**12**

**DevOps and Solution Architecture Framework**

In traditional environments, the development team and the IT operations team work in silos. The development team gathers requirements from business owners and develops the applications. System administrators are solely responsible for operations and for meeting uptime requirements. These teams generally do not have any direct communication during the development life cycle, and each team rarely understands the processes and needs of the other team.

Each team has its own set of tools, processes, and redundant approaches, which sometimes results in conflict. For example, the development and **quality assurance** (**QA**) teams could be testing the build on a specific patch of the **operating system** (**OS**). However, the operations team deploys the same build on a different OS version in the production environment, causing issues and delays in the delivery.

DevOps is a methodology that promotes collaboration and coordination between developers and operational teams to deliver products or services continuously. This approach is constructive in organizations where the teams rely on multiple applications, tools, technologies, platforms, databases, devices, and so on in the process of developing or delivering a product or service.

Although there are different approaches to the DevOps culture, all are about achieving a common goal. DevOps is about delivering a product or service in the shortest amount of time by increasing operational efficiency through shared responsibilities. DevOps helps to deliver without compromising on quality, reliability, stability, resilience, or security.

In this chapter, you will learn about the following DevOps topics:

* Introducing DevOps
* Understanding the benefits of DevOps
* Understanding the components of DevOps
* Introducing DevOps in security
* Combining DevSecOps and CI/CD
* Implementing a CD strategy
* Implementing continuous testing in the CI/CD pipeline
* Using DevOps tools for CI/CD
* Implementing DevOps best practices
* Building DevOps and DevSecOps in the cloud

By the end of this chapter, you will have learned about the importance of DevOps in application deployment, testing, and security. You will also learn DevOps best practices and different tools and techniques to implement them.

**Introducing DevOps**

In a **DevOps** (short for **development and operations**) approach, the development team and the operations team work collaboratively during the build and deployment phases of the software development life cycle, sharing responsibilities, and providing continuous feedback. The software builds are tested frequently throughout the build phase on production-like environments, allowing early detection of defects.

Sometimes, you will find a software application development and its operations are handled by a single team, where engineers work across the entire application life cycle, from development and deployment to operations. Such a team needs to develop a range of skills that are not limited to a single function. Application testing and security teams may also work more closely with the operations and development teams, from the inception to the production launch of an application.

Speed enables organizations to stay ahead in the competition and address customer requirements quickly. Good DevOps practices encourage software development engineers and operations professionals to work better together. This results in closer collaboration and communication, leading to a shorter **time to market**, reliable release, improved code quality, and better maintenance.

Developers benefit from feedback provided by the operations teams and create strategies for testing and deployment.

System administrators don't have to implement defective or untested software in production environments because they participate in the *build phase*. As all stakeholders in the software development and delivery life cycle collaborate, they can also evaluate the tools they intend to use at each step of the process, verify compatibility between the devices, and determine whether any tools can be shared across the teams.

DevOps is a combination of culture and practices. It requires organizations to change their culture by breaking down the barriers between all teams in the product development and delivery life cycle. DevOps is not just about development and operations; instead, it involves the entire organization, including management, business/application owners, developers, QA engineers, release managers, the operations team, and system administrators.

DevOps is gaining popularity as the preferred operating culture, especially for organizations that deal with cloud or distributed computing. Let's learn about some of the various benefits of DevOps and why it is important for your application workload.

**Understanding the benefits of DevOps**

The goal of DevOps is a **CI/CD** (**continuous integration and continuous deployment**) model that is repeatable, reliable, stable, resilient, and secure. These properties improve operational efficiency. To achieve this goal, teams must collaborate and get involved in the development and delivery process. All technical team members should have experience with the processes and tools involved in the development pipeline. A mature DevOps process provides benefits, as shown in the following diagram:



Figure 12.1: Benefits of DevOps

These benefits of DevOps are detailed further here:

* **Speed**: Releasing product features quickly helps to accommodate changing business needs of your customers and expand your market. A DevOps model enables an organization to achieve results faster.
* **Fast delivery**: DevOps processes facilitate more efficiency by automating end-to-end pipelines, from code build to code deploy and production launch. Rapid delivery helps you to innovate faster. The faster release of bug fixes and features allows you to gain a competitive edge.
* **Reliability**: DevOps processes provide all checks to ensure delivery quality and safe application updates rapidly. DevOps practices such as CI and CD embed automation testing and security checks for a positive end-user experience.
* **Scalability**: DevOps helps to scale your infrastructure and application on an on-demand basis by including automation everywhere.
* **Collaboration**: The DevOps model builds a culture of ownership whereby the teams consider their actions. The operations and dev teams work together in a shared responsibility model. Collaboration simplifies the process and increases efficiency.
* **Security**: In an agile environment, making frequent changes requires stringent security checks. The DevOps model automates security and compliance best practices, monitors them, and takes corrective action in an automated way.

The DevOps model optimizes the productivity of the development team and the reliability of system operations. As teams closely collaborate, this helps to increase efficiency and improve quality. Teams take full ownership of the services they deliver, often beyond the traditional scope of their roles, and develop thinking from a customer point of view to solve any issue.

**Understanding the components of DevOps**

DevOps tools and automation bring together development and system operations. The following are critical components of a DevOps practice:

* CI/CD
* Continuous monitoring and improvement
* Infrastructure as code
* Configuration management

A best practice across all the elements is **automation**. Automating processes allows you to efficiently perform these operations in a fast, reliable, and repeatable fashion. Automation can involve scripts, templates, and other tools. In a thriving DevOps environment, infrastructure is managed as code. Automation enables DevOps teams to set up and tune test and production environments rapidly. Let's explore more details about each component.

**Continuous integration/continuous deployment**

In **CI** (**Continuous Integration**), developers commit code frequently to a code repository. The code is built frequently. Each build is tested using automated unit tests and integration tests. In **CI** (**Continuous Integration**), you go a step further and deploy your code frequently in production. Builds are deployed to test environments and are tested using automated and possibly manual tests. Successful builds pass tests and are deployed to staging or production environments. The following diagram illustrates the impact of CI versus CD in the software development life cycle:

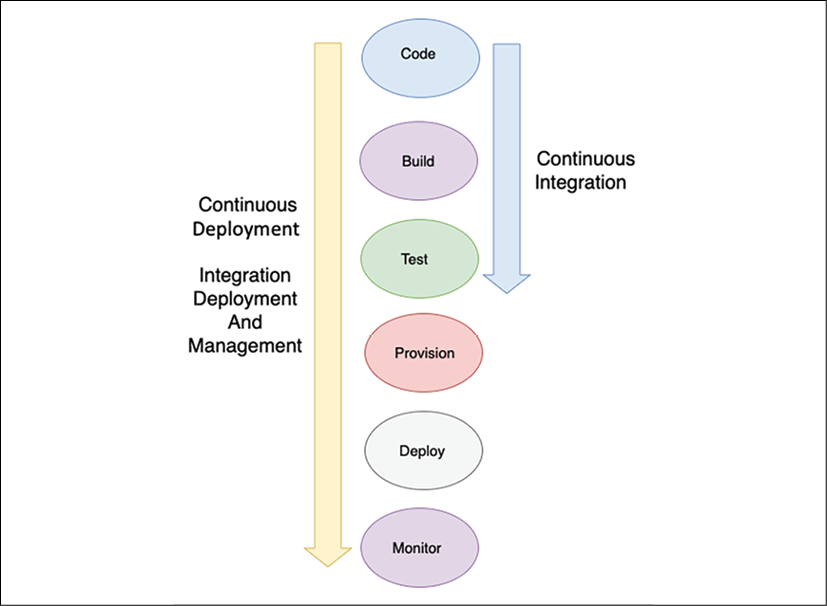


Figure 12.2: CI/CD

As shown in the preceding diagram, CI refers to the building and unit testing stages of the software development life cycle. Every update that is committed in the code repository creates an automated build and test. CD is an essential aspect of CI that extends the CI process further to deploy the build in production. In CI/CD practices, several people work on the code. They all must use the latest working build for their efforts. Code repositories maintain different versions of the code and also make the code accessible to the team. You check out the code from the repository, make your changes or write new code in your local copy, compile and test your code, and frequently commit your code back to the main repository.

CI automates most of the software release process. It creates an automated flow that builds, tests, and then stages the update. However, a developer must trigger the final deployment to a live production environment that is *not automated*. It expands upon CD by deploying all code changes to a testing environment and/or a production environment after the build stage. If CD is implemented correctly, developers will always have a tested and deployment-ready build.

The concepts in the following diagram illustrate everything related to the automation of an application, from code commits into a code repo to the deployment pipeline. It shows an end-to-end flow from the build to the production environment, where the developer checks in the code change in the code repository, which is pulled by the CI server. The CI server triggers the build to create a deployment package with new application binaries and corresponding dependencies. These new binaries are deployed in a targeted development or testing environment. Also, binaries get checked into the artifact repository for safe version-controlled storage:

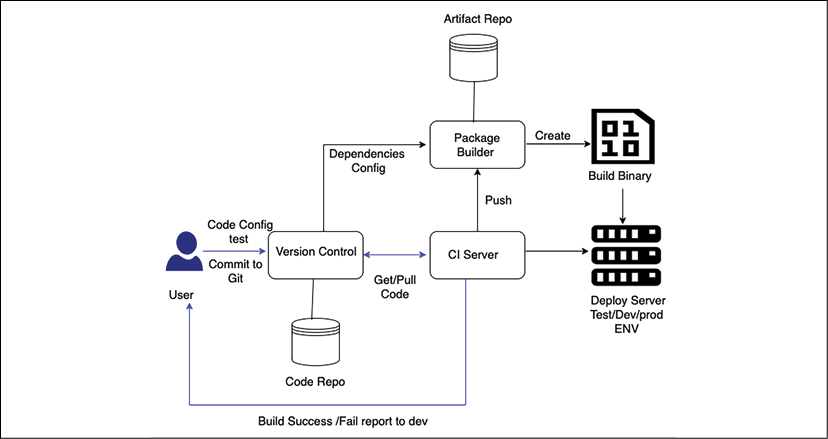


Figure 12.3: CI/CD for DevOps

In CI/CD, software development life cycle phases such as code, build, deploy, and test are automated using the DevOps pipeline. The deploy and provision phase needs to be automated using **Infrastructure as Code**(**IaC**) scripts. Monitoring can be automated with various monitoring tools.

A robust CD pipeline also automates the provisioning of infrastructure for testing and production environments and enables monitoring and management of test and production environments. CI/CD provides multiple benefits to the team, such as improving developer productivity by saving time on building, testing, and deploying code. It helps the dev team to detect and fix bugs quickly and launch feature updates faster in the production environment.

CD does not mean that every change committed by the developer goes into production. Instead, it means that every change is *ready* to go into production. When the changes are staged and tested in the stage environment, a manual approval process initiates and gives a green signal to deploy to production. Thus, in CD, deploying to production becomes a business decision and is still automated with tools.

**Continuous monitoring and improvement**

Continuous monitoring helps us to understand application and infrastructure performance impacts on the customer. By analyzing data and logs, you can learn how code changes impact users. Active monitoring is essential in the era of 24/7 services and constant updates to both applications and infrastructure. You can be more proactive about monitoring services by creating alerts and performing real-time analysis.

You can track various metrics to monitor and improve your DevOps practice. Examples of DevOps-related metrics are as follows:

* **Change volume**: This is the number of user stories developed, the number of lines of new code, and the number of bugs fixed.
* **Deployment frequency**: This indicates how often a team is deploying an application. This metric should generally remain stable or show an upward trend.
* **Lead time from development to deployment**: The time between the beginning of a development cycle to the end of deployment can be used to identify inefficiencies in the intermediate steps of the release cycle.
* **Percentage of failed deployments**: The percentage of failed deployments, including the number of deployments that resulted in outages, should be low.

This metric should be reviewed in conjunction with the change volume. Analyze potential points of failure if the change volume is low but the number of failed deployments is high.

* **Availability**: Track how many releases caused failures that possibly resulted in violations of **Service-Level Agreements** (**SLAs**). What is the average downtime for the application?
* **Customer complaint volume**: The number of complaint tickets filed by customers indicates the quality of your application.
* **Percentage change in user volume**: The number of new users signing up to use your application and the resulting increase in traffic can help you scale your infrastructure to match the workload.

After you deploy builds to the production environment, it is essential to monitor your application's performance continuously. As we discussed automating environments, let's explore more details on **IaC** (**Infrastructure as Code**).

**Infrastructure as Code**

Provisioning, managing, and even deprecating infrastructure is a costly activity in terms of human effort. Furthermore, repeated attempts to build and modify environments manually can be fraught with errors. Whether working from prior experience or a well-documented runbook, the tendency for a human to make a mistake is a statistical probability.

We can automate the task of creating a complete environment. Task automation can help to complete repetitive tasks and provide significant value effortlessly. With IaC, we can define our infrastructure in the form of **templates**. A single template may consist of a part or the entirety of an environment. More importantly, this template can be used repeatedly to create the same environment again.

In IaC, infrastructure is spun up and managed using code and CI. An IaC model helps you interact with infrastructure programmatically at scale and avoid human errors by automating resource configuration. That way, you can work with infrastructure the same way you would with code by using code-based tools. As the infrastructure is managed through code, the application can be deployed using a standardized method, and any patches and versions can be updated repeatedly without any errors.Some of the most popular IaC scripting tools are Ansible, Terraform, Azure Resource Manager, Google Cloud Deployment Manager, Chef, Puppet, and AWS CloudFormation.

The following is a code sample from AWS CloudFormation, which provides infrastructure as code capability to automated infrastructures on the AWS cloud platform.

{

"AWSTemplateFormatVersion" : "2010-09-09",

"Description" : "Create a S3 Storage with parameter to choose own bucket name",

"Parameters": {

"S3NameParam" : {

"Type": "String",

"Default" : "architect-book-storage",

"Description" : "Enter the S3 Bucket Name",

"MinLength" : "5",

"MaxLength" : "30"

}

},

"Resources" : {

"Bucket" : {

"Type" : "AWS::S3::Bucket",

"DeletionPolicy" : "Retain",

"Properties" : {

"AccessControl" : "PublicRead",

"BucketName" : {"Ref" : "S3NameParam" },

"Tags" : [ {"Key" : "Name" , "Value" : "MyBucket"} ]

}

}

},

"Outputs" : {

"BucketName" : {

"Description" : "BucketName" ,

"Value" : { "Ref" : "S3NameParam"}

}

}

}

The preceding code creates Amazon S3 object storage with the option for the user to provide their choice of storage name as shown below:

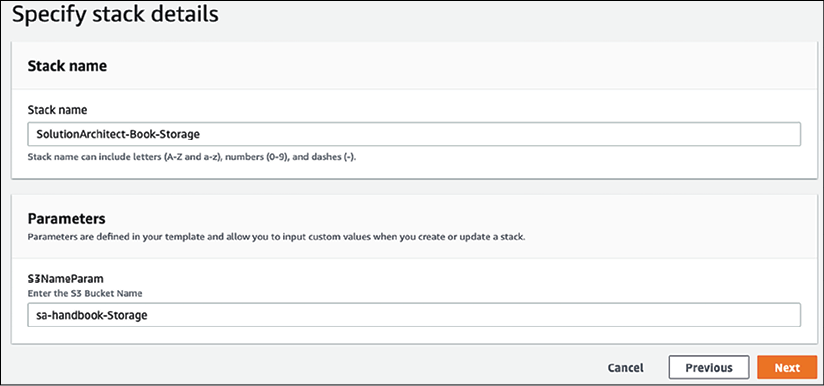


Figure 12.4: Infrastructure as code using AWS CloudFormation

After execution of the code, Amazon S3 object storage gets created as you can see in **Outputs**:

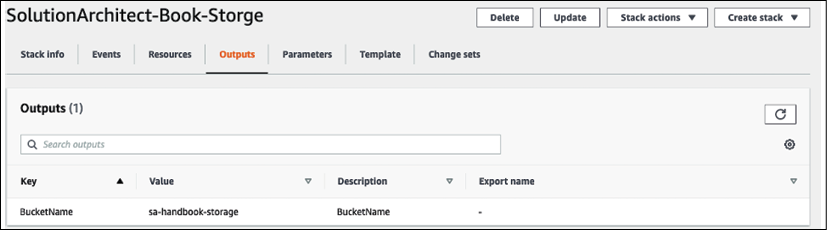


Figure 12.5: Automated AWS S3 object storage creation using AWS CloudFormation

The provided code can be used by multiple teams to create any amount of Amazon S3 storage. As data is so important, the admin chose to add bucket "DeletionPolicy": "Retain", which makes sure storage doesn't get deleted when infrastructure comes down and data is safe. You can see how you can implement standardization, consistency, and compliance across organizations using IaC. Configuration management is another important aspect of the DevOps process. Let's learn more about it.

**Configuration management**

**Configuration management**(**CM**)is the process of using automation to standardize resource configurations across your entire infrastructure and applications. CM tools such as Chef, Puppet, and Ansible can help you manage IaC and automate most system administration tasks, including provisioning, configuring, and managing IT resources. By automating and standardizing resource configurations across the development, build, test, and deployment phases, you can ensure consistency and eliminate failures caused by misconfiguration.

CM can also increase the productivity of your operations by allowing you to automatically deploy the same configuration to hundreds of nodes at the push of a button. CM can also be leveraged to deploy changes to configurations.

Although you can use registry settings or databases to store system configuration settings, a configuration management application allows you to maintain version control as well, in addition to storage. CM is also a way to track and audit configuration changes. If necessary, you can even maintain multiple versions of configuration settings for various versions of your software.

CM tools include a controller machine that manages server nodes. For example, Chef requires a client agent application installed on each server to manage, and a master Chef application installs on the controller machine. Puppet also works the same way with a centralized server. However, Ansible has a decentralized approach and doesn't require installing agent software on the server nodes. The following table shows a high-level comparison between the popular configuration management tools:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Ansible | Puppet | Chef |
| Mechanism | Controller machine applies changes to servers using **Secure Shell** (**SSH**) | Master synchronizes changes to Puppet node | Chef workstation looks for changes in Chef servers and pushes them to the Chef node |
| Architecture | Any server can be the controller | Centralized control by Puppet master | Centralized control by Chef server |
| Script Language | YAML | Domain-specific on Ruby | Ruby |
| Scripting Terminology | Playbook and roles | Manifests and modules | Recipes and cookbooks |
| Test Execution | Sequential order | Non-sequential order | Sequential order |

CM tools provide a domain-specific language and set of features for automation. Some of these tools have a steep learning curve whereby the team has to learn the tool. AWS provides a managed platform called OpsWorks to manage Chef and Puppet in the cloud. It provides various attributes to manage IT infrastructure through automation as shown below:

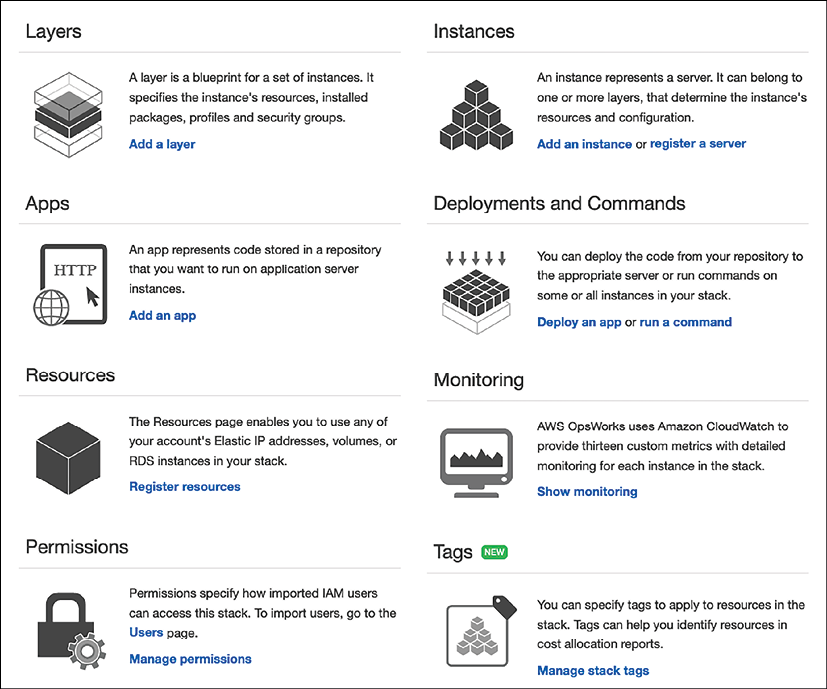


Figure 12.6: AWS OpsWorks service capabilities for managed Chef and Puppet

Security has become a priority for any organization, so complete automation security is the need of the hour. Organizations are moving to tight security implementations and monitoring to avoid human error, using the DevOps process popularly known as **DevSecOps**. Let's explore **DevSecOps** (short for **development, security, and operations**) in the next section.

**Introducing DevSecOps**

We are now more focused on security than ever. In many situations, security is the only way to win customer trust. DevSecOps is about the automation of security and the implementation of security at scale. The development team is constantly making changes, and the DevOps team is publishing them in production (changes are often customer-facing). DevSecOps is required to ensure application security in the overall process.

DevSecOps is not there to audit code or CI/CD artifacts. Organizations should implement DevSecOps to enable speed and agility, but not at the expense of validating security. The power of automation is to increase product feature launch agility while remaining secure by implementing the required security measures. A DevSecOps approach results in built-in security and is not applied as an afterthought. DevOps is about adding efficiency to speed up the product launch life cycle, while DevSecOps validates all building blocks without slowing the life cycle.

To institute a DevSecOps approach in your organization, start with a solid DevOps foundation across the development environment, as security is everyone's responsibility. To create collaboration between development and security teams, you should embed security in the architecture design from inception. To avoid any security gaps, automate continuous security testing and build it into the CI/CD pipeline. To keep track of any security breach, apply to extend monitoring to include security and compliance by monitoring for drift from the design state in real time. Monitoring should enable alerting, automated remediation, and removing non-compliant resources.

Codifying everything is a basic requirement that opens up infinite possibilities. The goal of DevSecOps is to keep the pace of innovation, which should meet the pace of security automation. A scalable infrastructure needs scalable security, requiring automatic incident response remediation to implement continuous compliance and validation.

**Combining DevSecOps and CI/CD**

A DevSecOps practice needs to be embedded with every step of the CI/CD pipeline. DevSecOps ensures the security of the CI/CD pipeline by managing the right access and roles assigned to each server and making sure the build servers such as Jenkins are hardened to be protected from any security glitch. In addition to that, we need to make sure that all artifacts are validated and code analysis is in place. It's better to be ready for incident response by automating continuous compliance validation and incident response remediation.

The following diagram provides us with multiple stages to test security boundaries and catch security issues and compliance with policies as early as possible:

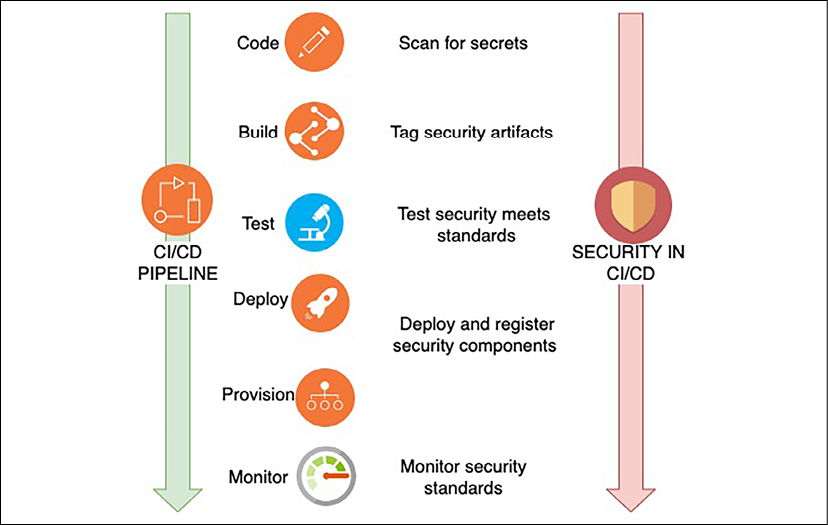


Figure 12.7: DevSecOps and CI/CD

At each integration point, you can identify different issues, as illustrated in the preceding diagram:

* In the coding phase, scan all code to ensure no secret key or access key is hardcoded in between code lines.
* During the build, include all security artifacts such as the encryption key and access token management, and tag them for easy identification.
* During the test, scan the configuration to make sure all security standards are met by test security.
* In the deploy and provision phase, make sure all security components are registered. Perform a checksum to make sure there are no changes in the build files. A checksum is a technique used to determine the authenticity of received files. Operating systems provide a checksum command to validate the file and make sure no changes are made during file transfer.
* Monitor all security standards during the monitoring phase. Perform continuous audit and validation in an automated way.

To identify security vulnerabilities at various stages, you can integrate multiple tools into DevSecOps pipelines and aggregate the vulnerability findings. **Application security testing**(**AST**) using tools that automate the testing, analysis, and reporting of security vulnerabilities is essential for application development. AST can be broken down into the following four categories to scan security vulnerabilities in software applications:

* **Software composite analysis**(**SCA**): SCA evaluates the open-source software's security, license compliance, and code quality in a codebase. SCA attempts to detect publicly disclosed vulnerabilities contained within a project's dependencies. Popular SCA tools are OWASP Dependency-Check, Synopsys' Black Duck, WhiteSource, Synk, and GitLab.
* **Static application security testing**(**SAST**):SAST scans an application before the code is compiled. SAST tools give developers real-time feedback as they code, helping them fix issues before passing the code build phase. It is a white-box testing methodology that analyzes source code to find security vulnerabilities that make your applications susceptible to attack. The best thing about SAST is that it can be introduced very early on in the DevOps cycle, during coding, as it does not require a working application and can take place without code being executed. Popular SAST tools are SonarQube, PHPStan, Coverity, Synk, Appknox, Klocwork, CodeScan, and Checkmarx.
* **Dynamic application security testing**(**DAST**):DAST looks for security vulnerabilities by simulating external attacks on an application while the application is running. It attempts to penetrate an application from the outside by checking its exposed interfaces for vulnerabilities and flaws—this type of black-box security testing is also known as a web application vulnerability scanner. Popular DAST tools are OWASP ZAP, Netsparker, Detectify Deep Scan, StackHawk, Appknox, HCL AppScan, GitLab, and Checkmarx.
* **Interactive application security testing**(**IAST**): IAST analyzes code for security vulnerabilities while the app is run by an automated test or activity validating application functionality. IAST tools report vulnerabilities in real time and do not add extra time to your CI/CD pipeline. IAST tools are deployed in a QA environment to implement automated functional tests. Popular IAST tools are GitLab, Veracode, CxSAST, Burp Suite, Acunetix, Netsparker, InsightAppSec, and HCL AppScan.

You will learn about integrating some of the above tools in the DevOps pipeline later in the chapter, under the *Building DevOps and DevSecOps in the cloud* section. DevSecOps CI/CD gives us confidence that the code is validated against the corporate security policy.

It helps to avoid any infrastructure and application failure in later deployment due to different security configurations. DevSecOps maintains agility and ensures security at scale without affecting DevOps' pace of innovation. Let's learn about the CD strategy in the DevOps pipeline.

**Implementing a CD strategy**

CD provides seamless migration of the existing version to the new version of the application. Some of the most popular techniques to implement through CD are as follows:

* **In-place deployment**: Update application in a current server
* **Rolling deployment**: Gradually roll out the new version in the existing fleet of servers
* **Blue-green deployment**: Gradually replace the existing server with the new server
* **Red-black deployment**: Instant cutover to the new server from the existing server
* **Immutable deployment**: Stand up a new set of servers altogether

Let's explore each option in more detail.

**In-place deployment**

In-place deployment is a method of rolling out a new application version on an existing fleet of servers. The update is done in one deployment action, thereby requiring some degree of downtime. On the other side, there are hardly any infrastructure changes needed for this update. There is also no need to update existing **Domain Name System** (**DNS**) records. The deployment process itself is relatively quick. If the deployment fails, redeployment is the only option for restoration.

As a simple explanation, you are replacing the existing application version (v1) on the application infrastructure with the new version (v2). In-place updates are low-cost and fast to deploy.

**Rolling deployment**

With a rolling deployment, the server fleet is divided into groups, so it doesn't need to be updated simultaneously. The deployment process runs both old and new software versions on the same server fleet but with different subgroups. A rolling deployment approach helps to achieve zero downtime. If a new version deployment fails, then only a subset of servers is impacted from the entire fleet, and the risk is minimal because half of the fleet will still be up and running. A rolling deployment helps to achieve zero downtime; however, deployment time is little more than in-place deployment.

**Blue-green deployment**

The idea behind blue-green deployment is that your blue environment is your existing production environment carrying live traffic. In parallel, you provision a green environment, which is identical to the blue environment other than the new version of your code. When it's time to deploy, you route production traffic from the blue environment to the green environment. If you encounter any issues with the green environment, you can roll it back by reverting traffic to the original blue environment. DNS cutover and swapping auto-scaling groups are the two most common methods to re-route traffic in blue-green deployment.

Using auto-scaling policies, you can gradually replace existing instances with instances hosting the new version of your application as your application scales out. This option is best used for minor releases and small code changes. Another option is to leverage DNS routing to perform sophisticated load balancing between different versions of our application.

As illustrated in the following diagram, after creating a production environment that hosts the new version of our application, you can use the DNS route to shift a small portion of traffic to the new environment:

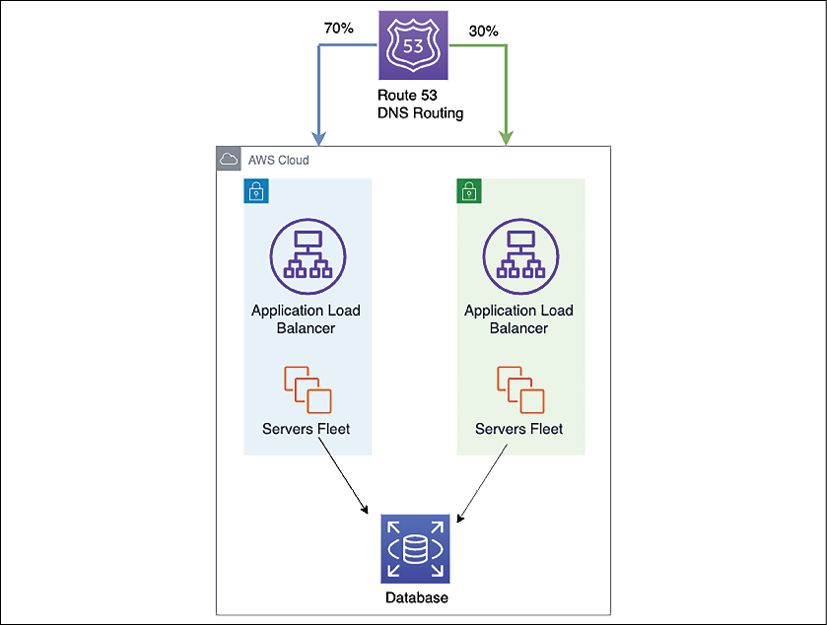


Figure 12.8: Blue-green deployment DNS gradual cutover

Test the green environment with a fraction of production traffic; this is called **canary analysis**. If the environment has functional issues, you'll be able to tell right away and switch traffic back before impacting your users significantly. Continue to gradually shift traffic, testing the ability of the green environment to handle the load. Monitor the green environment to detect issues, providing an opportunity to change traffic back, thus limiting the blast radius. Finally, when all the metrics are right, decommission the blue environment and release the resources.

Blue-green deployment helps to achieve zero downtime and provides easy rollback. You can customize the time to deploy as per your needs.

**Red-black deployment**

In red-black deployment, before standing up a new version of a system, first, perform canary testing. The canary replaces around 1% of its existing production system with the latest version of the application and monitors the newest version for errors. If the canary clears this initial test, the system is deemed ready for deployment.

In preparation for the switchover, a new version of the system stands up side by side with the old version of the system. The initial capacity of the new system is set manually by examining how many instances are currently running in production and setting this number as the desired capacity for the new auto-scaling group. Once the new system is up and running, both systems are red. The current version is the only version accepting traffic.

The system is then cut over from the existing version to the new version using the DNS service. At this point, the old version is regarded as black; it is still running but is not receiving any traffic. If any issues are detected with the new version, reverting becomes as simple as pointing the DNS server back to the old version's load balancer.

Red-black deployment is also known as **dark launch** and is slightly different from blue-green deployment. In red-black deployment, you do sudden DNS cutover from the old version to the new version, while in blue-green deployment, the DNS gradually increases traffic to the new version. Blue-green deployments and dark launches can be combined to deploy both versions of software side by side. Two separate code paths are used, but only one is activated. A feature flag activates the other code path. This deployment can be used as a beta test where you can explicitly enable the new features.

**Immutable deployment**

An immutable or disposable upgrade is an easier option if your application has unknown dependencies. An older application infrastructure that has been patched and re-patched over time becomes more and more difficult to upgrade. This type of upgrade technique is more common in an immutable infrastructure.

During the new release, a new set of server instances are rolled out by terminating older instances. For disposable upgrades, you can set up a cloned environment with deployment services such as Chef, Puppet, Ansible, and Terraform or use them combined with an auto-scaling configuration to manage the updates.

In addition to downtime, you need to consider the cost while designing your deployment strategy. Consider the number of instances you need to replace and your deployment frequency to determine the cost. Use the approach that best fits, taking your budget and downtime into consideration.

In this section, you learned about various CD strategies that help you to make your application release more efficient and hassle-free. You need to perform application testing at every step for high-quality delivery, which often requires significant effort. A DevOps pipeline can help you automate the testing process and increase the quality and frequency of feature releases. Let's learn more about continuous testing in the CI/CD pipeline.

**Implementing continuous testing in the CI/CD pipeline**

DevOps is key for the continually changing business scenarios based on customer feedback, demand for new features, or shifts in market trends. A robust CI/CD pipeline ensures further features/feedback are incorporated in less time, and customers get to use the new features faster.

With frequent code check-ins, having a good testing strategy baked into your CI/CD pipeline ensures you close that feedback loop with quality. Continuous testing is essential in balancing the CI/CD pipeline. While adding software features rapidly is good, ensuring that the features adhere to good quality is achieved by continuous testing.

Unit tests form the most significant amount of your testing strategy. They typically run on the developer's machine and are the fastest and cheapest. A general rule of thumb is to incorporate70% of your testing efforts in unit testing. Bugs caught at this stage can be fixed relatively quickly, with fewer complexities.

The developer often performs unit tests, and once the code is ready, it is deployed for integration and system testing. These tests require their environments and sometimes separate testing teams, which makes the testing process costlier. Once the team ensures that all intended features are working as expected, the operations team needs to run performance and compliance tests. These tests need production-like environments and are costly. Also, **user acceptance testing** (**UAT**) needs a replica of production-like environments, causing more expense.

As illustrated in the following diagram, developers perform unit tests to test code changes/new features in the development phase. Testing is usually done on a developer's machine after coding is complete.

It is also recommended to run static code analysis on the code changes and do code coverage, adherence to coding guidelines, and so on. Smaller unit tests with no dependencies run faster. Therefore, the developer can find out quickly if the test has failed:

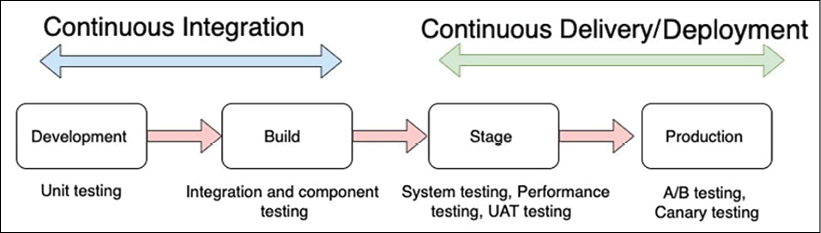


Figure 12.9: Continuous testing in CI/CD

The **build phase** is the first phase to test for integration between different components and individual components themselves. The build phase is also an excellent time to test if the code committed by a developer breaks any existing feature and to perform regression testing.

A **staging environment** is a mirror of the production environment. An end-to-end system test is performed at this stage (the UI, backend logic, and API are tested extensively). Performance testing tests the application performance under a particular workload. Performance tests include load tests and stress tests. UAT is also performed at this stage in readiness for production deployment. Compliance testing is done to test for industry-specific regulatory compliance.

A strategy such as A/B testing or canary analysis is used to test the new application version in the production phase. In A/B testing, the new application version is deployed to a small percentage of production servers and tested for user feedback. Gradually, depending on how well the users receive the new application, the deployment is increased to span all production servers.

**A/B testing**

Often, in software development, it isn't clear which implementation of a feature will be most successful in the real world. An entire computer science discipline—**human/computer interaction** (**HCI**)—is devoted to answering this question. While UI experts have several guidelines to help them design suitable interfaces, the best choice of design often can only be determined by giving it to users and seeing whether they can use the design to complete a given task.

As shown in the following diagram, A/B testing is a testing methodology in which two or more different versions of features are given to different sets of users. Detailed metrics on the usage of each implementation are gathered, and UI engineers examine this data to determine which implementation should be adopted going forward:

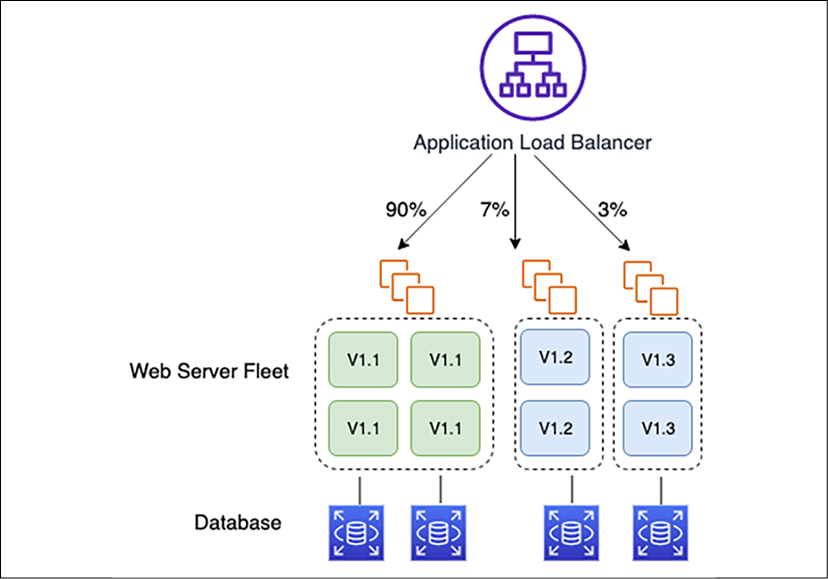


Figure 12.10: Split users by feature experiment using A/B testing

It's easy to launch several different versions of your application, each containing different implementations of a new feature. DNS routing can be used to send the majority of traffic to the current system while also sending a fraction of the existing traffic to the versions of the system running the new features. DNS round-robin resolution is supported by most DNS resolvers and is an effective way to spread incoming traffic.

Load and performance testing are other important factors. For Java-based applications, you can use JMeter to load-test a relational database by issuing **Java Database Connectivity** (**JDBC**) commands. MongoDB can use Mongo-Perf, which can generate a reproducible load on the database and record the response time. You can then hit the components and services that use the database and also simultaneously test the database.

One common way to measure the load on instances is through what is called **micro-benchmarking**. In micro-benchmarking, you measure the performance of a small sub-component of your system (or even a snippet of code) and then attempt to extrapolate general performance data from this test result. In the case of testing a server, you may test a slice of the system on a new instance type and compare that measurement to the same slice measured on your currently running system, which is now using another server type and configuration.

**Using DevOps tools for CI/CD**

To build a CI/CD pipeline, a developer requires various tools. These include a code editor, a source repository, a build server, a deployment tool, and orchestrating an overall CI pipeline. Let's explore some popular technology choices of developer tools for DevOps, both in the cloud and on-premises.

**Code editor**

DevOps is a hands-on coding role, where you often need to write a script to automate the environment. You can use the **ACE editor** or the **cloud-based AWS Cloud9 integrated development environment** (**IDE**). You can use a web-based code editor on your local computer or install a code editor in your local server that connects to the application environments—such as dev, test, and prod—to interact. An environment is where you store your project's files and run the tools to develop your apps. You can save these files locally on the instance or server or clone a remote code repository into your environment. The AWS Cloud9 IDE is the cloud-native IDE provided as a managed service.

The Ace editor lets you write code quickly and easily. It's a web-based code editor but provides performance similar to popular desktop-based code editors such as Eclipse, Vim, and **Visual Studio Code** (**VS Code**), and so on. It has standard IDE features such as live syntax and matching parentheses highlighting, auto-indentation and completion, toggling between tabs, integration with version control tools, and multiple cursor selections. It works with large files, having hundreds of thousands of lines without typing lag. It has built-in support for all popular coding languages and debugging tools, and you can also install your tools. For a desktop-based IDE, VS Code and Eclipse are other popular code editor options that DevOps engineers can choose.

**Source code management**

There are multiple choices available for your source code repository. You can set up, run, and manage your Git server, where you will be responsible for everything.

You can choose to use a hosting service such as GitHub or Bitbucket. If you are looking for a cloud solution, then **AWS CodeCommit** offers a secure, highly scalable, and managed source control system where you can host private Git repositories.

You need to set up authentication and authorization for your code repository to provide access to authorize team members for code to read or write. You can apply data encryption in transit and at rest. When you push into the code repository (git push), it encrypts the data and then stores it. When you pull from the code repository (git pull), it decrypts the data and then sends it back to the caller. The user must be an authenticated user with the proper access level to the code repository. Data can be encrypted in transit by transmitting through encrypted network connections using HTTPS or SSH protocols.

**Continuous integration server**

A CI server is also known as a **build server**. With teams working on multiple branches, it gets complicated to merge back into the main branch. CI, in this scenario, plays a key role. CI server hooks provide a way to trigger the build based on the event when code is committed to the repository. Hooks, which are incorporated in almost every version control system, refer to custom scripts triggered by specified necessary actions in a repository. Hooks can run either on the client side or on the server side.

Pull requests are common for developers to notify and review each other's work before it is merged into common code branches. A CI server provides a web interface to review changes before adding them to the final project. If there are any problems with the proposed changes, the source code can be sent back to the developer to tweak as per the organization's coding requirements.

As shown in the following diagram, server-side hooks in combination with the CI server are used to increase the velocity of integration:

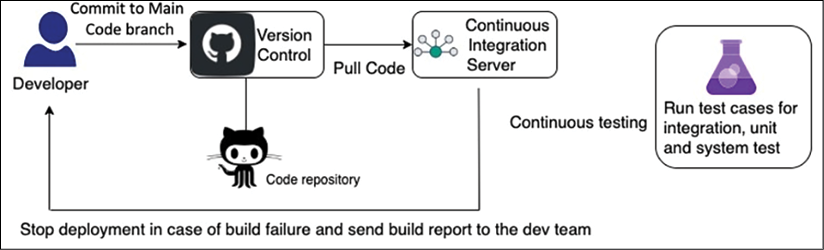


Figure 12.11: Automation of CI

As illustrated in the preceding diagram, using post-receive, you can direct new branches to trigger tests on a CI server to verify that the new build integrates correctly and that all units function correctly. The developer is notified of test failures and then knows to merge their branch with the mainline only after fixing the problems. The developer can build from their branch, test the changes there, and get feedback on how well their changes work before deciding whether to merge their branch into the mainline.

Running integration and unit tests significantly reduces resistance when that branch is merged into the mainline. Hooks can also be customized to test merges into the mainline and block any merges that don't pass. Integration is all accomplished best with a CI server.

Jenkins is the most popular choice to build the CI server. However, you have to maintain security and patching of the server by yourself. For native cloud options and managed services, you can use managed code-build services such as AWS CodeBuild, eliminating the need for server administration and significantly reducing costs with a **pay-as-you-go** model. The service scales as per your demand. Your team is empowered to focus on pushing code and lets a service build all the artifacts.

As illustrated in the following diagram, you can host the Jenkins cluster in the AWS **Elastic Compute Cloud** (**EC2**) server's fleet and auto-scale as per build load:

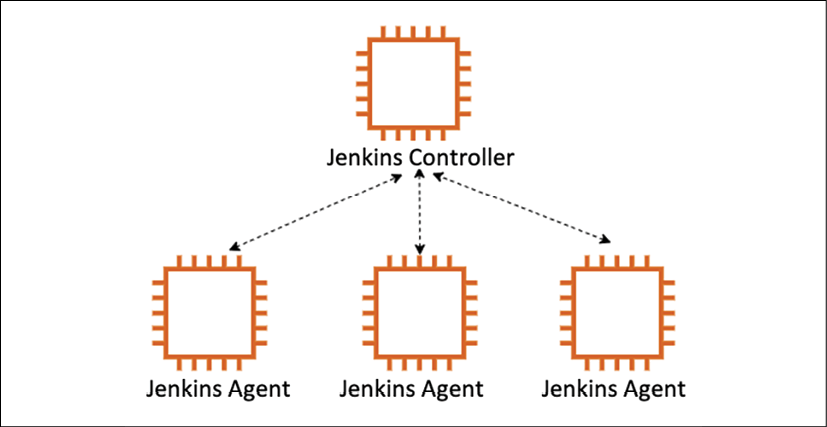


Figure 12.12: Auto-scaling of Jenkins CI servers

The**Jenkins Controller**offload builds to the agent node instance in the case of overload. When the load goes down, the **Jenkins Controller** automatically terminates agent instances.

While a CI server helps you build the correct version of code from a source code repository by collaborating across team members of the development team, code deployment helps the team get code ready for testing and release for end-user consumption. Let's learn about code deployment in more detail.

**Code deployment**

Once your build is ready, you can use the Jenkins server for deployment or choose AWS CodeDeploy as a cloud-native managed service. You can use other popular tools such as Chef or Puppet to create a deployment script. The options for specifying a deployment configuration are as follows:

* **OneAtATime**: Only a single instance in a deployment group at a time installs a new deployment. Suppose a deployment on a given instance fails. In that case, the deployment script will halt the deployment and return an error response detailing the number of successful versus the number of failed installations.
* **HalfAtATime**: Half of the instances in the deployment group install a new deployment. The deployment succeeds if half of the instances successfully install the revision. HalfAtATime can again be a good option for production/test environments where half of the instances are updated to a new revision, and the other half remain available in production at an older revision.
* **AllAtOnce**: Each instance installs the latest revision available whenever it next polls the deployment service. This option is best used for development and test deployments as it has the potential to install a non-functioning deployment on every instance in a deployment group.
* **Custom**: You can use this command to create a custom deployment configuration specifying a fixed number of healthy hosts that must exist in a deployment group at any given time. This option is a more flexible implementation of the OneAtATime option. It allows for the possibility that a deployment may fail on one or two instances that have become corrupt or are improperly configured.

The following diagram illustrates life cycle events during deployment:

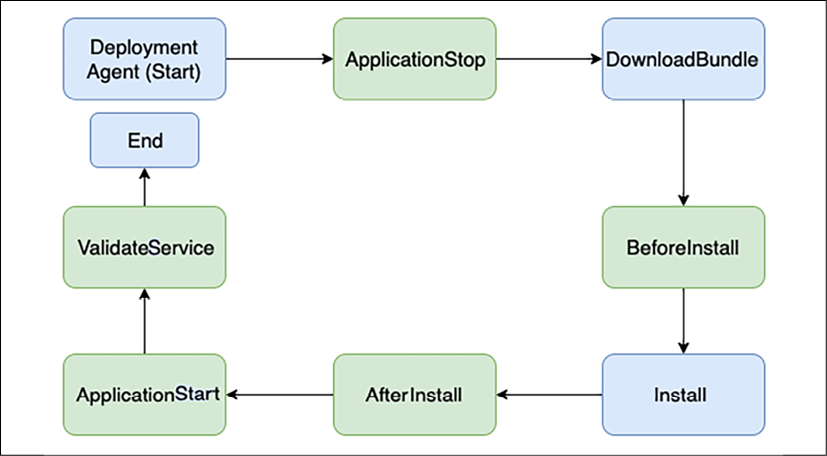


Figure 12.13: Deployment life cycle event

The deployment agent runs through a series of steps to execute a deployment. These steps are called life cycle events. In the preceding diagram, steps highlighted in light boxes can be controlled by human intervention; however, steps highlighted in darker boxes are automated and controlled by a deployment agent. Here are more details about each step:

* **ApplicationStop**: To trigger a deployment, the first requirement is to stop the application server so that traffic stops serving while files are copied. Examples of software application servers are Tomcat, JBoss, or WebSphere servers.
* **DownloadBundle:** After stopping the application server, the deployment agent starts downloading a pre-built deployment bundle from an artifactory such as JFrog Artifactory. The artifactory stores the application binary, which can be deployed and tested for application before the new version launch.
* **BeforeInstall**: The deployment agent triggers pre-install steps such as creating a backup of the current version and any required configuration update via a script.
* **Install**: In this step, deployment agents start the installation—for example, running an Ant or Maven script to install a Java application.
* **AfterInstall**: The deployment agent triggers this step after your application installation is completed. It may include updating post-installation configuration, such as local memory setting and log parameters.
* **ApplicationStart**: In this step, the agent starts the application and notifies the success or failure operations team.
* **ValidateService**: The validation step fires after everything else is done and gives you a chance to do a sanity check on the app. It includes steps such as performing automated sanity tests and integration tests to verify if the new version of the application has been installed properly. The agent also sends a notification to the team when testing is successful.

You have learned about various code deployment strategies and steps as independent components. However, to set up an automated CI/CD pipeline, you need to stitch all the DevOps steps together. Let's learn more about the code pipeline, which can help you build an end-to-end CI/CD pipeline.

**Code pipeline**

The code pipeline is about orchestrating everything together to achieve CD. The entire software release process is fully automated in CD, including build and deployment to the production release. Over some time, with experiments, you can set up a mature CI/CD pipeline. The path to the production launch is automated, thus enabling the rapid deployment of features and immediate customer feedback. You can use cloud-native managed services such as AWS CodePipeline to orchestrate the overall code pipeline or use the Jenkins server.

The code pipeline enables you to add actions to stages in your CI/CD pipeline. Each action can be associated with a provider that executes the action. The code pipeline action's categories and examples of providers are as follows:

* **Source**: Your application code needs to be stored in a central repository with version control called **source code repositories**. Some of the popular code repositories are AWS CodeCommit, Bitbucket, GitHub, **Concurrent Versions System** (**CVS**), **Subversion** (**SVN**), and so on.
* **Build**: The build tool pulls code from the source code repository and creates an application binary package. Some of the popular build tools are AWS CodeBuild, Jenkins, Solano CI, and so on. Once the build is completed, you can store binaries in an artifactory such as JFrog.
* **Deploy**: The deployment tool helps you to deploy application binaries on the server. Some popular deployment tools are AWS Elastic Beanstalk, AWS CodeDeploy, Chef, Puppet, Jenkins, and so on.
* **Test**: Automated testing tools help you to complete and perform post-deployment validation. Some popular test validating tools are Jenkins, BlazeMeter, Ghost Inspector, and so on.
* **Invoke**: You can use an events-based script to invoke activities such as backup and alert. Any scripting language such as a shell script, PowerShell, and Python can be used to invoke various customized activities.
* **Approval**: Approval is an essential step in CD. You can either ask for manual approval by an automated email trigger or approval can be automated from tools.

In this section, you learned about various DevOps tools to manage the **Software Development Life Cycle** (**SDLC**), such as a code editor, a repository, and build, test, and deployment tools. The other tools you need to integrate into DevOps pipelines are continuous logging, continuous monitoring, and operation handling, which you learned in *Chapter 10*, *Operational Excellence Considerations*. As of now, you have learned about various DevOps techniques for each SDLC phase. Let's learn more about best practices and anti-patterns.

**Implementing DevOps best practices**

While building a CI/CD pipeline, consider your need to create a project and add team members to it. The project dashboard provides visibility to the code flow through the deployment pipeline, monitoring the build, triggering alerts, and tracking application activities. The following diagram illustrates a well-defined DevOps pipeline:

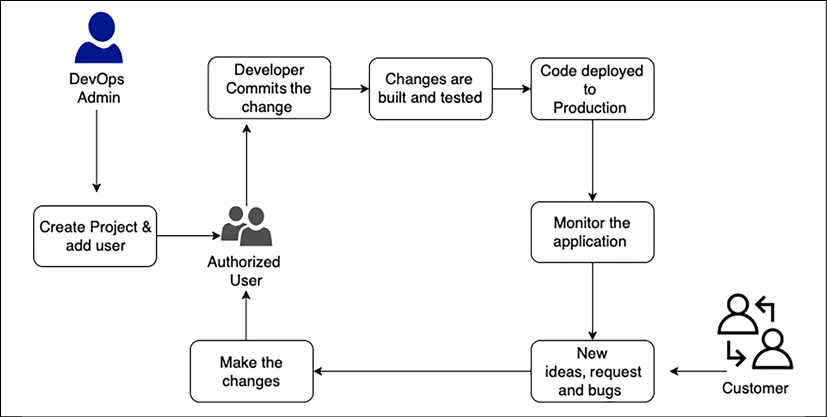


Figure 12.14: CI/CD workflow best practice

Consider the following points while designing the pipeline:

* **The number of stages**: Stages could be development, integration, system, user acceptance, and production. Some organizations also include dev, alpha, beta, and release stages.
* **Types of tests in each stage**: Each stage can have multiple types of tests such as unit tests, integration tests, system tests, UATs, smoke tests, load tests, and A/B tests at the production stage.
* **The sequence of a test**: Test cases can be run in parallel or need to be in sequence.
* **Monitoring and reporting**: Monitor system defects and failures and send notifications as failures occur.
* **Infrastructure provisioning**: Methods to provision infrastructure for each stage.
* **Rollback**: Define the rollback strategy to fall back to the previous version if required.

Having a system that requires manual intervention where it's avoidable slows down your process. So, automating your process using CD will accelerate your process.

Another common anti-pattern is keeping configuration values for a build inside the code itself or even having developers use different tools in their build processes, leading to inconsistent builds between developers. It takes lots of time and effort to troubleshoot why particular builds work in one environment and not in others. To overcome this, it is better to store build configurations outside of code. Externalizing these configurations to tools that keep them consistent between builds enables better automation and allows your process to scale much more quickly. Not using a CD process can lead to last-minute, middle-of-the-night rushes to get a build to work. Design your CD process to *fail fast* to reduce the likelihood of any last-minute surprises.

To apply architecture best practice at each step of application development, the twelve-factor methodology can be used, as recommended by The Twelve-Factor App (<https://12factor.net/>), which is adopted by enterprises for end-to-end development and delivery of web applications. This applies to all coding platforms regardless of programming languages. Nowadays, most applications are built as web apps and utilize a cloud platform. Let's learn about how to build end-to-end DevOps along with security automation in the cloud.

**Building DevOps and DevSecOps in the cloud**

As you have learned in previous sections, building a CI/CD pipeline requires multiple tools, and adding security automation on top of that increases the complexity. Integrating various tools and aggregating the vulnerability findings can be a challenge to do from scratch. A public cloud provider such as AWS provides the flexibility to build DevSecOps pipelines with easy integrations of cloud-native and third-party tools and aggregate security findings.

The following DevSecOps pipeline architecture covers CI/CD practices, including SCA, SAST, and DAST tools to visualize the concepts of security automation in the pipeline:

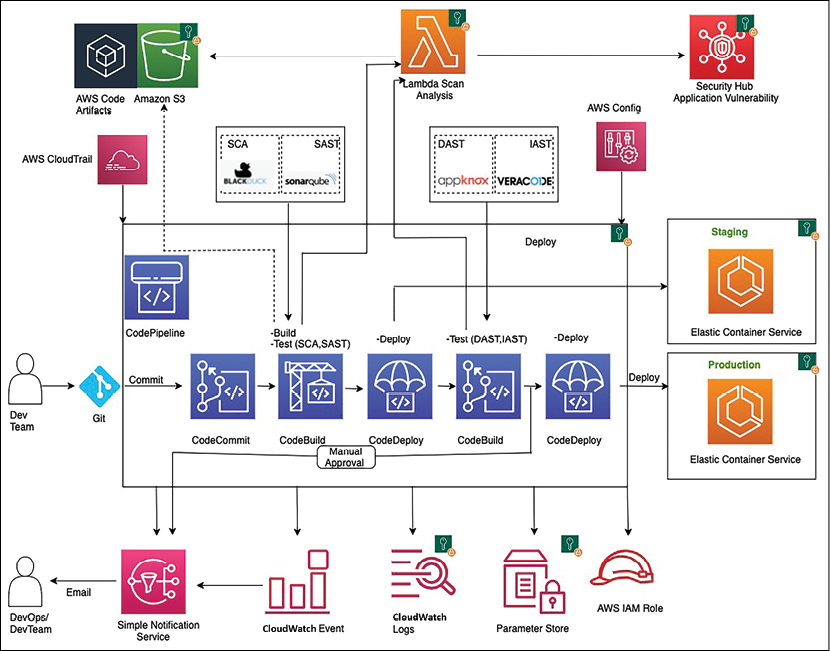


Figure 12.15: DevSecOps CI/CD pipeline architecture in the AWS cloud

As shown in the preceding diagram, the CI/CD pipeline gets triggered when a developer commits the code in GitHub. An event gets generated to start the AWS CodePipeline using AWS CloudWatch. AWS CodePipeline orchestrates the CI/CD pipeline including code commit, build, and deploy. AWS CodeBuild packages the build and uploads the artifacts to AWS CodeArtifact. AWS CodeBuild retrieves the authentication information such as scanning tool tokens from AWS Parameter Store to initiate the scanning.

You need to integrate AST tools in your pipeline to implement DevSecOps. CodeBuild scans the code with an SCA tool such as Synopsys' Black Duck or WhiteSource, and an SAST tool such as SonarQube or Coverity. SCA or SAST devices may detect vulnerabilities that need to post into AWS Security Hub. AWS CodeBuild invokes the Lambda function to consolidate all security findings in one place under AWS Security Hub. You can also add IAST such as Veracode or CxSAST while the app goes through an automated test to validate application functionality. CodeDeploy deploys the code to the staging AWS **Elastic Container Service** (**ECS**) environment if there are no vulnerabilities.

After the deployment succeeds, CodeBuild triggers the DAST scanning with the OWASP ZAP or Appknox tool. Again, if there are any vulnerabilities, CodeBuild invokes the Lambda function, which posts security findings into AWS Security Hub. Suppose DAST finds no security issues. In that case, the build can advance for approval, and the pipeline notifies the approver for action to push the build into the production AWS ECS environment. During the CI/CD pipeline run, AWS CloudWatch monitors all the changes and sends email notifications to DevOps and the dev team through SNS notifications.

AWS CloudTrail tracks any critical changes such as pipeline update, deletion, and creation, and sends notifications to the DevOps team for audit purposes. Further, AWS Config tracks all the configuration changes.

For DevSecOps, CI/CD pipeline security is implemented using AWS IAM roles to restrict access to required resources only. Any pipeline data at rest and in transit is protected using encryption and SSL. You can use AWS Parameter Store to store sensitive information such as API tokens and passwords.

Aggregation of security findings in one place in Security Hub provides opportunities to automate the remediation. Based on the security finding, you can trigger a Lambda function to take the needed remediation action. For example, if someone accidentally opens an SSH port to everyone, it can automatically block the servers from internet traffic. Automation takes away the burden from the DevOps and security teams as they can now address the vulnerabilities from one tool instead of logging into multiple dashboards.

For any team, identifying the security threat during the early stages of application development can drastically reduce the overall cost of application changes. Doing it in an automated fashion can accelerate the delivery of these changes. A DevSecOps pipeline is critical to building a thriving application development environment.

DevOps combines culture, practices, and tools that combine application development with its operations. DevOps practice enables organizations to deliver new application features at speed. DevSecOps takes it a step further by integrating security into DevOps. With DevSecOps, you can provide secure and compliant application changes rapidly while running operations consistently with automation.

**Summary**

In this chapter, you have learned about the key components of a strong DevOps practice: CI, CD, and continuous monitoring and improvement. The agility of CI/CD can be achieved only by applying automation everywhere. To automate, you learned about IaC and configuration management. You also looked at various automation tools such as Chef, Puppet, and Ansible to automate configuration management.

As security is the priority, you learned about DevSecOps, which is DevOps in security. CD is one of the key aspects of DevOps. You learned about various deployment strategies, including rolling, blue-green, and red-black deployment. Testing is another aspect of ensuring the quality of your product. You learned about the concept of continuous testing in DevOps and how A/B testing can help improve the product by taking direct feedback from a customer in the live environment.

You have learned about the stages in a CI/CD pipeline. You have learned about the tools and services that you can use and best practices that you can follow for a robust CI/CD pipeline. You have learned how individual services work and discussed how to integrate services to build a sophisticated solution.

Until this point, you have learned about various aspects of solution architecture. As every organization has lots of data, they put great effort into getting insight into their data. In the next chapter, you will learn about collecting, processing, and consuming data to get a more in-depth insight.