?

We can help.

Visit our GitHub Enterprise page or connect with our sales team.