**13**

**Implementing DevSecOps**

The typical reason why most enterprises adopt the cloud is to accelerate application development. Applications are constantly evaluated and changed to add new features. Since everything is codified in the cloud, these new features need to be tested on the infrastructure of the target cloud. The final step in the development process is the actual deployment of applications to the cloud and the handover to operations so that developers have their hands free to develop new features again based on business requirements and feedback from the customer and operations.

To speed up this process, organizations work in DevOps cycles, with continuous development and the possibility to test, debug, and deploy code multiple times per week or even per day so that these applications are constantly improved. Consistency is crucial: the source code needs to be under strict version control. That is what **CI/CD** pipelines are for: **continuous integration and continuous delivery and deployment**. But we also need to make sure that code, pipelines, applications, and infrastructure remain secure. Hence, we must embed security into DevOps and aim for DevSecOps.

We will study the principles of DevSecOps, how CI/CD pipelines work, and how they are designed so that they fit multi-cloud environments and comply with security policies.

In this chapter, we’re going to cover the following topics:

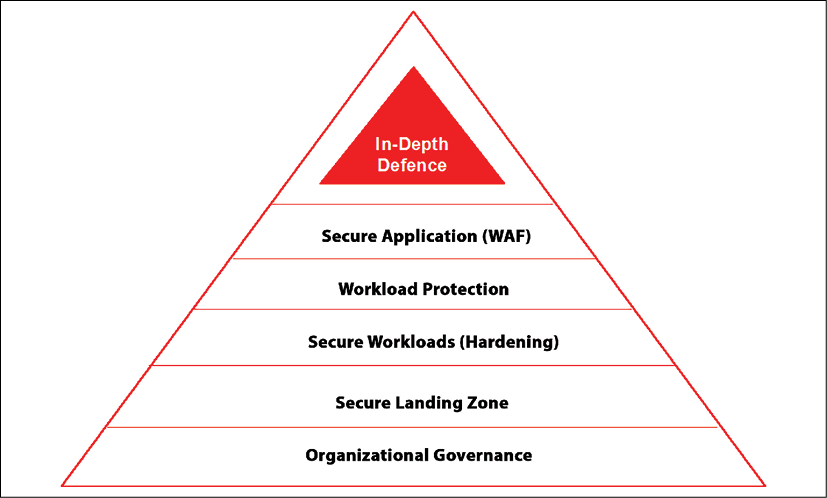
* Understanding the need for DevSecOps
* Starting with implementing a DevSecOps culture
* Setting up CI/CD
* Working with CI/CD in multi-cloud
* Exploring tools for CI/CD
* Following the principles of Security by Design
* Securing development and operations using automation

**Understanding the need for DevSecOps**

Before we dive into the layers of DevSecOps, it’s good to understand why DevSecOps is important in multi-cloud. First, we must understand the layers of securing the cloud. There are four layers to be considered:

1. Organizational level, or the overarching governance
2. Enterprise level, ensuring the security across accounts, auditing centralized compliance through monitoring and logging, and promoting automation
3. Subscription level, using **Role-Based Access Control** (**RBAC**), threat detection, and in-depth defense
4. Solution level, using CI/CD with validated templates, blueprints, and images

We must define security at all levels. The following diagram shows all levels of defense in the cloud.



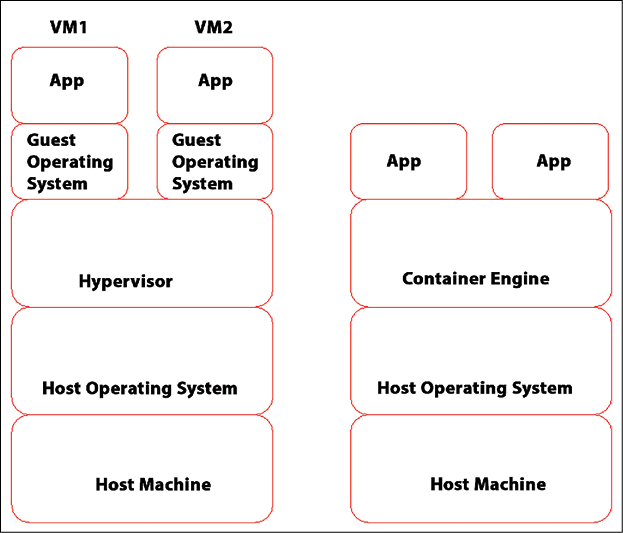
*Figure 13.1: Levels of security in application stacks*

The top of the stack is formed by the application payloads. In multi-cloud, enterprises will likely use containers and CI/CD pipelines. With multi-cloud, enterprises acquire the capabilities to:

* Manage and provision workloads across multiple cloud platforms in various regions
* Scale workloads for the best performance, leveraging the benefits of each specific platform
* Create high-availability solutions using the resilience of multiple cloud platforms
* Simultaneously operate applications across multiple clouds

How do we do that in multi-cloud? The best answer to this is: with containers that we can deploy on these various cloud platforms. Hence, we first must understand how containers and container orchestration work in multi-cloud. Containers allow the software to run reliably, regardless of the environment or cloud. Developers only have to take care of what’s in the container, while the operations of the containers themselves are typically fully automated using container orchestration, which enables, for instance, unlimited scaling without manual intervention.

Containers work differently from VMs. With VMs, we have a hypervisor telling a server that it’s multiple servers, able to run multiple applications on one of the virtual servers. However, each virtual server requires its own operating system, making a VM rather heavy. Containers don’t require their own operating system; they simply use the operating system that’s running on the host machine. This makes containers very light and easily transportable. The difference between containers and VMs is explained in *Figure 13.2*:



*Figure 13.2: Difference between Virtual Machines (left) and containers (right)*

So, containers can run on any platform: they are portable, meaning that a container can easily be deployed to different servers on different platforms. That is true, but we need to prepare our clouds to run containers. In practice, it means that we need a runtime environment for our containers, the container engine. The industry standard to orchestrate containers in any cloud is Kubernetes, originally invented by Google.

**Kubernetes** takes care of the deployment and management of containers. It handles all the heavy operational tasks, such as allocating networking, compute, and storage to the workloads that are hosted in the containers. It also enables automated scaling: if workloads need more capacity, Kubernetes will scale out the container infrastructure, allowing for more containers to run. If usage decreases, Kubernetes will scale down again. The best part: it can do that on all clouds. All cloud providers offer Kubernetes services:

* **Azure Kubernetes Services** (**AKS**)
* **Elastic Kubernetes Services** (**EKS**) in AWS
* **Google Kubernetes Engine** (**GKE**) in GCP
* **Alibaba Cloud Container Service for Kubernetes** (**ACK**)
* **Container Engine for Kubernetes in OCI** (**OKE**)

Now we have a runtime environment, and we have containers. Both must be secured. First, we must understand that Kubernetes is software that runs on servers, forming a Kubernetes cluster with multiple virtual machines or nodes. As with any server, we need to protect the servers and the software that runs on top of it, preventing unauthorized access, malicious injections, or other breaches. This is a shared responsibility: PaaS Kubernetes cloud offerings protect and secure the nodes in the cluster.

Our first concern is related to the hardening of the cluster. In the design and implementation, we have to think about:

* EnablingRBAC
* Enabling authentication
* Protection with firewalls
* The isolation of Kubernetes nodes
* Monitoring network traffic and audit logging

Then, we must secure our containers. For this, we can refer to the **National Institute of Standards and Technology**(**NIST**) Application Container Security Guide (<https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf>).

To secure containers, we must consider hardening the host operating system but also make sure that we use secure images to build our containers. Especially as the image is crucial in securing containers. The image is a file with executable code and includes everything that is required to run a container: the container engine (for instance, Docker), system libraries, and configuration settings. Components in images are typically reusable to prevent that we have to build new images over and over again.

These components and images are stored in controlled container registries. These registries are crucial in every enterprise environment using containers so that the CI/CD pipelines only use approved images from these registries. Containers are immutable by their nature. That’s another reason why it’s crucial to protect the source images. With that, we are stepping into DevSecOps.

In this chapter, we will look at DevSecOps in three layers. The first layer is about creating awareness in the teams. Every member in a DevOps team must be aware of the fact that every decision they make will have consequences, financially and in terms of security. Developers must learn the basics of writing secure code and understand the implications of writing code that leaves “holes” and introduces vulnerabilities. Engineers must be proficient in managing the environments in which developers work, keeping this secure. Think of the landing zones and the CI/CD pipelines that must be secured. Hence, education is important at all levels, in all roles.

The second layer is about applying the principles of Security by Design, making security an integrated part of everything that developers and engineers build and manage in the cloud. A lot of tasks can, however, be automated, limiting the fault tolerance of manual work. As such, the third layer is securing by automation.

In the next sections, we will explore these layers one by one and discover why these are important in multi-cloud, starting with education and creating awareness. We will then also discover how we can set up CI/CD pipelines and what tools we can use in the various clouds, but ultimately, we want a process and a pipeline that is able to address multi-cloud.

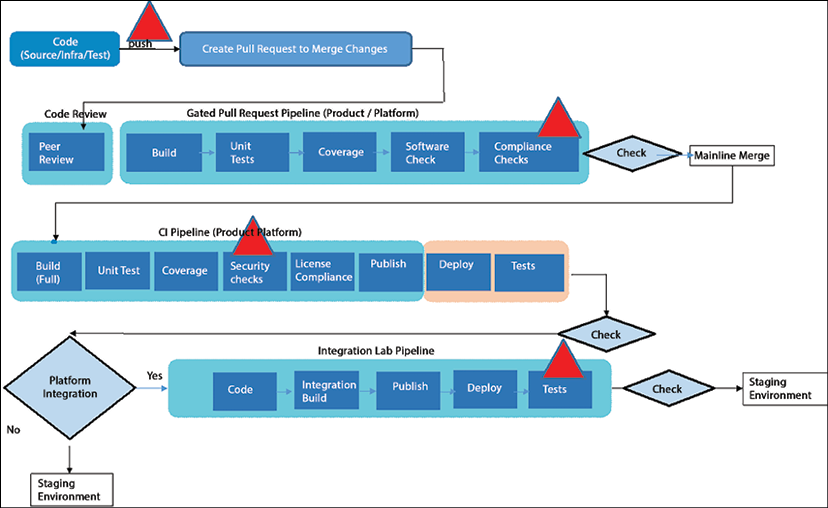
**Starting with implementing a DevSecOps culture**

In the previous section, we already mentioned the NIST guide to secure containers. That guide starts with something non-technical: a mindset. The first advice NIST gives is:

*Tailor the organization’s operational culture and technical processes to support the new way of developing, running, and supporting applications made possible by containers.*

Why would we need to change the culture in the way we do IT? Because with cloud, cloud-native, and containers, the way of doing software development changes drastically. Developers and operations might be less concerned with traditional IT processes such as patching and upgrading systems. We want to integrate security in to the builds of the applications, including the way applications utilize the underlying infrastructure without having to worry about the physical infrastructure or even the virtual machines. Let software take care of it, but then the software must be programmed, configured, and maintained in the most secure way.

There’s one question that we must ask upfront: where does security start? The answer to that question is: at the very beginning, also in CI/CD pipelines. The following diagram shows a high-level representation of a pipeline. The triangles here indicate the security practices that must be followed as a minimum:



*Figure 13.3: CI/CD pipeline with security indication*

This is a good point to explain the principle of shift-left: the principle where we shift testing as early as possible in the lifecycle of application development, so indeed, already at the very beginning of the pipeline where the code is pulled or written. It’s a crucial principle in DevSecOps. Another principle is the close cooperation between developers and operations, including security definition and management.

There are a lot of views on DevOps, but this book sticks to the definition and principles as defined by the **DevOps Agile Skills Association** (**DASA**). They define a DevOps framework based on six principles:

* **Customer-centric action**: Develop an application with the customer in mind—what do they need, and what does the customer expect in terms of functionality? This is also the goal of another concept, domain-driven design, which contains good practices for designing.
* **Create with the end in mind**: How will the application look when it’s completely finished?
* **End-to-end responsibility**: Teams need to be motivated and enabled to take responsibility from the start to the finish of the application life cycle. This results in mottos such as “you build it, you run it, and you break it, you fix it”. One more to add is “you destroy it, you rebuild it better”.
* **Cross-functional autonomous teams**: Teams need to be able and allowed to make decisions themselves in the development process.
* **Continuous improvement**: This must be the goal—to constantly improve the application.
* **Automate as much as possible**: The only way to really gain speed in delivery and deployment is by automating as much as possible. Automation also limits the occurrence of failures, such as misconfigurations.

It’s a new way of thinking about developing and operating IT systems based on the idea of a feedback loop. Since cloud platforms are code-based, engineers can apply changes to systems relatively easily. Systems are code, and code can be changed as long as changes are applied in a structured and highly controlled way. That’s the purpose of CI/CD pipelines.

What do we need to do to build a DevSecOps culture? The answer: education and experimenting. Organizations should support experimenting in isolated environments so that developers and operations can learn. Instructor-led and self-paced training, peer reviews, and support in architecture, development, and security will accelerate the adoption of the DevSecOps mindset.

The following step is to integrate this mindset and the forthcoming principles in the setup of the CI/CD pipelines. We will discuss that in the next section.

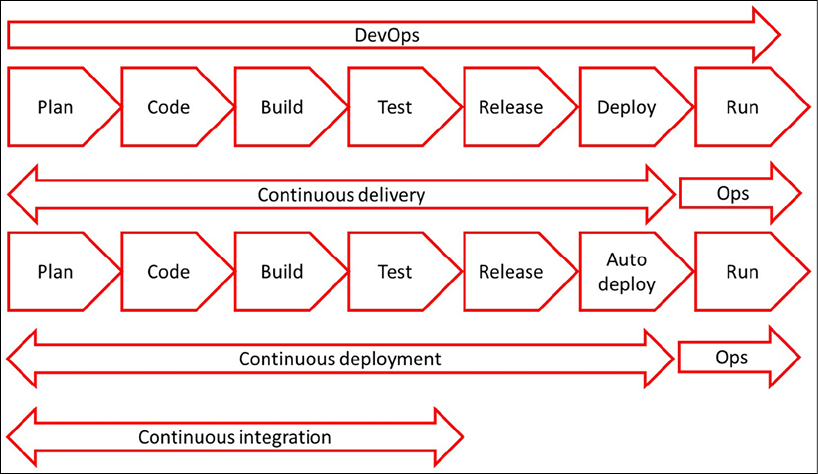
**Setting up CI/CD**

Before we start building pipelines, it’s good to have a definition of CI and CD.

**Continuous Integration** (**CI**) is built on the principle of a shared repository, where code is frequently updated and shared across teams that work in the cloud environments. CI allows developers to work together on the same code at the same time. The changes in the code are directly integrated and ready to be fully tested in different test environments.

**Continuous Delivery and Deployment** (**CD**) focuses on the automated transfer of software to test environments. The ultimate goal of CD is to bring software to production in a fully automated way. Various tests are performed automatically. After deployment, developers immediately receive feedback on the functionality of the code.

**CI/CD** enables the DevOps cycle. Combined with CI/CD, all responsibilities from planning to management lie with the team, and changes can reach the customer much faster through an automated and robust development process. The following diagram shows the DevOps cycle with CI/CD:



*Figure 13.4: DevOps cycle with CI/CD*

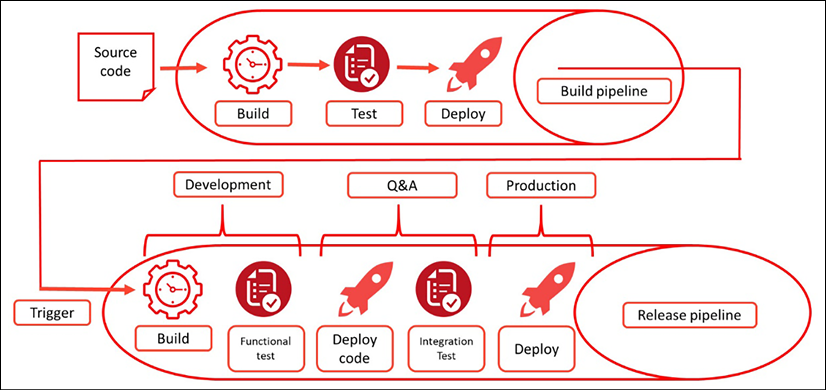
CI/CD is widely adopted by enterprises, but a lot of projects fail, typically because of a lack of consistency. With CI, development teams can change code as often as they want, leading to the continuous improvement of systems. Enterprises will have multiple development teams working in multi-cloud environments, which makes it necessary to have one way of working. Fully automated processes in CI/CD pipelines can help keep environments consistent. That’s why CI/CD and DevOps are not about tools in the first place. They’re about culture and sticking to processes.

To get to a successful implementation of DevOps, an organization is advised to follow these steps:

1. Implementing an effective CI/CD pipeline begins with all stakeholders implementing DevOps processes. One of the key principles in DevOps is autonomous teams that take end-to-end responsibility. It’s imperative that the teams are given the authority to make decisions and act on them. Typically, DevOps teams are agile, working in short sprints of 2 to a maximum of 3 weeks. If that time is wasted on getting approval for every single detail in the development process, the team will never get to finish anything in time.
2. Choose the CI/CD system. The repository is really important, and it’s one of the first steps. Then you can decide on the CI/CD system: one that includes a repository such as GitHub, Azure DevOps, or GitLab. There are a lot of tools on the market that facilitate CI/CD. GitHub Actions and Jenkins are popular, but a lot of companies that work in Azure choose to work in Azure DevOps. Involve the people who have to work daily with the system and enable them to take a test drive. Then, make a decision and ensure all teams work in that system. Again, it’s about consistency.
3. It’s advised to do a proof of concept. An important element of CI/CD is the automation of testing, so the first step is to create an automated process pipeline. Enterprises often already have quality and test plans, possibly laid down in a **Generic Test Agreement** (**GTA**). This describes what and how tests must be executed before systems are pushed to production. This is a good starting point, but in DevOps, organizations work with a **Definition of Done** (**DoD**).
   * The DoD describes the conditions and the acceptance criteria a system must meet before it’s deployed to production. The DoD is the standard of quality for the end product, the application, or the IT system that needs to be delivered.
4. Automating as much as possible is one of the principles of DevOps. This means that enterprises will have to adopt working in code, including IaC. In CI/CD, teams work from one repository, and this means that the application code and the infrastructure code are in the same repository so that all teams can access it whenever they need to.

In DevOps, teams work with user stories. An example of a user story is: *as a responsible business owner for an online retail store, I want to have multiple payment methods so that more customers can buy our products online*. This sets requirements for the development of applications and systems. The DoD is met when the user story is fulfilled, meaning that unit testing is done, the code has been reviewed, the acceptance criteria are signed off, and all functional and technical tests have been passed.

The following diagram shows the concept of implementing a build and release pipeline with various test stages. The code is developed in the build pipeline and then sent to a release pipeline, where the code is configured and released for production. During the release stages, the full build is tested in a test or **Quality and Assurance Environment** (**Q&A**). In Q&A, the build is accepted and released for deployment into production:



*Figure 13.5: Conceptual diagram of a build and release pipeline*

If all these steps are followed, an organization can start working in DevOps teams using CI/CD. But how will this work in multi-cloud? We’ll discuss this in the next section.

**Working with CI/CD in multi-cloud**

The development of code for applications can be cloud-agnostic, meaning that it doesn’t matter to which cloud the code is pushed: the functionality of the code remains the same. That’s the reason why we use containers: to abstract the code from the underlying infrastructure. However, a lot of developers will discover that it does matter and that it’s not that simple to develop in a truly multi-cloud fashion.

In multi-cloud, developers also work from one repository, but during deployment, platform-specific configuration might be added and tested, even when we utilize Kubernetes platforms in our clouds. You will find that Kubernetes is the same, but the various implementations of Kubernetes platforms in clouds might differ.

There are a few steps that developers have to take to make it successful. First, the DevOps way of working should be consistent, regardless of the platform where applications will eventually land. A company might want to run applications in Azure, AWS, GCP, or even on-premises, but the way application code is written in DevOps cycles should be the same. As said, each of these platforms will have specific features to run the code, but that’s a matter of configuration. The process called *staging* is meant to find out if the application package, including the configuration of the underlying infrastructure, is ready for release to production.

We need to think in terms of layers: abstracting the application layer from the resources in the infrastructure and the configuration layer. That’s the only way to get to a consistent form of application development with CI/CD. One other challenge that developers need to tackle in multi-cloud environments is the high rate of changes in the various cloud platforms. DevOps tools must be able to adapt to these changes in deployment, but without having to constantly change the application code itself.

Developers, however, need to have a good understanding of the target platforms and their specifics. It also makes sense to study the best practices in DevOps for these platforms and the recommended tooling: this will be discussed in more detail in the *Exploring tooling for CI/CD* section of this chapter about tooling for CI/CD. Most common tools are cloud-agnostic, meaning that they can work with different clouds, leveraging native APIs.

So, the ground rules for the successful implementation of DevOps are as follows:

* One repository
* One framework or way of working
* A unified toolset that can target multi-cloud environments

In terms of one framework, SAFe by Scaled Agile should be mentioned. **SAFe** stands for **Scaled Agile Framework**, and it’s used by a lot of enterprises as the foundation of DevOps. One of the key assets of SAFe is the Agile Release Train.

The **Agile Release Train** is built around the principle of the standardization of application development and the release management of code. This is done by automating everything. The steps that must be included are:

* Build by automating the process of compiling the code
* Automated unit, acceptance, performance, and load tests
* Automated security tests
* Continuous integration, automated by running integration tests and releasing units that can be deployed to production
* Continuous delivery by automated deployments to different environments of the version-controlled code
* Additional automation tools for configuration, provisioning, security by design, code review, audit trail, logging, and management

This supports the application life cycle management, continuous improvement, and release of the application code. Details and courseware on SAFe can be found at <https://www.scaledagileframework.com/DevOps/>.

**Exploring tools for CI/CD**

The tooling landscape for CI/CD and DevOps is massive and changes almost every month. There’s no right or wrong answer in choosing the toolset, as long as it fits the needs of the enterprise and people are trained in the usage of the tools. In this section, the native CI/CD tooling in the major clouds is discussed: Azure DevOps, AWS CodePipeline and CloudFormation, and Google Cloud Build. We will also look at tooling for Alibaba Cloud and OCI.

First, we will look at Kubernetes since that will likely be the platform that we’ll use in multi-cloud. The key components of Kubernetes CI/CD are:

* Container management
* Cluster operations
* Configuration management for the underlying infrastructure
* Version control system
* Image registries for container images
* Security testing and audits
* Continuous monitoring and observability

To create a good pipeline, we should consider the following design principles:

* All-in-one CI/CD tool instead of cloud-specific solutions when we plan to have CI/CD multi-cloud
* Managed or self-managed CI/CD, including the deployment, configuration, and management of infrastructure
* Automated or manual code testing and validation
* Deployment methodologies: rolling or blue/green

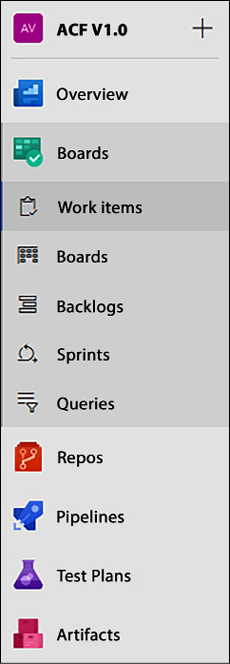
Now, what do clouds have to offer? We will look at the various solutions.

**Azure DevOps**

Azure DevOps enables teams to build and deploy applications; it caters to the full development cycle. Azure DevOps contains the following:

* **Boards**: This is the planning tool in Azure DevOps and supports scheduling with Kanban and Scrum. Kanban works with cards, moving tasks through stages, while Scrum works with short sprints to accomplish tasks.
* **Repos**: This is the repository in Azure DevOps for version control based on Git or **Team Foundation Version Control** (**TFVC**). The term Team Foundation still refers to the original name of Azure DevOps: Visual Studio **Team Foundation Server** (**TFS**).
* **Pipelines**: This is the CI/CD functionality in Azure DevOps, which supports the build and release of code to cloud environments. It integrates with repos and can execute scheduled tasks from Boards.
* **Test Plans**: This allows teams to configure test scripts manually and automatically.
* **Artifacts**: This feature allows developers to share code packages for various sources and integrate these into pipelines or other CI/CD tooling. Artifacts support Maven, **Node Packet Manager** (**NPM)** (for Node.js and JSON), NuGet packages, and Python package feeds.

The following screenshot shows the main menu of Azure DevOps with a project defined as Scrum that divides the work items into backlog items and sprints:



*Figure 13.6: Main menu of Azure DevOps*

When a team starts in Azure DevOps, the first thing to do is to define a project and assign project members. Next, the project manager or product owner defines the development methodology. Depending on that choice, DevOps presents the possibility to define work items, such as features, backlog items, and tasks, and a board to plan the execution of these items. A work item or product can be a piece of code that may be deployed; in DevOps, this can be automated with Azure Pipelines, which after review, deploys the code to the target cloud environment.

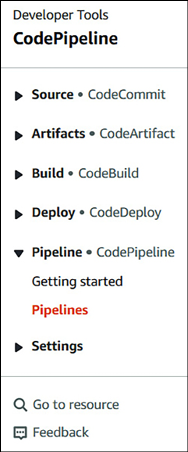
Developers can use Azure DevOps for AWS too using the AWS Toolkit for Azure DevOps. The toolkit even accepts the use of AWS CloudFormation templates to provision and update AWS resources within Azure DevOps. In this chapter’s *Further reading* section, a link to the documentation has been added.

**AWS CodePipeline**

AWS CodePipeline is the CI/CD tool for AWS and offers development teams a tool to deploy applications and infrastructure resources. CodePipeline provides the following:

* **Workflow modeling**: You could see this as the planning tool in CodePipeline. Workflow modeling defines the different stages for the release of code: build, test, and deploy. Teams can create tasks that need to be executed in the different stages.
* **Integrations**: As with any CI/CD tool, CodePipeline works with version control for the source code. With Integrations, developers can use various sources, such as GitHub, but also the native AWS service CodeCommit (the default in the main menu of CodePipeline), Amazon **Elastic Container Registry** (**ECR**), and Amazon S3. Provisioning and updating code is done with AWS CloudFormation. AWS Integrations can do a lot more, such as continuous delivery to serverless applications with the **Serverless Application Model** (**SAM**) and automating triggers with AWS Lambda functions to test whether application code has been deployed successfully.
* **Plugins**: It looks like AWS mainly uses its own tools, but developers absolutely have freedom of tools. AWS Plugins allow the use of GitHub for version control and Jenkins for deployment, for example.

The following screenshot shows the main menu of CodePipeline:



*Figure 13.7: The main menu of AWS CodePipeline*

As shown in the screenshot, creating a pipeline from the CodePipeline main menu will start by pulling code from a repository that sits in CodeCommit. The pipeline itself is built in CodeBuild and deployed in CodeDeploy.

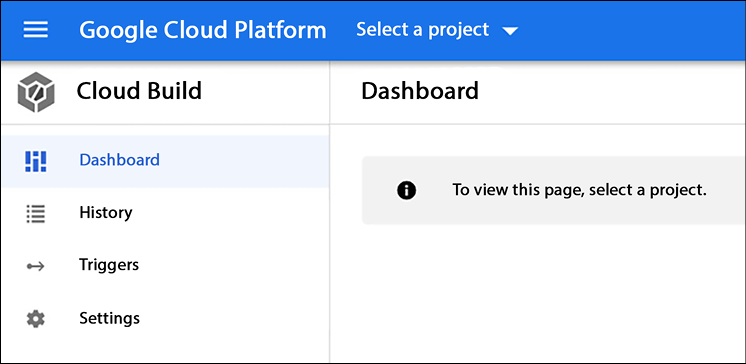
Be aware that Artifacts is not the same as it is in Azure. In AWS, Artifacts uses an S3 artifacts bucket where CodePipeline stores the files to execute actions in the pipeline.

**Google Cloud Build**

The CI/CD tool in GCP is Cloud Build. The main functions in Cloud Build are as follows:

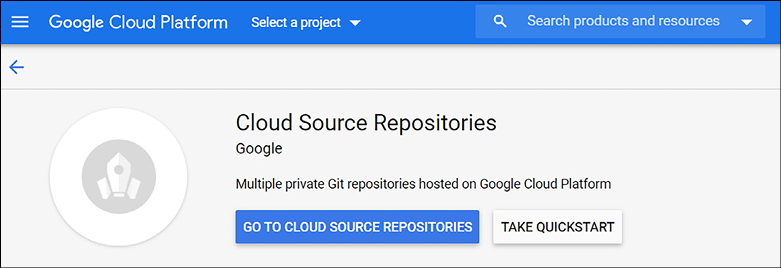
* **Cloud Source Repositories**: These are private Git repositories that are hosted on GCP. This is where the pipeline workflow starts: developers can store, access, and pull code from this repository using Cloud Build and Cloud Pub/Sub. Creating a repository can be done through the GCP UI portal or Google Cloud Shell with the gcloud source repos create command. After the creation of the repository, developers can start pushing code to it with the git add, git commit, and git push commands from the gcloud console.
* **Artifact Registry**: This is basically the same service as Artifacts in Azure DevOps. It allows the creation and management of repositories that hold Maven and NPM packages. In GCP, Artifact Registry is also used to create repositories for Docker container images.
* **Cloud Build**: This is the engine of the CI/CD functionality in GCP. In Cloud Build, developers define the pipelines. It imports source code from Cloud Source Repositories but can also pull code from other sources, such as GitHub. Cloud Build tests and deploys the code to the targeted GCP infrastructure. Cloud Build integrates with a lot of different solutions—for example, with Jenkins and the open source tool Spinnaker for automated testing and continuous delivery. These solutions can also be used to work with GKE to enable CI/CD on container platforms running Kubernetes.

*Figure 13.8* shows the menu of Cloud Build:



*Figure 13.8: Introduction screen to start with Cloud Build in the GCP console*

The main menu is very lean, as shown in the preceding screenshot. Only when a project is defined and started can developers start using Cloud Build to create the code repository. That service is available from the console, as shown in *Figure 13.9*:



*Figure 13.9: Starting a repository in GCP’s Cloud Source Repositories*

So far, we have discussed the CI/CD tools that are available in the major public clouds. It must be noted that these tools are provider-centric, meaning that AWS CodeLine, for instance, is very suitable to build in AWS. There are extensions and toolkits available to run pipelines from Azure DevOps to AWS and GCP. In that case, the toolkit will pick up the actions from Azure DevOps Pipelines and transport these to AWS CloudFormation, which will deploy the resources and code using CloudFormation Stack.

Alternatives are to use agnostic tools such as GitHub and Jenkins. GitHub is a relevant CI/CD tool. It allows you to deploy to Azure, AWS, GCP, and any other cloud provider that supports OpenID Connect to access cloud resources.

Over the years, Jenkins has also evolved to become an industry-leading standard for automated CI/CD since it offers different tools, languages, and automation tasks in creating pipelines. Jenkins is platform-agnostic and can target all clouds that we discuss in this book, including the final two that we still have to explore: Alibaba Cloud and OCI.

**CI/CD in Alibaba Cloud**

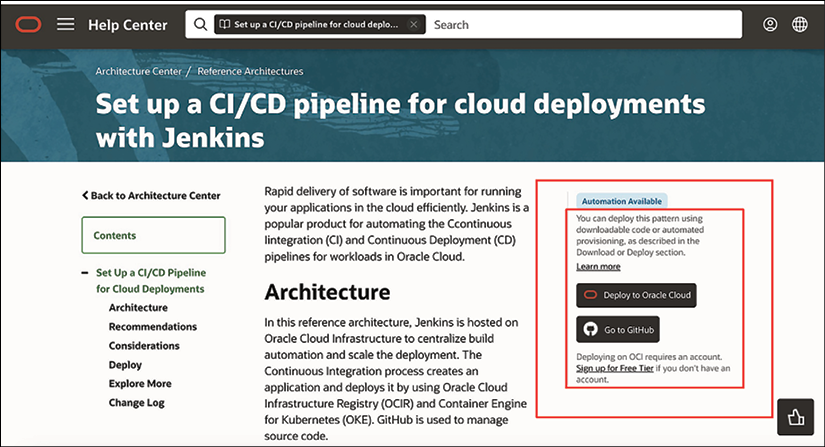
Key in the Alibaba Cloud solution for CI/CD are the **Kubernetes Container Service** and the **Alibaba Cloud Image Service**. The latter is used for image replication and image signature. The service enables fast replication across clusters and regions in Alibaba Cloud. With image signature, we can use Key Management Service to add signatures to Kubernetes images and security policies.

Like other public clouds, Alibaba Cloud utilizes commonly known and industry-standards technologies and tools in the CI/CD chain. We can, for instance, use ArgoCD to deploy applications to clusters. **ArgoCD** is a Kubernetes controller that automates the deployment and lifecycle management of applications in a declarative way through application definitions and configurations. ArgoCD continuously compares the running applications with the desired state. The technologies are integrated into Application Center in Alibaba Cloud.

**CI/CD in OCI**

We can use Jenkins to set up CI/CD architectures in OCI. In that case, we host the Jenkins master node in OCI. Application code is deployed through the **Oracle Cloud Infrastructure Registry** (**OCIR**) for Docker images and the **Container Engine for Kubernetes** (**Oracle Kubernetes Engine** — **OKE**), which deploys worker nodes with Oracle Linux. OCI uses GitHub to manage the source code and Terraform to automate the infrastructure.

A cool feature is that we can immediately deploy this solution to OCI from the website <https://docs.oracle.com/en/solutions/cicd-pipeline/index.html#GUID-D741A2F7-7E15-44ED-8B57-CB7197FE5D07> by clicking **Deploy to Oracle Cloud** or get the Terraform code in Github. It’s shown in the image below:



*Figure 13.10: Option to deploy CI/CD with Jenkins to OCI*

It is worth mentioning that OCI has recently leased a new service with OCI DevOps, which is a complete CI/CD platform. More information about this can be found at <https://www.oracle.com/devops/devops-service/>.

**Tools for multi-cloud container orchestration and application development**

The tools and technologies that we discussed so far are mostly cloud-native and hosted by a specific cloud provider. In multi-cloud, we might have Kubernetes clusters and containers in various clouds. Some of the tools have APIs to use in other clouds, such as Azure DevOps, which can be used for AWS too, using the AWS Toolkit as an extension.

Two other popular options for multi-cloud container orchestration are VMware’s Tanzu Mission Control and Red Hat OpenShift. Tanzu allows for managing Kubernetes clusters in multiple clouds through one console, including centralized policy management. Tanzu Mission Control supports the deployment, scaling, and management of clusters on AWS, Azure, GCP, OCI, and, of course, clouds that are built on vSphere.

Red Hat OpenShift is also an agnostic environment to deploy, scale, and manage Kubernetes clusters using the Red Hat suite with Red Hat Enterprise Linux and Ansible for configuration management of clusters. OpenShift has editions for Azure, AWS, and IBM Cloud. A fully managed RedHat OpenShift Dedicated edition is available for AWS and Google Cloud.

At the beginning of this section, it was mentioned that there are a lot of tools available to developers for developing, storing, testing, and deploying code. The main principles of CI/CD are the same, but there are differences in the way that these tools deploy code and especially how they test and validate code in an automated way. The tools discussed cover a lot of functionality already, but typically, additional tools are required. Just to name a few of the most used and well-known ones:

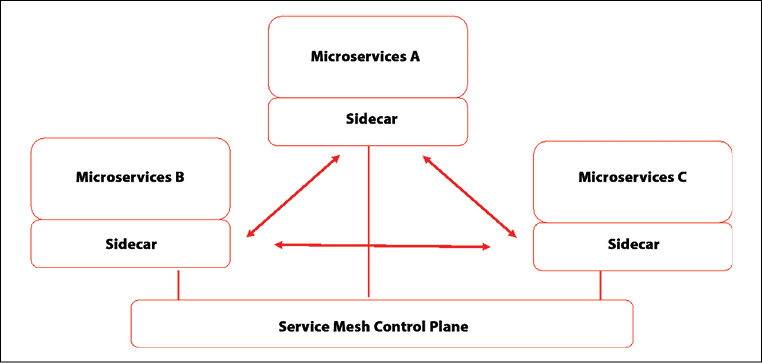
* GitHub and CI/CD with GitHub Actions to manage repositories
* **GitLab**: To manage code, but also a tool that makes deployment of Kubernetes clusters easy with Auto Build, executing Docker builds, and managing Docker files
* **GitOps**and**Spinnaker**: For cloud-agnostic continuous deployment

We have established a DevOps culture and built CI/CD pipelines. The next step is securing the pipelines and the code that we are developing. This is the topic of the next section.

**Following the principles of Security by Design**

In the previous sections, we designed our CI/CD pipelines. But as we have concluded, security starts at the very beginning of DevOps and should be integrated throughout the entire process, from the moment the code is pulled or new code is written up until deployment to production. We need to apply security by design. This is the second layer of DevSecOps and includes the following activities:

* **Securing pipelines**: A best practice is to apply zero-trust principles to the pipeline. Pipelines should only be accessed through least privilege policies. Also, continuous testing must be integrated into the pipeline. This includes **Static Application Security Testing** (**SAST**) and **Dynamic Application Security Testing**(**DAST**)**,**but also penetration testing to find any backdoors in the pipelines or any other vulnerabilities.
* **Clean code practice**: This one is subject to multiple interpretations, but the key is the principle of KISS: keep it simple, stupid. Try to keep the code as simple and short as possible. Document it well, so other developers know what the code is about. Most important of all: be consistent.
* **Application security design principles**: We can use the **Open Web Application Security Project** (**OWASP**) security design principles for this. These principles are based on CIA: confidentiality, integrity, and availability. In essence: allow only access to data to which the user is authorized, make sure that data is not altered in an unidentified way or by unauthorized users, and that data is available to authorized users when they request it. Following this, OWASP lists security principles, amongst others:
  + Limit the attack surface as much as possible.
  + Applications are secured by default, meaning that users have to ‘prove’ they are eligible to use the application and the associated data.
  + Least privilege.
  + Defense in depth using layered security controls.
  + Separation of duties, ensuring that users can only perform actions that they are entitled to without having control over the lifespan of an entire transaction in an application. Simply explained: someone initiating a request can’t authorize the request by themselves.
* **Microservices with containers and sidecars**: Microservices are loosely coupled services that together form the functions of an application. They are considered to be more secure than monolith architectures where all functions are integrated into one big application. Because microservices are decoupled and communicate with each other using APIs, breaches that exploit vulnerabilities are often limited to one particular microservice instead of breaching the entire application.
* Keep in mind that the security of a system depends on a multitude of factors, and the architecture is just one aspect of it.
* Microservices with containers and sidecars can offer security benefits, such as increased isolation and modularity, which can make it easier to manage security vulnerabilities in individual components. However, the complexity of managing multiple containers and services can also introduce new security challenges, such as securing the communication between components and ensuring the proper configuration of security settings.
* Monolith architectures, on the other hand, may have fewer components to secure, but a single vulnerability in the monolithic codebase can have a larger impact. The size and complexity of a monolithic codebase can also make it difficult to identify and remediate security vulnerabilities.
* However, building a microservices architecture is complex, and developers would still want to be able to reuse artifacts instead of programming every microservice over and over again. This will likely be the case with functions that apply to every application. These functions can be coupled using sidecars. In sidecars, components of an application are deployed as a separate process using containers, providing isolation and encapsulation. The sidecar is, like a real sidecar to a motorcycle, attached to a parent application and provides supporting features for the application. We can use sidecars to deploy monitoring, logging, and configuration files. *Figure 13.11* shows the pattern for sidecars:



*Figure 13.11: Architecture pattern for sidecars*

* To integrate the sidecars with the other components of the applications, we need a service mesh, which enables service-to-service-communication between microservices. An industry-leading technology for using service mesh with sidecars is the open source Istio service mesh. This is supported by all public clouds using Kubernetes as a container orchestration platform.
* **Conduct threat modeling**: Organizations should adopt the process to identify potential threats and evaluate these against the security policies and implemented safeguards in the application and data stacks. Threat modeling will reveal vulnerabilities and indicate what mitigating actions must be taken to prevent these vulnerabilities from being exploited. A common way to perform threat modeling is using **Tactics, Techniques, and Procedures** (**TTP**), amongst others used by the Mitre Att@ck framework. Tactics describe the behavior of an attacker and what and how techniques are used to breach a system.

Testing applications and infrastructure is a critical component of software development lifecycles. This includes testing to verify that vulnerabilities and potential risks have been addressed. With that, it’s best practice to consider the full stack:

* Code
* CI/CD pipelines
* Registries and repositories holding container images and services
* IaC used to deploy resources in the cloud
* Hosts in the cloud (virtual machines)
* Container runtime environments (Kubernetes)
* Containers

Something that we haven’t touched upon yet is serverless functions to run containers in the cloud. Examples are Azure Container Instances, AWS Fargate, and Google Cloud Run; both offering compute as a service. Fargate is a serverless, completely managed service that builds infrastructure and is compatible with EKS in AWS. Google Cloud Run is not quite the same but it does run applications without you having to worry about the underlying infrastructure. Hence, there’s no need to deploy your own virtual machines and configure clusters. It’s a high level of automation that we can use in the cloud. In the next section, we will discuss automation in more detail.

**Securing development and operations using automation**

Automation is a must in DevSecOps. At a minimum, we should consider automating the following processes:

* Version Control
* Continuous Integration
* Continuous Testing
* Configuration Management and Deployment
* Continuous Monitoring

In the previous section, we talked about containers, sidecars, and orchestration. Clouds also offer tools to automate this as a completely managed service. Containerization with automated orchestration offers great benefits to companies. Orchestration platforms such as Kubernetes and the various managed Kubernetes services in public clouds take care of the installation, scaling, and management of containerized workloads and services, including debugging and deployment of new versions of applications.

Containers are isolated by default. One failing container will not impact other containers. Individual containers can be updated without causing downtime for other containers. This also means that containers are protected very well from malicious code spreading through entire environments. However, it does mean that we have to test intensively. These tests can be automated too to a large extent:

* Application and application runtime security testing
* **Software composition analysis** (**SCA**)
* IaC validation and testing
* Container image scanning
* Dynamic threat analysis
* Network scanning
* Vulnerability scanning and management

One item that we must discuss is the method of deployment from a DevSecOps perspective. There are two common ways to do deployments: rolling and blue/green. Deciding between one of the two is a decision that is key in DevSecOps thinking. In **rolling** deployments, we replace the old versions of the application and the underlying infrastructure with new versions. In containerized environments, we will replace all containers that run the previous versions of the application one by one. This is generally the fastest way of deployment.

However, there will be no isolation between the old and the new containers. A more secure way of deployment is **blue/green**. The blue environment is the old environment and the new one is green. Both will run in production at a certain time, where traffic is gradually transferred from blue to green. The two environments are strictly separated. Only when the traffic on blue is completely transferred and the green line tested can the old environment can be decommissioned. The blue line will be a fallback along the entire process.

This way of deployment is safer but also more costly since multiple resources will run at the same time. Container pods are copied and used to set up the green line. Tools such as Kubernetes and Jenkins support the blue/green deployment very well using the principle of Declarative Deployment, which allows for automated, fast copying of container pods.

This concludes the chapter about setting up DevSecOps and CI/CD. But we’re not done yet with security. In the next chapter, we will look at the security policies themselves and how to apply them to cloud environments.

**Summary**

After completing this chapter, you should have a good understanding of the DevOps way of working and the use of CI/CD pipelines in cloud environments. Everything is coded in the cloud, from the application to the infrastructure and the configuration. Code needs to be stored in a central repository and brought under version control. That’s where the CI/CD pipeline starts. Next, the DevOps team defines the phases of the pipeline, typically build, test, and deploy. Actions in these phases are automated as much as possible.

We learned that we must integrate security from the moment developers start writing or pulling code. Access to the code, the code itself, the pipelines, and the target clouds that we use must be secured and protected from breaches by identifying vulnerabilities and taking appropriate mitigating actions. We studied the various technologies and tools that provide CI/CD functionality and explored how we can protect our applications when we work with CI/CD by applying the principles of security by design.

Prevention, protection, and the defense of our applications and data are defined in policies that developers and operations need to comply with. In the next chapter, we will discuss how we can define security policies in cloud environments.

**Questions**

1. All cloud providers offer solutions to run and manage Kubernetes clusters. Name three of these solutions.
2. What is the function of GitHub?
3. Microservices will likely share functions such as monitoring and logging. We can use sidecars to attach these functions to microservices. We need to “merge” these functions with our microservices. What do we call this process of integrating services using sidecars?
4. We mentioned two ways of deployment: rolling and blue/green line. Which one is considered to be more secure?

**Further Reading**

* **DevOps Agile Skills Association** (**DASA**): [https://www.DevOpsagileskills.org/](https://www.devopsagileskills.org/)
* Documentation on Azure DevOps: <https://docs.microsoft.com/en-us/azure/DevOps/get-started/?view=azure-DevOps>
* Information on integrating Azure DevOps with AWS: <https://aws.amazon.com/vsts/>
* Documentation on AWS CodePipeline: <https://aws.amazon.com/codepipeline/>
* Documentation on Google Cloud Build: <https://cloud.google.com/docs/ci-cd/>
* Documentation on CI/CD for Alibaba Cloud: <https://www.alibabacloud.com/solutions/devops/CI-CD>
* Documentation on CI/CD for OCI: <https://docs.oracle.com/en/solutions/cicd-pipeline/index.html>
* *Hands-On DevOps for Architects*, by Bob Aiello, Packt Publishing

**Join us on Discord!**

Read this book alongside other users, cloud experts, authors, and like-minded professionals.Ask questions, provide solutions to other readers, chat with the au