**18**

**Developing for Multi-Cloud with DevOps and 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 life cycle of applications 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.

To speed up this process, organizations work in DevOps cycles, using release cycles for applications 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.

We will study the principles of DevOps, how CI/CD pipelines work with push and pull mechanisms, and how pipelines are designed so that they fit multi-cloud environments. Next, we will discuss how we must secure our DevOps processes using the principles of the DevSecOps Maturity Model and the most common security frameworks.

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

* Introducing DevOps and CI/CD
* Using push and pull principles in CI
* Designing the multi-cloud pipeline
* Using the DevSecOps Maturity Model
* Automating security best practices using frameworks

**Introducing DevOps and CI/CD**

Before we get into the principles of DevSecOps, we need to have a good understanding of DevOps. 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**). It defines 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. But DevOps applies to more than *just* the application: it’s also about the processes, the people, and the tools. DevOps, at its core, is a culture, a mindset.
* **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.

DevOps has been described in the literature as *culture*, a new way of working. 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**.

**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 environment. 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. We have to make a note here: continuous delivery and continuous deployment are not the same thing.

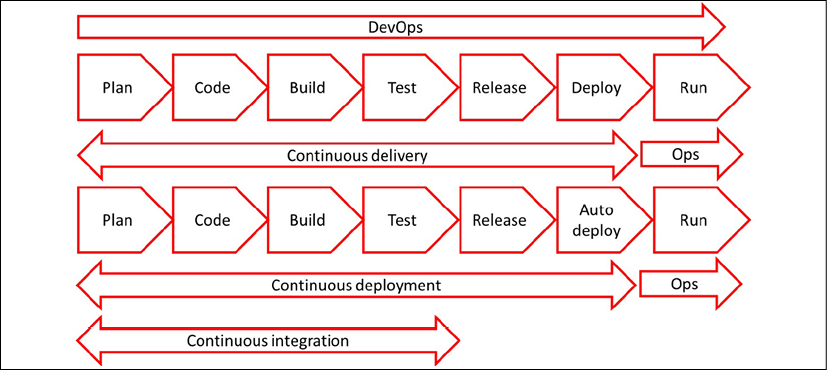
Continuous delivery is putting the artifacts, built in the CI process, in environments, typically development, staging, and production, one after the other, with testing and approvals in between. Continuous deployment means code being put into the production environment from CI in a completely automated way without any human intervention.

The following are some key points to differentiate both practices:

* Continuous delivery usually takes time from development to production; weeks or days in the best case.
* The kinds of changes applied under continuous deployment can go to production several times per day.

To learn more, we refer you to the following blog by Martin Fowler: <https://www.martinfowler.com/bliki/ContinuousDelivery.html>

**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. *Figure 18.1* shows the DevOps cycle with CI/CD:



*Figure 18.1: DevOps cycle with CI/CD*

In the next section, we will study how to get started with CI/CD.

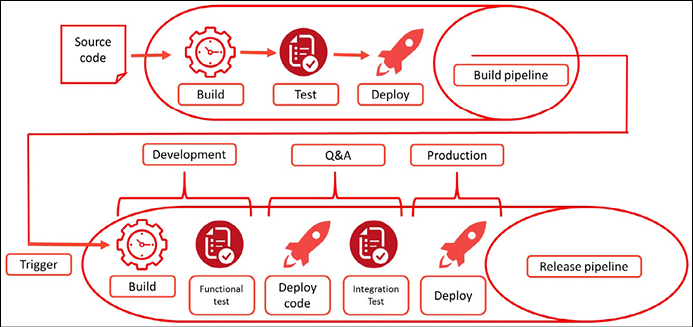
**Getting started with CI/CD**

CI/CD is widely adopted by enterprises, but a lot of projects fail. This section explains how enterprises can successfully implement CI/CD and how they can avoid pitfalls. The major takeaway should be that an implementation starts with consistency. That counts for cloud implementations as well as for CI/CD.

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. CI/CD and DevOps are, however, not about tools. 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 4 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. There are a lot of tools on the market that facilitate CI/CD. Jenkins is a popular one, but a lot of companies that use Azure choose to work in Azure DevOps. Next, GitHub Actions has gained a lot of popularity. Involve the people who have to work with the system daily and enable them to take a *test drive* with the application. Then, make a decision on the CI/CD system and ensure all teams work with that system. Again, it’s about consistency.
3. It’s advised to perform 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, application, or IT system that needs to be delivered. 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, acceptance criteria are signed off, and all functional and technical tests have passed.
   * *Figure 18.2* 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**(**Q&A**) Assurance environment. In Q&A, the build is accepted and released for deployment into production:



*Figure 18.2: Conceptual diagram of a build and release pipeline*

1. Automate as much as possible, as one of the principles of DevOps states. This means that enterprises will have to adopt working in code, including **Infrastructure as Code** (**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 both whenever they need to.

If all these steps are followed, organizations are able to start working in DevOps teams using CI/CD. In the next sections, CI/CD is explained in more detail, starting with version control, and then discussing the functionality of commits and push and pull mechanisms in the pipeline.

**Working under version control**

By working from one code repository with different teams, version control becomes crucial in CI/CD. Git and Subversion are popular version control systems that enable teams to organize their files that form the source code, test scripts, deployment scripts, and configuration scripts used for applications and infrastructure components. Everything is code, which means that systems consist of a number of code packages: the code for the VM, the code for how the VM should be configured based on policies, and the application code itself. A version control system also enables teams to retrieve the historical state of systems, in case a deployment fails or systems are compromised and need to be rebuilt.

Version control systems keep track of changes to files that are stored in the repository. In DevOps, these changes are commonly referred to as commits, something that we’ll discuss further in the next section, *Using push and pull principles in CI/CD*. A **commit** comprises the code change itself, along with metadata on who made the change and the rationale behind the code change. This ensures that code is kept consistent and, with that, repeatable and predictable. It also means that teams are forced to document everything in the repository and bring it under version control.

This list contains many of the items that need to be under version control:

* Application code
* API scripts and references (what is the API for?)
* IaC components such as components such as VMs, network devices, storage, images for operating systems, DNS files, and firewall configuration rules
* Infrastructure configuration packages
* Cloud configuration templates, such as AWS CloudFormation, **Desired State Configuration** (**DSC**) in Azure, and Terraform files
* Code definitions for containers, such as Docker files
* Container orchestration files, such as Kubernetes and Docker Swarm files
* Test scripts

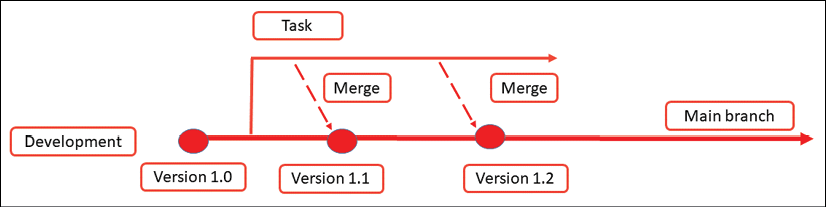
Once companies have implemented this, they need to maintain it. This is not a one-time exercise. Teams should confirm that version control is applied to application code, systems configuration, and automation scripts that are used in the CI/CD pipeline. Only if this is applied and used in a consistent way will enterprises be able to deploy new applications and systems rapidly, yet securely and reliably.

**Using push and pull principles in CI**

CI/CD pipelines work with branches, although other terms can be used for this. The main branch is sometimes referred to as a mainline or, when teams work in GCP, as a trunk. The most important principle to remember is that a development team has one main branch or mainline. Next, we will see two ways of pushing new code to that main branch in the following sections.

**Pushing the code directly to the main branch**

In this method, the developers work directly in the main code; they change small pieces of the code and merge these directly back into the main branch. Pushing code back to the main branch is called a **commit**. These commits are done several times per day, or at least as soon as possible. Working in this way ensures that releases can be done very frequently, as opposed to working in code forks that result in separate or feature branches, which are described in the second method. *Figure 18.3* shows how direct pushes to the main branch work:



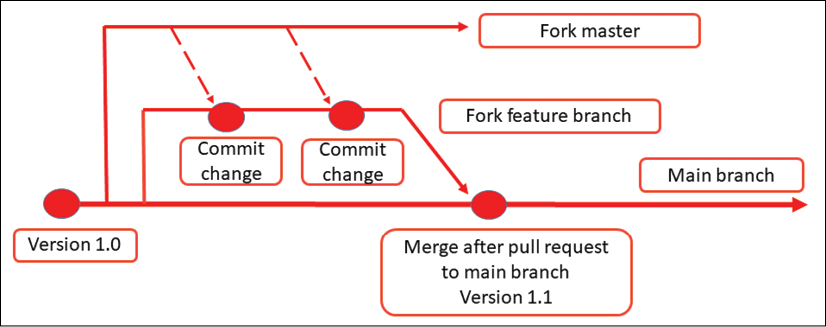
*Figure 18.3: Developers merging code directly to the main branch*

The idea behind CI is that companies get rid of long, complex integrations. Developers work in small batches of the code that they frequently commit to the main. The big advantage is that developers immediately see whether the change is done correctly, with the possibility to revert the change without having a huge impact on the main as a whole. This is DevOps—the developers are responsible for the build, the commit, and the result: you break it, you fix it. Automated tests that are executed after the code commit are crucial to ensure that systems keep running without failures.

**Pushing code to forks of the main**

In this method, teams copy code from the main and create a separate or feature branch. This is also referred to as **forking**: developers create a feature branch by taking a copy from the source code on the main branch. They do their development on this **forked code**. In GCP, this is not trunk-based development, or better said: this is referred to as feature-driven development.

This method is often used for major developments, creating new features. Developers can work in isolation on the forked code, and when they’re done, commit the code back to the main branch, merging the new features or builds with it. The downside is that this can lead to complex integrations. This can’t be done on a frequent basis as intensive testing is required before the merging takes place. *Figure 18.4* shows how feature branches operate in a workflow:



*Figure 18.4: Developers working in a feature branch before merging to the main branch*

In both methods, code gets pushed to the repository in, for example, GitHub. As soon as a developer has committed their code, they execute a pull request. This is the stage where the new, changed code is reviewed before the changes are actually merged into the main branch.

**Best practices for working with CI/CD**

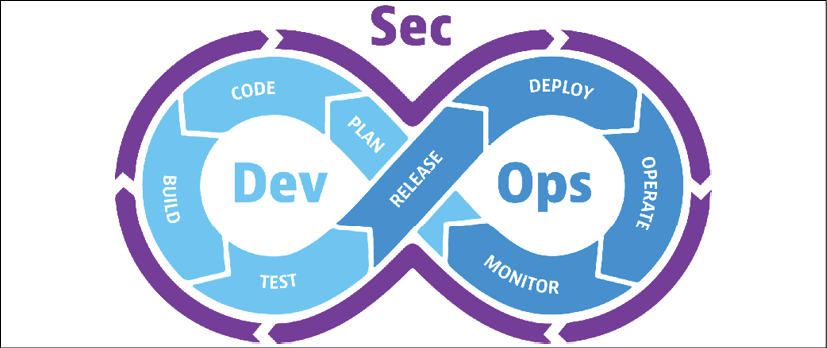
There are a few best practices to remember when working with CI/CD. One of the biggest pitfalls is that code reviews can often be too extensive, meaning that developers have to get approval from different stakeholders before they can push the code to production. This will cost a lot of time and effectively slow down the DevOps process. Companies that adopt DevOps should have two principles implemented:

* The **four-eyes principle**: Have code reviewed while it’s being written by working in developer pairs, where the second developer reviews the code of the first developer. This is also referred to as **pair programming**. Peer review is another method: here, the authors of the code each review at least one other developer’s work, typically at the end of the development process.
* Running automated test scripts is most important. These scripts must be executed before code is actually committed to the main branch to make sure that systems keep functioning after the code commit.

By following these principles, we are already applying the first principles of DevSecOps. But it’s not enough: we must really integrate security into DevOps. To do that, we can work with the DevSecOps Maturity Model. We will study this in the next section.

**Using the DevSecOps Maturity Model**

Security is not a sauce that we put on top of products when they are finished. Security policies have to be applied from the first moment of development, all the way up to deployment to production. That’s where **DevSecOps** comes in. The position of security in the DevOps cycle is shown in the following diagram:



*Figure 18.5: The DevSecOps cycle*

The **DevSecOps Maturity Model** of the **Open Web Application Security Project** (**OWASP**) is a framework that helps organizations assess and improve their software development and delivery practices. The model aims to integrate security practices into the DevOps process that we described in the previous sections. By using this model, businesses can improve the security of their software products and reduce the risk of data breaches and cyber-attacks.

The DevSecOps Maturity Model can be found at <https://owasp.org/www-project-devsecops-maturity-model/>.

The OWASP **DevSecOps Maturity Model** (**DSOMM**) is divided into five levels, each representing a different level of maturity in terms of security integration into the DevOps process. These five levels represent where companies stand in terms of the maturity of their DevSecOps implementation:

* **Level 0—No security culture**: There is no formalized security program or processes, and security is not a consideration in the software development process.
* **Level 1—Siloed security**: Security is taken into account but is limited to a separate team or department. Security is not integrated into the software development process, and there is no collaboration between the security and development teams.
* **Level 2—Integrated security**: Security is integrated into the software development process and there is collaboration between security and development teams. However, the security process is not yet automated, and security testing is not performed in every stage of the development process.
* **Level 3—Continuous security**: Security is integrated into every stage of the software development process. Security testing is automated, and security checks are performed continuously throughout the development process.
* **Level 4—Continuous improvement**: The DevOps process is continuously monitored and improved to ensure that security is always a top priority. The organization has a culture of security, and security is considered as part of every decision and action taken.

Now, let’s look at how businesses can use this maturity model to improve their DevSecOps practices:

1. **Assess current security practices**: The first step in using the DevSecOps maturity model is to assess the current security practices of the organization. During the assessment, we identify the current level of security integration into the DevOps process and consider the gaps and weaknesses in the process.
2. **Set goals for security integration**: The goals should be **Specific, Measurable, Achievable, Relevant, and Time-Constrained** (**SMART**). For example, the organization may set a goal to automate security testing in every stage of the development process by the end of a specified period.
3. **Create a roadmap for improvement**: The roadmap should include specific actions and milestones that need to be achieved to reach the desired level of maturity. The roadmap should also include timelines and responsibilities for each action item.
4. **Implement security automation tools**: We must automate security testing in every stage of the development process. This can be achieved by implementing security automation tools, such as **Static Application Security Testing** (**SAST**) and **Dynamic Application Security Testing** (**DAST**) tools. These tools can identify vulnerabilities and weaknesses in the software code and provide developers with feedback on how to fix them. With this, we reach Level 3 of the DSOMM.
5. **Integrate security into the DevOps process**: Security aspects must be considered in the design, development, testing, deployment, and operations phases of the process. This is driven by culture: the mindsets of all of those involved have to be focused on developing and deploying software that is secure. To achieve this, the organization needs to create a culture of security, where everyone in the organization is responsible for the security of the software. This is the target of Level 4 of DSOMM.

We have been talking about software in this section, but DevSecOps also includes the underlying infrastructure in the clouds that we use to run our software. In major clouds such as Azure, AWS, and GCP, infrastructure is provided through a variety of services, including virtual machines, containers, and serverless computing, as we have learned throughout this book. These infrastructure services can be configured and managed using IaC tools such as AWS CloudFormation, **Azure Resource Manager**(**ARM**), and Google Cloud Deployment Manager. IaC tools allow infrastructure to be treated as code, enabling developers and operations teams to version, test, and automate infrastructure changes just like they would with software changes.

DevSecOps teams can leverage the infrastructure services and IaC tools provided by the cloud providers to implement security controls throughout the software development life cycle. For example, they can use **Identity and Access Management** (**IAM**) policies to control access to resources, implement network security controls to protect against attacks, and use logging and monitoring to detect and respond to security incidents. For that reason, cloud infrastructure must also be included in the DevSecOps practices.

One of the first topics that we must address in security and DevSecOps is observability: we need to be able to see what is happening in our pipelines and detect issues in a timely manner. That’s the topic of the next section.

**Manage traceability and auditability**

DevSecOps starts with observability in order to enable the management of traceability and auditability. This is becoming increasingly relevant in today’s complex cloud-native environments where companies execute multiple releases of their software per month, week, or even day. And in the context of multi-cloud models and workflows, they might release software across various clouds and use services from different providers.

Observability is essential for maintaining the security and stability of modern software systems. By prioritizing traceability and auditability, organizations can achieve a higher level of observability, enabling them to identify potential security threats and respond to them quickly and effectively. This will help them to reduce the risk of security incidents, improve the reliability and performance of software systems, and ensure compliance with regulatory requirements.

At its core, observability refers to the ability to collect and analyze data from across the entire software stack, including infrastructure, applications, and user interactions. This can be in one cloud, but also across stacks that are deployed in multi-cloud settings. Here, we certainly need to know what service is hosted where and how services are interconnected.

Two critical components of observability are traceability and auditability. Traceability refers to the ability to track and trace events and activities across the entire software system. This means that DevSecOps teams must be able to follow the flow of data and code through the system, identifying potential points of vulnerability and ensuring that security controls are in place at each stage of the process.

Auditability is closely related to traceability, but it focuses more on the ability to review and verify the actions taken by various stakeholders in the software development and deployment process. This includes developers, security analysts, operations teams, and other stakeholders who play a role in ensuring the security and stability of the software system. By maintaining a complete and accurate audit trail of all actions taken, organizations can ensure accountability and transparency while also facilitating more effective incident response and forensic analysis in the event of a security incident.

To achieve this level of observability, we must invest in the right tools and processes. This includes leveraging automation tools to collect and analyze data from across the software stack, the cloud infrastructure that we have deployed and manage with IaC, implementing robust logging and monitoring capabilities, and integrating security and compliance checks into every stage of the software development and deployment process. In multi-cloud systems, we therefore need “agnostic” tools that can work with various clouds and software stacks. Popular examples of such tools are:

* **Prometheus**: An open-source monitoring system that can be used to collect and query metrics from multiple sources, including different cloud providers. It supports a variety of data sources and can be integrated with many different tools.
* **Grafana**: Like Prometheus, Grafana is open source. It helps to visualize and analyze processes to create dashboards and alerts based on data from multiple sources, including various cloud stacks.
* **Fluentd**: Fluentd is an open-source data collector that can be used to collect, process, and forward log data from multiple sources, including different cloud providers. It supports a variety of outputs and can be configured to integrate with many different tools, making it a flexible choice for multi-cloud observability.
* **Jaeger**: Jaeger is an open-source distributed tracing system that can be used to trace requests across multiple services and cloud providers. It supports various tracing protocols and can be integrated with many different tools, making it a good choice for multi-cloud observability.
* **OpenTelemetry**: OpenTelemetry is a vendor-neutral observability framework that provides a standard way to collect, process, and export telemetry data from multiple sources, including different cloud providers. It supports various programming languages and can be integrated with many different tools, making it a popular choice for multi-cloud observability.

Choosing the right tools is one thing, but what do we monitor against? The answer to that question is: we implement security guardrails and guidelines to be secure and compliant. We can use security frameworks to help us in setting the appropriate levels when securing our platforms. This is the topic of the final section of this chapter.

**Automating security best practices using frameworks**

The hardest part in getting security to the appropriate level in organizations is to define when the organization is compliant, and environments are “secure enough”—if such a thing exists. The problem with security in any IT environment is that just like cloud technology itself, the tactics, techniques, and processes used to attack environments are also evolving fast. Hackers will constantly find new ways to compromise environments. That’s why every team member in a DevOps team must be fully aware of security risks. Every choice that a team makes comes with a consequence that must be thought through in terms of security. Are we introducing a vulnerability or other risk by developing and deploying software or by using a specific cloud service? What do we need to do to protect the data, application, underlying infrastructure, connectivity, and ultimately, the user?

Frameworks such as OWASP, CIS, and MITRE ATT&CK can help in defining the level of risk and determining what security measures must be applied. The next step is to automate this. Automating security practices is an essential aspect of modern DevSecOps practices. By automating security practices, developers and operations teams can ensure that security controls are implemented consistently and effectively across their software and infrastructure.

Let’s have a more detailed look at the most commonly used frameworks—OWASP, the CIS Controls, and MITRE ATT&CK:

* The OWASP Top 10 is a list of the most critical security risks to web applications. To automate security practices based on the OWASP Top 10, organizations can use automated tools to scan their applications for vulnerabilities, such as SQL injection or **Cross-Site Scripting** (**XSS**). These tools can be integrated into the CI/CD pipeline to automatically test code changes for vulnerabilities before they are deployed. Additionally, organizations can use automated code analysis tools to identify vulnerabilities in code as it is being written.
* The CIS Controls are a set of guidelines for securing information systems. To automate security practices based on the CIS Controls, organizations can use configuration management tools such as Chef, Puppet, or Ansible to enforce security policies on infrastructure. These tools can be used to automatically configure security settings on servers, networks, and applications, ensuring that security controls are consistent across the organization. CIS has issued controls for each separate cloud, including Azure, AWS, GCP, Alibaba Cloud, and OCI.
* Lastly, the MITRE ATT&CK framework is a knowledge base of **Tactics, Techniques, and Processes**(**TTPs**). To automate security practices based on the MITRE ATT&CK framework, organizations can use tools that monitor network and system activity for signs of attack. For example, they can use **Security Information and Event Management** (**SIEM**) tools to detect suspicious activity and use automated incident response tools to respond to security incidents as they occur. Using the framework to define what specific vulnerabilities are in an environment is not an easy task. Some adjacent tools might be handy when working with MITRE ATT&CK. One example is DeTTECT, originally invented for the Dutch bank Rabobank, one of the biggest financial organizations in the Netherlands. It assists in using ATT&CK to score and compare data log source quality, visibility coverage, detection coverage, and threat actor behavior.

Mitre ATT&CK references can be found at <https://attack.mitre.org/> .

DeTTECT has a GitHub repository that can be found at <https://github.com/rabobank-cdc/DeTTECT>.

Automation can and must be used to implement security best practices, such as vulnerability management, identity and access management, and data protection. By automating security practices, we can achieve more effective and efficient security controls, reducing the risk of security incidents and improving an organizations, overall security posture.

This concludes our chapter on DevOps and DevSecOps. We have two more Ops-related concepts that we have to discuss, both of which are up and coming in the world of multi-cloud. The next chapter covers AIOps and GreenOps.

**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 code 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 discussed the concepts of push and pull in CI/CD pipelines using main and feature branches, describing the different methodologies to push and commit code to branches. If teams work consistently from one repository using a unified way of working, they can deploy code to different clouds.

Security must be intrinsic, from the first moment developers pull or start developing new code, all the way up to deployment to production. Adopting DevSecOps enables the integration of security practices into the DevOps cycle. To help us in achieving this, we can work with the DevSecOps Maturity Model from OWASP, which sets guidelines to improve security practices in the development of cloud environments.

In the next chapter, the final concepts of operations will be discussed: AIOps and GreenOps.

**Questions**

1. Systems must meet the acceptance criteria before they can be signed off as ready. In DevOps, a specific term is used for this sign-off—what is that term?
2. What’s the term that we use when we push code back to the main branch?
3. Name the three security frameworks that we discussed in this chapter and that help organizations in setting up security practices.
4. What are Prometheus and Grafana?

**Further reading**

You can refer to the following links for more information on the topics covered in this chapter:

* **DevOps Agile Skills Association** (**DASA**): [https://www.DevOpsagileskills.org/](https://www.devopsagileskills.org/)
* *Enterprise DevOps for Architects,*by Jeroen Mulder, Packt Publishing