Part 1

Getting started with Jenkins

This first part of this book takes you through the DevOps essential concepts. You'll learn about CI/CD practices and how they allow you to integrate small pieces of code at one time and ease technical debt. After that, I'll introduce the new approach of building CI/CD pipelines, pipeline as code, and how it can be implemented with Jenkins. Finally, I'll lay the groundwork for a well-designed CI/CD workflow by introducing the GitFlow branching model.

1 What's CI/CD?

This chapter covers

- The path organizations have taken to evolve from monolith to cloud-native applications
- The challenges of implementing CI/CD practices for cloud-native architectures
- An overview of continuous integration, deployment, and delivery
- How CI/CD tools like Jenkins can bring business value to organizations that undertake the journey of continuous everything

Software development and operations have experienced several paradigm shifts recently. These shifts have presented the industry with innovative approaches for building and deploying applications. More importantly, two significant paradigm shifts have consolidated capabilities for developing, deploying, and managing scalable applications: cloud-native architecture and DevOps.

Cloud-native architecture emerged with cloud adoption, with cloud providers like Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure taking ownership of the infrastructure. Open source tools like Kubernetes, Docker, and Istio offer horizontal scaling ability, letting developers build and run modern scalable applications without worrying about the underlying infrastructure. As a result, operational overhead is reduced, and the development velocity of applications is increased.

DevOps bridged the divide between developers and ops teams, and brought back harmony through collaboration, automated tools, and iterative and Agile development and deployment.

With these two significant, powerful approaches combined, organizations now have the capability to create scalable, robust, and reliable applications with a high level of collaboration and information sharing among small teams. However, to build, test, and safely deploy cloud-native applications, two essential DevOps practices must be implemented in a cloud-native manner: continuous integration (CI) and continuous deployment/delivery (CD).

The first part of this book takes you through the evolution of cloud-native applications. You'll learn about the main principles of CI/CD and how automation invented the way those principles are implemented through the *pipeline-as-code* approach. This first chapter lays the foundation. It introduces basic principles of DevOps and cloud-native approaches, in addition to selecting the tools for implementing CI/CD pipelines.

1.1 Going cloud native

Before exploring the essential characteristics of cloud-native applications and how CI/CD practices contribute to standardizing feedback loops for developers and enabling fast product iterations, we will cover the changes the software development model went through and the challenges associated with each model, starting with the monolithic approach.

1.1.1 Monolithic

In the past, organizations used to build their software in a *monolithic* way: all functionalities were packaged in a single artifact and deployed in a single server running one process. This architecture comes with many drawbacks and limitations:

- *Development velocity*—Adding new features on top of an existing application is next to impossible. Application modules are tightly coupled and, most of the time, not documented. As a result, adding new features is often slow, expensive, and requires extra synchronization when working with multiple developers within distributed teams on a large codebase. Moreover, the release cycle can take months, if not several years, because of the application's large codebase. This delay puts companies at risk of being surpassed by new competitors and ultimately undercuts the company's profits.

- *Maintainability*—Modules in a monolithic architecture are frequently tightly coupled, which makes them hard to maintain and test. Plus, upgrading to new technology is limited to the framework used to develop the application (no polyglot programming).
- *Scaling and resiliency*—Applications are designed with no scalability in mind, and the application may face downtime if traffic increases. The monolithic application works as a single unit and is developed in a single programming language using a single tech stack. As a result, to achieve partial horizontal scaling, the whole application needs to be scaled (inefficient usage of server resources).
- *Cost-effectiveness*—The application is expensive to maintain in the long run (for example, finding an experienced COBOL developer is time-consuming and expensive).

In the late 2000s, many web giants (including Facebook, Netflix, Twitter, and Amazon) came onto the tech scene with innovative ideas, aggressive strategies, and a "move fast" approach that led to the exponential growth of their platforms. These companies introduced a new architecture pattern that is known today as *microservices*. So, what exactly is microservices architecture?

1.1.2 Microservices

James Lewis and Martin Fowler defined microservices architecture as follows in 2014:

> *In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.*

This architecture uses the same technique of "divide and conquer" to tackle the complexity of an application. An application is split into smaller, independent, and composable services/fragments, each responsible for a specific functionality or task of the application (organized around business capabilities).

Those microservices communicate using an application programming interface (API), typically over HTTP or HTTP/2 (for example, gRPC, RESTful APIs, Google Protocol Buffers, or Apache Thrift), or through message brokers (such as Apache ActiveMQ or Kafka). Each microservice can be implemented in a different programming language running on a different OS platform.

In contrast to microservices, the monolithic architecture means the code's components are designed to work together as one cohesive unit, sharing the same server resources (memory, CPU, disk, and so forth). Figure 1.1 illustrates the differences between monolith and microservices architectures.

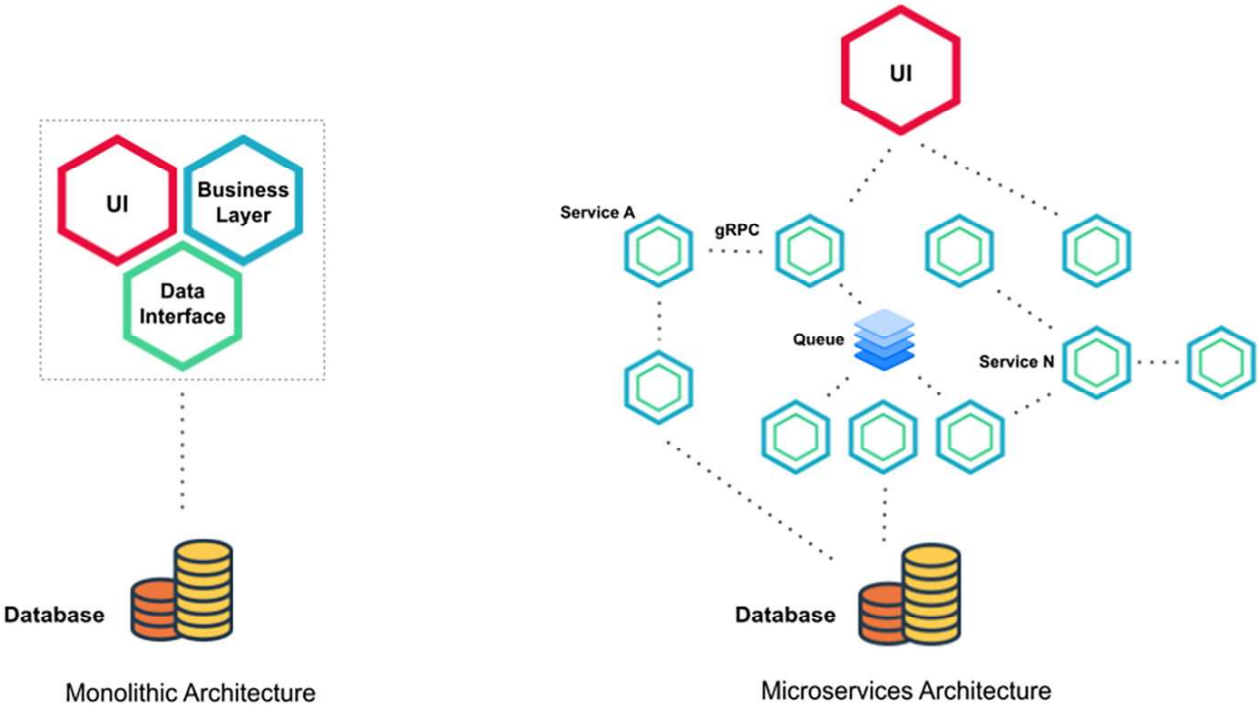

Figure 1.1 Comparing monolith and microservices architectures

Microservices architecture is an extension of *service-oriented architecture* (SOA). Both architectures rely on services as the main component, but they vary greatly in terms of service characteristics:

- *Granularity*—Service components within a microservices architecture are generally single-purpose services that do one thing. In SOA, service components can range in size, anywhere from small application services to very large enterprise services.
- *Sharing*—SOA enhances component sharing, whereas microservices architecture tries to minimize sharing through bounded context (loosely coupled services or modules) with minimal dependencies.
- *Communication*—Microservices rely on lightweight protocols such as HTTP/REST and simple messaging, while SOA architectures rely on enterprise service bus (ESB) for communication; early versions of SOA used object-oriented protocols to communicate with each other, such as Distributed Component Object Model (DCOM) and object request brokers (ORBs). Later versions used messaging services such as Java Message Service (JMS) or Advanced Message Queuing Protocol (AMQP).
- *Deployment*—SOA services are deployed to application servers (IBM WebSphere Application Server, WildFly, Apache Tomcat) and virtual machines. On the other hand, microservices are deployed in containers. This makes microservices more flexible and lighter than SOA.

NOTE For more details about microservices architecture, I recommend reading *Microservices in Action* by Morgan Bruce and Paulo A. Perreira (Manning, 2018). It covers what makes a microservice, how it can be composed by an individual or a dedicated team, the constant back-and-forth comparison between a monolithic application, and things to consider when deploying your microservices.

The advantages of microservices convinced some big enterprise players such as Amazon, Netflix, and Uber to adopt the methodology. Following their footsteps, other companies are working in the same direction: evolving from monolithic to flexible microservice-based architecture.

But what makes it so special? Compared to more monolithic design structures, microservices architecture comes with the following benefits:

- *Scalability*—Applications built as microservices can be broken into multiple components so that each component can be deployed and scaled independently without service interruption. Also, for stateless microservices, usage of Docker or Kubernetes can offer horizontal scaling within seconds.
- *Fault tolerance*—If one microservice fails, the others will continue to work because of loosely coupled components. A single microservice can be easily replaced by a new one without affecting the whole system. As a result, modernization in microservices architecture can be incremental, while modernization in monolithic architecture can cause service outages.
- *Development velocity*—Microservices can be written in different languages (polyglot programming) and use different databases or OS environments. If one microservice is, for example, CPU intensive, it could be implemented in highly productive languages such as Golang or C++, while other components could be implemented in lightweight programming languages such as JavaScript or Python. So companies can easily hire more developers and scale development. Also, because microservices are autonomous, developers have the freedom to independently develop and deploy services without bumping into each other's code (avoiding synchronization hell within the organization) and having to wait for one team to finish a chunk of work before starting theirs. As a result, team productivity increases, and vendor or technology stack lock-in reduces.
- *Continuous everything*—Microservices architecture combined with Agile software development enable continuous delivery. The software release cycle in microservice applications becomes much smaller, and many features can be released per day through CI/CD pipelines with open source CI tools like Jenkins.

To summarize, microservices make solving big problems easier, increase productivity, offer flexibility in choosing technologies, and are great for cross-functional teams. At the same time, running microservices in a distributed cloud environment can be a tough challenge for organizations. Here are some of the potential pain areas associated with microservices designs:

- *Complexity*—Increased complexity over a monolithic application due to the number of services involved. As a result, enormous effort, synchronization, and automation are required to handle interservice communication, monitoring, testing, and deployment.
- *Operational overhead*—Deploying a microservice-based application can be complex. It needs a lot of coordination among multiple services. Each service must be isolated with its own runtime environment and resources. Hence, traditional deployment solutions like virtualization can't be used and must be replaced with containerization solutions like Docker.
- *Synchronization*—Microservices require cultural changes in organizations seeking to adopt them. Having multiple development teams working on different services requires a huge effort to ensure that communication, coordination, and automated processes are in place. Cultures like Agile and DevOps practices are mandatory to take on microservice-based applications.

NOTE While Docker comes with no learning curve, it can quickly become a nightmare when handling deploying microservices among a cluster of machines or nodes.

Most of these drawbacks were addressed with the consumption of cloud computing services offered by AWS and with the rise of open source tools—particularly Kubernetes. It brought a completely new approach to managing infrastructure and enabled applications to be architected in a distributed manner. As a result, a new software architecture style arose in 2014: cloud-native applications.

1.1.3 Cloud native

The Cloud Native Computing Foundation (CNCF), a Linux Foundation project founded in 2015 to help advance container technology, defines *cloud native* as follows:

> *Cloud-native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach. These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.*

Cloud native is a paradigm for building applications as microservices and running them on containerized and dynamically orchestrated platforms that fully exploit the advantage of the cloud computing model. These applications are developed using the language and framework best suited for the functionality. They're designed as loosely coupled systems, optimized for cloud scale and performance, use managed services, and take advantage of continuous delivery to achieve reliability and faster time to market.

The overall objective is to improve the speed, scalability, and finally, profit margin. Figure 1.2 illustrates an example of a cloud-native application.

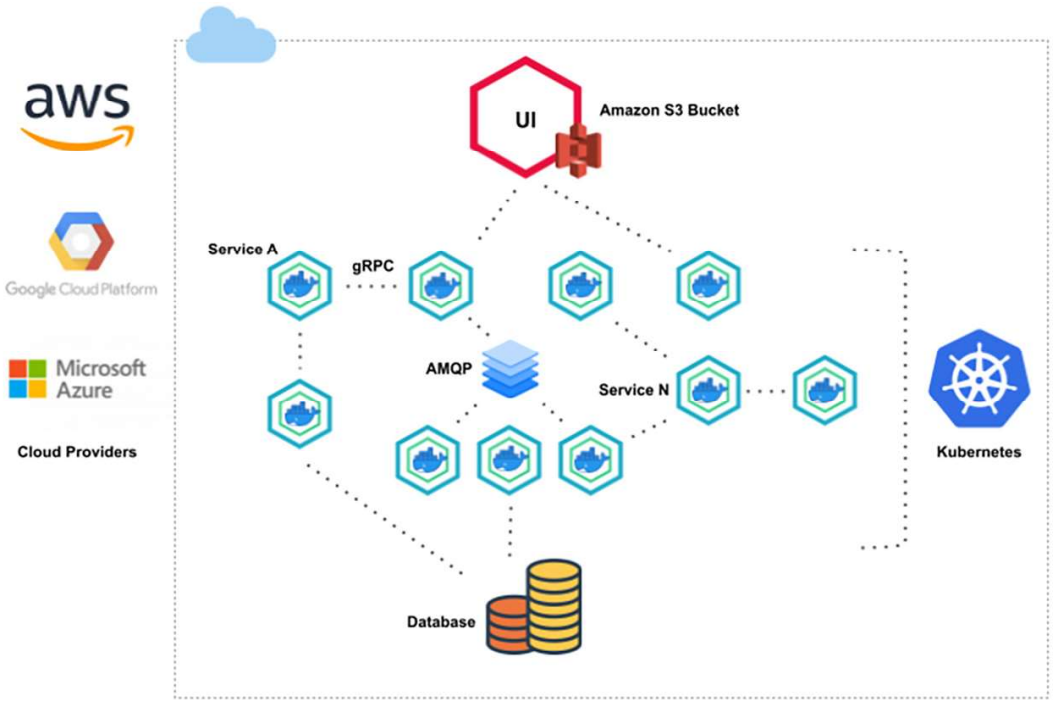

Figure 1.2 Overview of a cloud-native application

Cloud-native applications are packaged in lightweight containers and efficiently deployed as microservices. They use a lightweight API to expose their functionality, and binary and nonbinary protocols to communicate with each other internally. A step further, the applications are managed on elastic cloud infrastructure through Agile DevOps processes having continuous delivery workflows.

> **NOTE** Docker has become the standard for container technology. It has revolutionized the way we think about developing microservices, and enables us to easily deploy microservices locally, on premises, or in the cloud.

Kubernetes (https://kubernetes.io/) is one of the preferred platforms for running workloads that function as cloud-native applications. It's an open source container orchestration platform originally developed at Google. It ensures high-end automated deployment, scaling, and management of containerized applications. This new paradigm of building and deploying applications comes with many benefits:

- *No operational overhead*—Developers can focus on developing features and adding business value instead of dealing with infrastructure provisioning and management.
- *Security compliance*—Simplified security monitoring is required because the various parts of an application are isolated. A security problem could happen in one container without affecting other areas of the application.
- *Autoscaling*—Containers can be deployed into a fleet of servers in different availability zones or even multiple isolated data centers (regions). As a result,

cloud-native apps can take advantage of the elasticity of the cloud by scaling resources in or out during a use spike without the need to procure and provision physical servers. Also, by adopting cloud services, the business can go global in minutes with lower adaptation costs and increased revenue and without worrying about scalability.

- *Development speed*—The application architecture is easy to understand since each container represents a small piece of functionality, and is easy for developers to modify, so they can help a new team member become productive quickly. Also, adopting cloud-native technologies and practices enables companies to create software in-house, allowing business people to closely partner with IT people, keep up with competitors, and deliver better services to their customers.
- *Resiliency*—Cloud-native microservices allow for failure at a granular level. They do this by providing adequate isolation between each service and offer multiple design patterns that might improve the components' availability and resilience such as Circuit Breaker (https://martinfowler.com/bliki/CircuitBreaker.html), Throttling (www.redhat.com/architect/pros-and-cons-throttling), and Retry patterns. Companies like Netflix used it to develop a new approach called *chaos engineering* to build a resilient streaming platform.

Figure 1.3 shows the differences between monolithic, microservices, and cloud-native architectures.

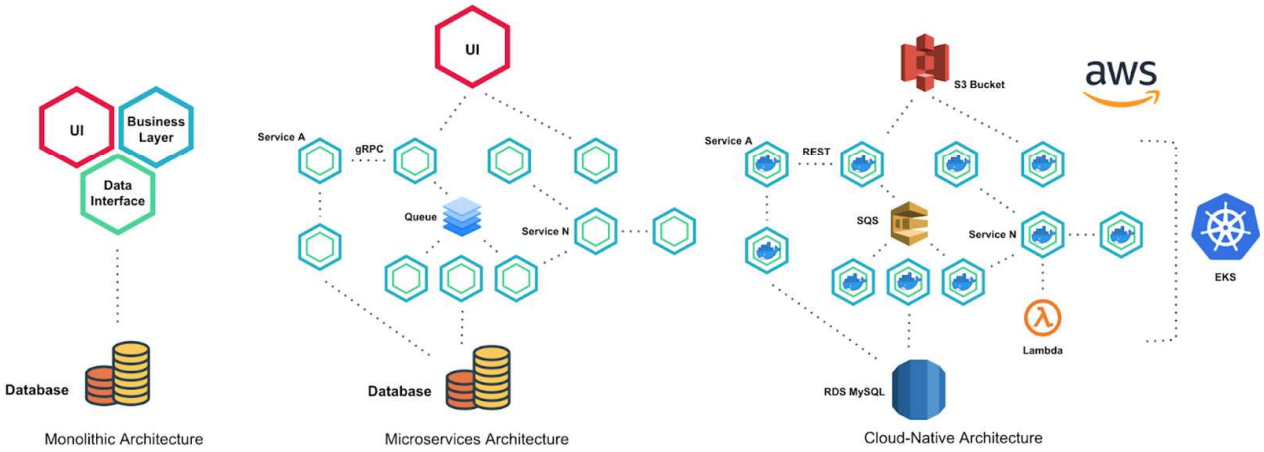

Figure 1.3 Monolith, microservices, and cloud-native architectures

To summarize, cloud-native architecture allows you to dynamically scale and support large numbers of users, events, and requests on distributed applications. A real-world example of the adoption of cloud-native architecture is the serverless model.

1.1.4 Serverless

The *serverless* computing model was kicked off with AWS Lambda in 2014. In this architecture, developers can write cost-efficient applications without provisioning or maintaining a complex infrastructure.

Cloud providers deploy customers' code to fully managed, ephemeral, time-boxed containers that live only during the invocation of the functions. Therefore, businesses can grow without customers having to worry about horizontal scaling or maintaining complex infrastructure.

> **NOTE** Serverless doesn't mean "no ops." You're just outsourcing sysadmin with serverless services. You will still deal with monitoring, deployment, and security.

An application built based on serverless architecture may end up looking like figure 1.4.

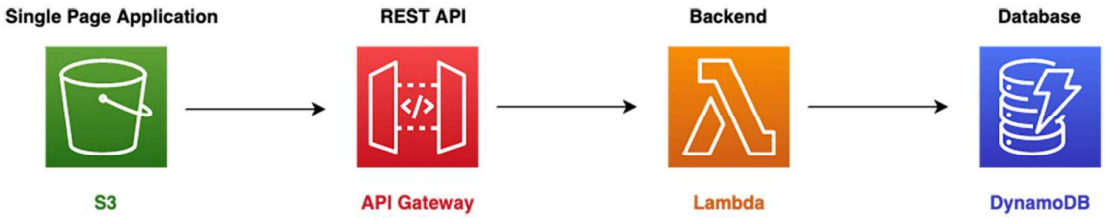

Figure 1.4 An example of a serverless application

Instead of maintaining a dedicated container or instance to host your static web application, you can combine an Amazon Simple Storage Service (S3) bucket to benefit from scalability at a cheaper cost. The HTTP requests coming from the website go through Amazon API Gateway HTTP endpoints that trigger the right AWS Lambda function to handle the application logic and persist data to a fully managed database service such as DynamoDB. For particular use cases, going serverless can make sense for several reasons:

- *Less operational overhead*—The infrastructure is managed by the cloud provider, and this reduces the overhead and increases developer velocity. OS updates are taken care of, and patching is done by the function-as-a-service (FaaS) provider. This results in decreased time to market and faster software releases and eliminates the need for a system administrator.
- *Horizontal autoscaling*—Function becomes the unit of scale that leads to small, loosely coupled, stateless components that, in the long run, lead to scalable applications. Plus, the scaling mechanism is shifted to the cloud provider, which decides how to use its infrastructure effectively to serve the client's requests.
- *Cost optimization*—You pay for only the compute time and resources that you consume. As a result, you don't pay for idle resources, which significantly reduces infrastructure costs.
- *Polyglot*—Another benefit is the ability to choose a different language runtime depending on the use case. One part of the application can be written in Java, while another in Python; it doesn't really matter as long as the job gets done.

NOTE A big concern while going serverless is vendor lock-in. Although you should favor development speed and efficiency above all, it's important to choose a vendor based on your use case.

Cloud-native architectures, in general, are gaining massive adoption, but the learning curve for many teams is steep. Plus, the shift to cloud-native architecture can be a double-edged sword for many organizations, and one of the challenges when moving to a fully cloud-native approach can be CI/CD.

But what do these practices mean? And how can they be applied when you're building cloud-native applications?

1.2 Defining continuous integration

Continuous integration (CI) is the practice of having a shared and centralized code repository, and directing all changes and features through a complex pipeline before integrating them into the central repository (such as GitHub, Bitbucket, or GitLab). A classic CI pipeline is as follows:

1. Triggers a build whenever a code commit occurs
2. Runs the unit tests and all pre-integration tests (quality and security tests)
3. Builds the artifact (for example, Docker image, zip file, machine learning training model)
4. Runs acceptance tests and pushes the result to an artifact-management repository (such as a Docker Registry, Amazon S3 bucket, Sonatype's Nexus, or JFrog Artifactory)

Figure 1.5 shows an example of a CI pipeline for a containerized application.

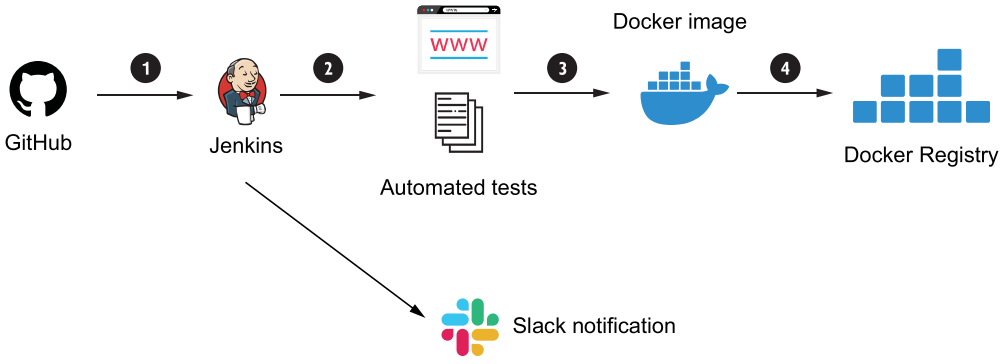

Figure 1.5 Basic CI workflow for cloud-native applications

Basically, CI automatically monitors the commits that each developer makes and launches automated tests. Automated testing is an integral part of CI/CD pipelines. Without automated tests, CI/CD pipelines will lack quality checks, which are important in order for the application to be released.

You can implement various types of testing to ensure that your software meets all the initial requirements. Here are the most famous ones:

- *Unit tests*—These test each piece of the source code. They consist of testing individual functions and methods. You could also output your test coverage and validate that you're meeting your code coverage requirements.
- *Quality tests*—Check that the code is well formatted, follows best practices, and has no serious coding errors. This is also called *static code analysis*, as it helps to produce high-quality code by looking for patterns in code that might generate bugs.
- *Security tests*—Inspect source code to uncover common security vulnerabilities and common security flaws (for example, leaked usernames and passwords).
- *UI tests*—Simulate user behavior through the system to ensure that the application works correctly in all supported browsers (including Google Chrome, Mozilla Firefox, and Microsoft Internet Explorer) and platforms (such as Windows, Linux, and macOS) and that it delivers the functionality promised in user stories.
- *Integration tests*—Check that services or components used by the application work well together and no defects exist. For example, an integration test might test an application's interaction with the database.

Manually executing all these tests can be time-consuming and counterproductive. Therefore, you should always use a testing framework that suits your application requirements to perform those tests on a scale in a repeatable and reliable way.

NOTE Chapter 8 covers how to run automated tests with Jenkins and Headless Chrome, as well as how to integrate SonarQube for code analysis.

Once tests are successful, the application will be compiled and packaged, and a releasable artifact will be generated and versioned in a remote repository.

1.3 Defining continuous deployment

Continuous deployment (CD) is an extension of continuous integration. Every change that passes all stages of your continuous integration pipeline is released automatically to your staging/preproduction environment.

In such a process, there's no need to decide what will be deployed and when. The pipeline will automatically deploy whatever build components/packages successfully exit the pipeline. Figure 1.6 illustrates a typical CI/CD pipeline for microservices running in Kubernetes.

This CI workflow has four steps, and the CD pipeline is the deployment to Kubernetes (step 5). However, a pure continuous deployment approach is not always appropriate for everyone.

For example, many clients would not appreciate new versions falling into their laps several times a week, and prefer a more predictable and transparent release cycle.

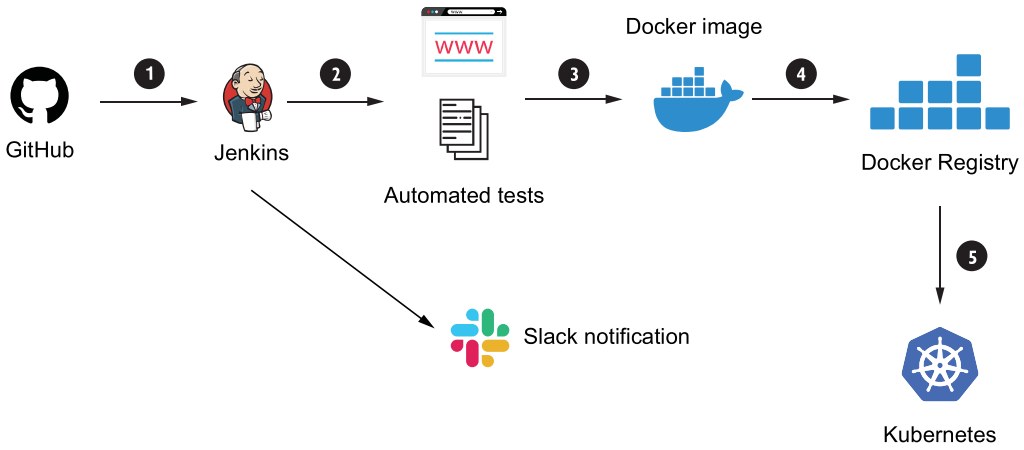

Figure 1.6 Basic CI/CD workflow for cloud-native applications

Commercial and marketing considerations might also play a role in when a new release should actually be deployed.

While continuous deployment may not be right for every company, continuous delivery is an absolute requirement of DevOps practices. Only when you continuously deliver your code can you have true confidence that your changes will be serving value to your customers within minutes of pushing the "go" button, and that you can actually push that button any time the business is ready for it.

1.4 Defining continuous delivery

Continuous delivery (CD) is similar to continuous deployment but requires human intervention or a business decision before deploying the release to production. Figure 1.7 shows how the CI/CD practices relate to each other.

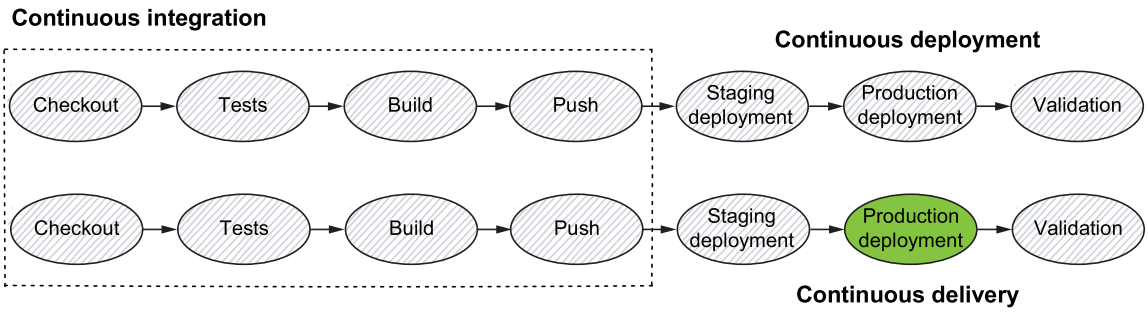

Figure 1.7 The continuous deployment maturity model

NOTE A Monitor and Optimize stage can occur in a sophisticated CI/CD workflow. This step consists of collecting and analyzing metrics and feedback to eliminate risks and waste and to optimize the release time.

1.5 Embracing CI/CD practices

CI/CD and continuous delivery can bring more agility to cloud-native applications through daily builds, which leads to the following:

- Detecting anomalies at an earlier stage (reducing the risk) and minimizing technical debt through unit and functional tests. According to Atlassian (www.atlassian.com/software-development/practices), 75% of development teams face issues with bugs, defects, or delays when it's time to release.
- Building features your users actually want. This often results in better user interaction and quicker feedback regarding released features, which can help the product team focus on the most demanded features and build a high-quality product.
- Having a production-ready package available. This is an excellent way to accelerate the time to market.
- Increasing product quality and reliability through quality and stress tests, and tracking with better visibility into project status and health.
- Driving innovation from feedback while building high-quality products through each iteration.

However, the journey from a manual to a highly automated deployment process can take several months. Therefore, companies need to be iterative in adopting CI/CD, as illustrated in figure 1.8.

You should always prioritize the steps in CI/CD. First and foremost, automate the process for compiling the source code. Ideally, you will develop new features and fix multiple bugs per day. Manually, this process takes a few minutes to a couple of hours. Also, you should prioritize functional testing before UI testing, as it often changes and

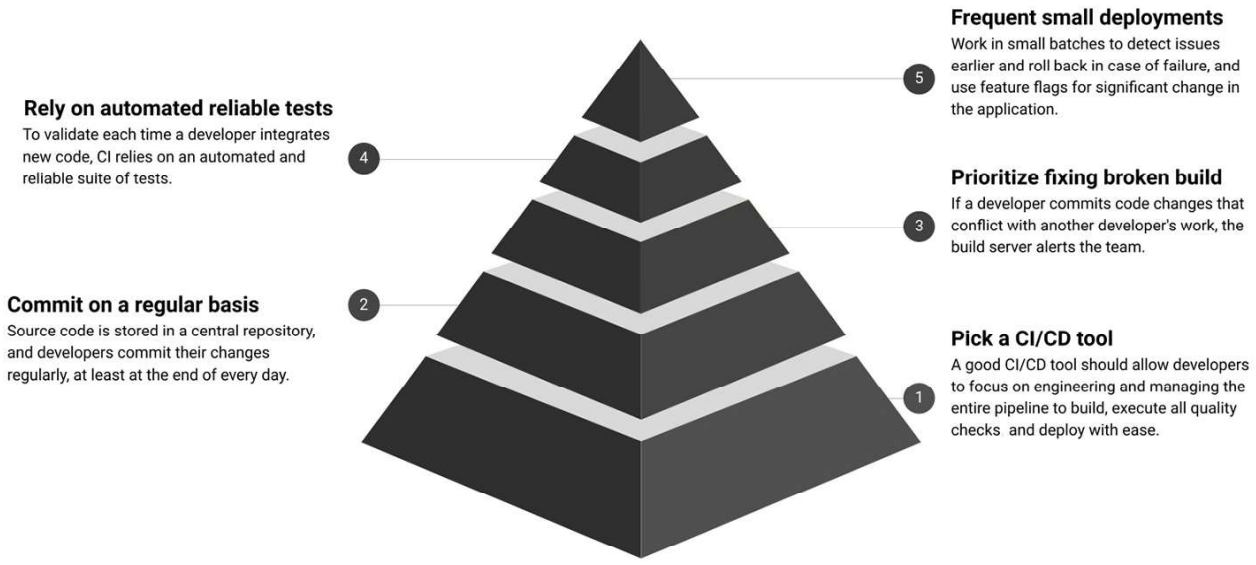

Figure 1.8 Introducing CI/CD to an organization

thus requires frequent pipeline changes. So make sure to break your CI/CD steps into smaller segments and automate in patches to make the best use of your resources.

Another concern is that the complexity of CI/CD will be increasing, from handling singular applications to dozens of microservices (multiple pipelines). Therefore, adapting your CI/CD tools and processes is mandatory to keep pace.

Moreover, you need to have a clear road map of your product with a proven track record of development success. Your end customers should be able to consume constant product changes. Therefore, using CI/CD requires a high degree of discipline, dedication to quality, and a learning curve (new skill sets). If you can't handle that, stop thinking about CI/CD immediately.

As a result, moving to CI/CD should not be an isolated decision, made alone by the DevOps team. A successful rollout of CI/CD must be a decision for your whole organization and should be made only when your entire organization agrees to it.

Although you need to keep some concerns in mind, the benefits of CI/CD almost always outweigh the challenges. To realize the full promise of cloud-native applications, you must implement CI/CD practices that are best suited to your unique business goals.

In this book, we will go through some real-world use cases for building CI/CD pipelines for most adopted cloud-native architectures, such as Dockerized microservices with both Docker Swarm and Kubernetes, as well as Lambda-based serverless applications. We will also cover how to manage and scale a CI tool with less maintenance hassle to help you increase deployment speed. But first, what makes a modern CI tool, and which one are we going to use?

> **NOTE** While monoliths may not be trendy, many companies still have monolith flagship products and can still benefit tremendously from a well-architected CI/CD solution. So most of the examples in the book can also be applied to modernizing monolithic applications.

1.6 Using essential CI/CD tools

A lot of excellent CI tools are out there. Some have been here for a long time, and others are relatively new. It's a bit redundant to say that a modern CI tool must be fast, user-friendly, and flexible, since those are the features we already expect out of the box. CI tools can be divided into the following three main categories:

- Cloud-managed solutions like AWS CodePipeline (https://aws.amazon.com/codepipeline/), Google Cloud Build (https://cloud.google.com/build), and Microsoft Azure Pipelines (https://azure.microsoft.com/services/devops/pipelines/).
- Open source solutions such as Jenkins (www.jenkins.io), Spinnaker (https://spinnaker.io/), or GoCD (www.gocd.org).
- Software-as-a-service (SaaS) solutions like Travis CI (https://travis-ci.org/), CircleCI (https://circleci.com/), and TeamCity (www.jetbrains.com/teamcity/).

1.6.1 Choosing a CI/CD tool

Figure 1.9 shows the most popular CI/CD tools on the market today. These tools are the mature ones, with the essential capabilities for your project.

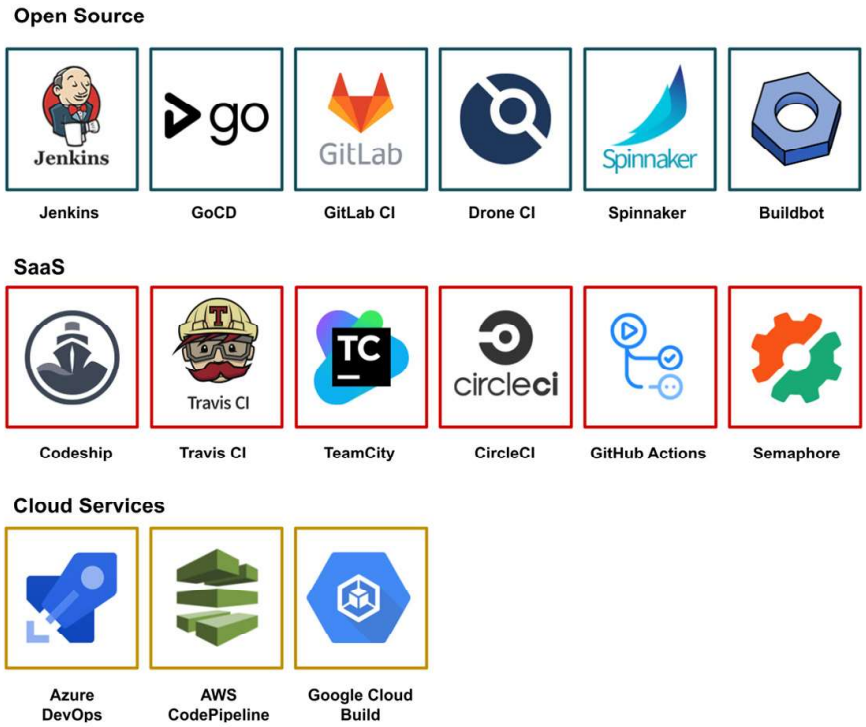

Figure 1.9 Top CI/CD tools in 2021

Plenty of excellent CI tools are available, so you need to pick the best one based on the following factors:

- *Team experience and skills*—While many tools use configuration YAML files to declare the CI/CD pipeline, they might require some sysadmin skills to set up and provision the needed infrastructure to run the CI/CD platform. Also, maintaining the underlying infrastructure might cause a lot of headaches and become a bottleneck for your company's growth once your project codebase becomes bigger (scaling capabilities), as you need to maintain distributed CI/CD complex pipelines across multiple nodes or servers.
- *Target platform*—Consider the operating system your application or project is running on (some CI tools don't support macOS and ARM architecture), and the use of a self-hosted infrastructure or a cloud provider.
- *Programming language and architecture*—Most CI tools support the top cutting-edge languages including Java, Ruby, Python, PHP, and JavaScript. However, some tools like TeamCity offer better integration and support for Java and .NET

projects. Similarly, Bamboo, as a creation of Atlassian, has native support for Jira and Bitbucket. Additionally, the deployment solution can be a factor in choosing the right CI tool for your project. Tools like Drone (www.drone.io) and GitLab CI (https://docs.gitlab.com/ee/ci/) offer native Docker support with an integrated Docker registry.

1.6.2 Introducing Jenkins

Although no single tool can satisfy the needs of every project, in this book, we will rely heavily on *Jenkins*. It's considered one of the most popular CI tools on the market today, with over one million users. It was written in Java, making it a cross-platform (Windows, Linux, and macOS) continuous integration tool.

Originally a part of the Hudson project, the community and codebase split following trademark conflicts with Oracle after it acquired Sun Microsystems. Hudson was originally released in 2005, while the first release as Jenkins was made in 2011.

> **NOTE** Hosted SaaS platforms can be beneficial if you're willing to pay a bit of extra money for someone else to maintain and update the solution. Businesses tend to choose this option when they need a UI superior to what Jenkins offers and when they lack infrastructure skills. But a major benefit of self-hosting solutions like Jenkins is that you have more control and flexibility over your own data security and job pipelines.

A rich set of plugins enables Jenkins to support any type of language or technology such as Docker, Maven, Git, Mercurial, and AWS. Being an open source project makes it customizable and easy for developers to extend by creating custom plugins. Here are some of Jenkins key features:

- Extensible with a huge community-contributed plugin resource (more than 1,400 plugins).
- A free and open source tool as well as a paid enterprise edition offered by CloudBees (www.cloudbees.com/jenkins) with speedy customer support.
- Has an active community that helps developers reduce the time to build a working CI/CD workflow.
- Can be deployed on premises or in the cloud with an easy configuration through the user interface or the command line.
- Supports distributed builds with master-worker architecture with a built-in parallelism mechanism.
- A powerful and flexible tool with complete control over workflow that can serve every CI/CD need.
- Works on many platforms and has the support for a wide variety of tools and frameworks.
- Supports containers as build agents for teams planning to use Docker.
- Seamless integration with GitHub, GitLab, Bitbucket, and most of the source code management (SCM) systems and Apache Subversion (SVN).

- Flexible user management, user roles assignment, sorting users into groups, different ways of user authentication (including LDAP, GitHub OAuth, and Active Directory).
- The CI process can be defined using the Groovy language in files within the repository itself or through text fields in the Jenkins web UI, thanks to the Jenkins pipeline workflow.

NOTE If you like to test a small application for one particular platform, you won't need the complexity of running a Jenkins server.

Another key feature of Jenkins is *pipeline as code*. We're going to use this approach to create Jenkins jobs. The cool part of using this approach is that our entire Jenkins jobs configuration can be created, updated, and version-controlled along with the rest of the application source code.

It is helpful to note that Jenkins must be hosted on a server, so it often needs the attention of someone with infrastructure skills. You can't just set it up and then expect it to run itself; the system requires frequent updates and maintenance. The main barrier to entry for most teams is the initial setup, procrastination, or failed previous attempts to set it up. People tend to know it's good, but many teams neglect it for more urgent coding work. Perhaps someone on your team tried to deploy Jenkins at some point but did not successfully maintain it. Maybe the wasted effort gave your boss a bad impression about it.

The reasons people do not implement Jenkins are usually very practical. That's why, throughout this book, we will be using the magical power of infrastructure as code with open source tools like Terraform and Packer to set up our entire CI infrastructure out of thin air on most popular public cloud providers such as AWS, GCP, and Microsoft Azure.

Another problem we will tackle in this book is how to write tests. Writing tests is something most developers want to do, but often don't find the time to do. Understandably, coding the actual application is usually a higher priority for the business. Also, tests break, meaning when the functionality under test changes, it needs to be updated. If functionality is not updated, it stops delivering value. We will cover how to run various types of tests within CI/CD pipelines and how to integrate external code analysis tools.

To sum up, implementing CI/CD for cloud-native architecture requires a cultural and mindset shift, especially from management. Managers have to allow time for this "unproductive stuff" to be done.

Still, the brief sacrifice of time translates into long-term benefits for the whole company. With Jenkins, your code becomes easier to maintain, and fewer bugs sneak into production. Your team becomes more integrated, and builds take less time. Your business can ship faster and keep up with the changing needs of your customers (by shipping code faster, organizations can quickly respond to changes and keep products on the market).

CI/CD is not an expense but an investment. And the return on investment (ROI) for implementation can be counted in time saved, errors avoided, and higher-quality products delivered more easily to your clients.

Summary

- Cloud-native architectures are changing the landscape, forcing organizations to think about new models and new delivery methods.
- Continuous integration, delivery, and deployment are practices designed to help increase the velocity of development and the release of well-tested, usable products.
- Choosing the right CI/CD tool is critical to the long-term success of cloud-native applications and should be based on platform complexity, integration, learning curve, pricing, and work-time efficiency.
- Jenkins can leverage the team's current workflow to best exploit the automation features and create a solid CI/CD pipeline.