You can embed these steps beautifully in the CI/CD pipeline example shown here:

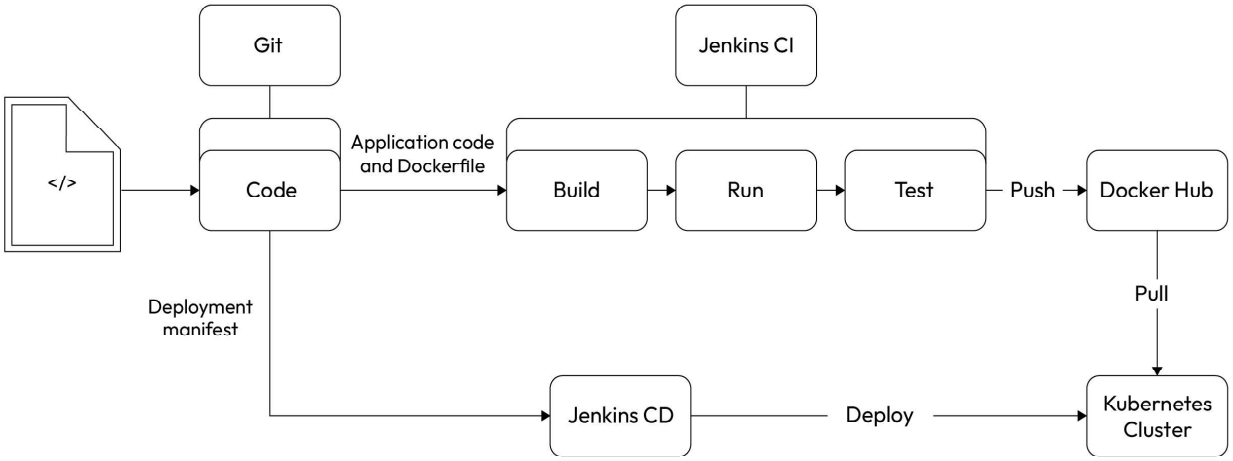

Figure 1.7 – Container CI/CD pipeline example

This means your application and its runtime dependencies are all defined in the code. You follow configuration management from the very beginning, allowing developers to treat containers like ephemeral workloads (ephemeral workloads are temporary workloads that are dispensable, and if one disappears, you can spin up another one without it having any functional impact). You can replace them if they misbehave – something that was not very elegant with virtual machines.

Containers fit very well within modern CI/CD practices as you now have a standard way of building and deploying applications, irrespective of the language you code in. You don't have to manage expensive build and deployment software as you get everything out of the box with containers.

Containers rarely run on their own, and it is a standard practice in the industry to plug them into a container orchestrator such as **Kubernetes** or use a **Container-as-a-Service** (**CaaS**) platform such as **AWS ECS** and **EKS**, **Google Cloud Run** and **Kubernetes Engine**, **Azure ACS** and **AKS**, **Oracle OCI** and **OKE**, and others. Popular **Function-as-a-Service** (**FaaS**) platforms such as **AWS Lambda**, **Google Functions**, **Azure Functions**, and **Oracle Functions** also run containers in the background. So, though they may have abstracted the underlying mechanism from you, you may already be using containers unknowingly.

As containers are lightweight, you can build smaller parts of applications into containers to manage them independently. Combine that with a container orchestrator such as Kubernetes, and you get a distributed microservices architecture running with ease. These smaller parts can then scale, auto-heal, and get released independently of others, which means you can release them into production quicker than before and much more reliably.

You can also plug in a **service mesh** (infrastructure components that allow you to discover, list, manage, and allow communication between multiple components (services) of your microservices application) such as **Istio** on top, and you will get advanced Ops features such as traffic management, security, and observability with ease. You can then do cool stuff such as **blue/green deployments** and **A/B testing**, operational tests in production with **traffic mirroring**, **geolocation-based routing**, and much more.

As a result, large and small enterprises are embracing containers quicker than ever, and the field is growing exponentially. According to `businesswire.com`, the application container market shows a compounded growth of 31% per annum and will reach $6.9 billion by 2025. The exponential growth of 30.3% per annum in the cloud, expected to reach over $2.4 billion by 2025, has also contributed to this.

Therefore, modern DevOps engineers must understand containers and the relevant technologies to ship and deliver containerized applications effectively. This does not mean that virtual machines are unnecessary, and we cannot completely ignore the role of IaaS-based solutions in the market, so we will also cover some config management with **Ansible** in further chapters. Due to the advent of the cloud, IaC has been gaining much momentum recently, so we will also cover **Terraform** as an IaC tool.

Migrating from virtual machines to containers

As we see the technology market moving toward containers, DevOps engineers have a crucial task – *migrating applications running on virtual machines so that they can run on containers*. Well, this is in most DevOps engineers' job descriptions and is one of the most critical things we do.

While, in theory, containerizing an application is as simple as writing a few steps, practically speaking, it can be a complicated beast, especially if you are not using config management to set up your virtual machines. Virtual machines that run on current enterprises these days were created from a lot of manual labor by toiling sysadmins, improving the servers piece by piece, and making it hard to reach out to the paper trail of hotfixes they might have made until now.

Since containers follow config management principles from the very beginning, it is not as simple as picking up the virtual machine image and using a converter to convert it into a Docker container.

Migrating a legacy application running on virtual machines requires numerous steps. Let's take a look at them in more detail.

Discovery

First, we start with the discovery phase:

- Understand the different parts of your applications
- Assess what parts of the legacy applications you can containerize and whether it is technically possible to do so
- Define a migration scope and agree on the clear goals and benefits of the migration with timelines

Application requirement assessment

Once the discovery phase is complete, we need to do the application requirement assessment:

- Assess if it is a better idea to break the application into smaller parts. If so, then what would the application parts be, and how will they interact with each other?
- Assess what aspects of the architecture, its performance, and its security you need to cater to regarding your application, and think about the container world's equivalent.
- Understand the relevant risks and decide on mitigation approaches.
- Understand the migration principle and decide on a migration approach, such as what part of the application you should containerize first. Always start with the application with the least amount of external dependencies first.

Container infrastructure design

Container infrastructure design involves creating a robust and scalable environment to support the deployment and management of containerized applications.

Designing a container infrastructure involves considering factors such as scalability, networking, storage, security, automation, and monitoring. It's crucial to align the infrastructure design with the specific requirements and goals of the containerized applications and to follow best practices for efficient and reliable container deployment and management.

Once we've assessed all our requirements, architecture, and other aspects, we can move on to container infrastructure design:

- Understand the current and future scale of operations when you make this decision. You can choose from many options based on your application's complexity. The right questions include; how many containers do we need to run on the platform? What kind of dependencies do these containers have on each other? How frequently are we going to deploy changes to the components? What is the potential traffic the application can receive? What is the traffic pattern on the application?
- Based on the answers you get to the preceding questions, you need to understand what sort of infrastructure you will run your application on. Will it be on-premises or the cloud, and will you use a managed Kubernetes cluster or self-host and manage one? You can also look at options such as CaaS for lightweight applications.
- How will you monitor and operate your containers? Will it require installing specialist tools? Will it require integrating with the existing monitoring tool stack? Understand the feasibility and make an appropriate design decision.
- How will you secure your containers? Are there any regulatory and compliance requirements regarding security? Does the chosen solution cater to them?

Containerizing the application

Containerizing an application involves packaging the application and its dependencies into a container image, which can be deployed and run consistently across different environments.

Containerizing an application offers benefits such as improved portability, scalability, and reproducibility. It simplifies the deployment process and allows for consistent application behavior across different environments.

Once we've considered all aspects of the design, we can now start containerizing the application:

- This is where we look into the application and create a Dockerfile containing the steps to create the container just as it is currently. This requires a lot of brainstorming and assessment, mostly if config management tools don't build your application by running on a virtual machine such as Ansible. It can take a long time to figure out how the application was installed, and you need to write the exact steps for this.
- If you plan to break your application into smaller parts, you may need to build your application from scratch.
- You must decide on a test suite that works on your parallel virtual machine-based application and improve it over time.

Testing

Testing containerized applications is an important step to ensure their functionality, performance, and compatibility.

By implementing a comprehensive testing strategy, you can ensure the reliability, performance, and security of your containerized application. Testing at various levels, integrating automation, and closely monitoring the application's behavior will help you identify and resolve issues early in the development life cycle, leading to a more robust and reliable containerized application.

Once we've containerized the application, the next step in the process is testing:

- To prove whether your containerized application works exactly like the one in the virtual machine, you need to do extensive testing to prove that you haven't missed any details or parts you should have considered previously. Run an existing test suite or the one you created for the container.
- Running an existing test suite can be the right approach, but you also need to consider the software's non-functional aspects. Benchmarking the original application is a good start, and you need to understand the overhead the container solution is putting in. You also need to fine-tune your application so that it fits the performance metrics.

- You also need to consider the importance of security and how you can bring it into the container world. Penetration testing will reveal a lot of security loopholes that you might not be aware of.

Deployment and rollout

Deploying and rolling out a containerized application involves deploying the container images to the target environment and making the application available for use.

Once we've tested our containers and are confident enough, we can roll out our application to production:

- Finally, we roll out our application to production and learn from there if further changes are needed. We then return to the discovery process until we have perfected our application.
- You must define and develop an automated runbook and a CI/CD pipeline to reduce cycle time and troubleshoot issues quickly.
- Doing A/B testing with the container applications running in parallel can help you realize any potential issues before you switch all the traffic to the new solution.

The following diagram summarizes these steps, and as you can see, this process is cyclic. This means that you may have to revisit these steps from time to time based on what you learned from the operating containers in production:

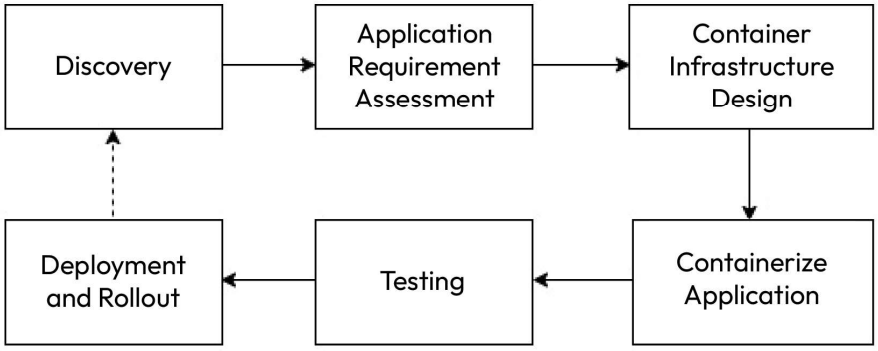

Figure 1.8 – Migrating from virtual machines to containers

Now, let's understand what we need to do to ensure that we migrate from virtual machines to containers with the least friction and also attain the best possible outcome.

What applications should go in containers?

In your journey of moving from virtual machines to containers, you first need to assess what can and can't go in containers. Broadly speaking, there are two kinds of application workloads you can have – **stateless** and **stateful**. While stateless workloads do not store state and are computing powerhouses, such as APIs and functions, stateful applications, such as databases, require persistent storage to function.

Though it is possible to containerize any application that can run on a Linux virtual machine, stateless applications become the first low-hanging fruits you may want to look at. It is relatively easy to containerize these workloads because they don't have storage dependencies. The more storage dependencies you have, the more complex your application becomes in containers.

Secondly, you also need to assess the form of infrastructure you want to host your applications on. For example, if you plan to run your entire tech stack on Kubernetes, you would like to avoid a heterogeneous environment wherever possible. In that scenario, you may also wish to containerize stateful applications. With web services and the middleware layer, most applications rely on some form of state to function correctly. So, in any case, you would end up managing storage.

Though this might open up Pandora's box, there is no standard agreement within the industry regarding containerizing databases. While some experts are naysayers for its use in production, a sizeable population sees no issues. The primary reason is insufficient data to support or disprove using a containerized database in production.

I suggest that you proceed with caution regarding databases. While I am not opposed to containerizing databases, you must consider various factors, such as allocating proper memory, CPU, disk, and every dependency you have on virtual machines. Also, it would help if you looked into the behavioral aspects of the team. If you have a team of DBAs managing the database within production, they might not be very comfortable dealing with another layer of complexity – containers.

We can summarize these high-level assessment steps using the following flowchart:

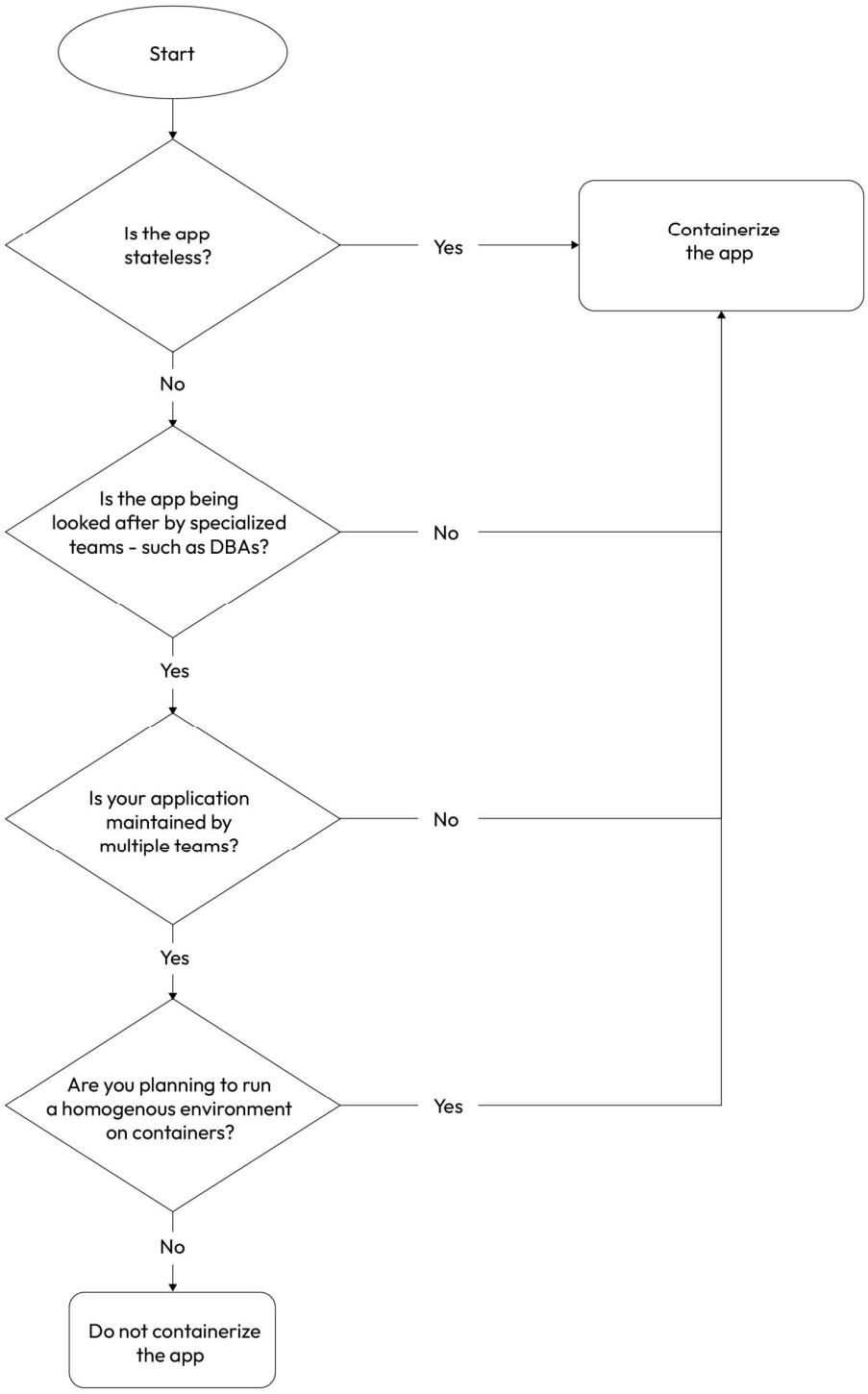

Figure 1.9 – Virtual machine to container migration assessment

This flowchart accounts for the most common factors that are considered during the assessment. You also need to factor in situations that are unique to your organization. So, it is a good idea to take those into account as well before making any decisions.

Let's look at some use cases that are suitable for containerization to get a fair understanding. The following types of applications are commonly deployed using containers:

- **Microservices architecture**: Applications that follow a microservices architecture, where the functionality is divided into small, independent services, are well-suited for containerization. Each microservice can be packaged as a separate container, enabling easier development, deployment, scaling, and management of the individual services.
- **Web applications**: Web applications, including frontend applications, backend APIs, and web services, can be containerized. Containers provide a consistent runtime environment, making it easier to package and deploy web applications across different environments, such as development, testing, and production.
- **Stateful applications**: Containers can also be used to run stateful applications that require persistent data storage. By leveraging container orchestration platforms' features, such as persistent volumes or stateful sets, stateful applications such as databases, content management systems, or file servers can be containerized and managed effectively.
- **Batch processing or scheduled jobs**: Applications that perform batch processing tasks or scheduled jobs, such as data processing, periodic backups, or report generation, can benefit from containerization. Containers provide a controlled and isolated environment for running these jobs, ensuring consistent execution and reproducibility.
- **CI/CD tools**: Containerizing CI/CD tools such as Jenkins, GitLab CI/CD, or CircleCI allows for consistent and reproducible build, test, and deployment pipelines. Containers make it easier to manage dependencies, isolate build environments, and enable rapid deployment of CI/CD infrastructure.
- **Development and testing environments**: Containers are valuable for creating isolated and reproducible development and testing environments. Developers can use containers to package their applications along with the required dependencies, libraries, and development tools. This enables consistent development and testing experiences across different machines and team members.
- **Internet of Things (IoT) applications**: Containers can be used to deploy and manage applications in IoT scenarios. They provide lightweight and portable runtime environments for IoT applications, enabling easy deployment across edge devices, gateways, or cloud infrastructures.
- **Machine learning and data analytics applications**: Containerization is increasingly used to deploy machine learning models and data science applications. Containers encapsulate the necessary dependencies, libraries, and runtime environments, allowing for seamless deployment and scaling of data-intensive applications.

It's important to note that not all applications are ideal candidates for containerization. Applications with heavy graphical interfaces, legacy monolithic architectures tightly coupled to the underlying infrastructure, or applications that require direct hardware access may not be suitable for containerization. Virtual machines or other deployment approaches may be more appropriate in such cases.

Breaking the applications into smaller pieces

You get the most out of containers if you run parts of your application independently of others.

This approach has numerous benefits, as follows:

- You can release your application more often as you can now change a part of your application without this impacting something else; your deployments will also take less time to run.
- Your application parts can scale independently of each other. For example, if you have a shopping app and your *orders* module is jam-packed, it can scale more than the *reviews* module, which may be far less busy. With a monolith, your entire application would scale with traffic, and this would not be the most optimized approach from a resource consumption point of view.
- Something that impacts one part of the application does not compromise your entire system. For example, customers can still add items to their cart and check out orders if the *reviews* module is down.

However, you should also not break your application into tiny components. This will result in considerable management overhead as you will not be able to distinguish between what is what. In terms of the shopping website example, it is OK to have an *order* container, a *reviews* container, a *shopping cart* container, and a *catalog* container. However, it is not OK to have *create order*, *delete order*, and *update order* containers. That would be overkill. Breaking your application into logical components that fit your business is the right way.

But should you break your application into smaller parts as the very first step? Well, it depends. Most people will want to get a **return on investment** (**ROI**) out of their containerization work. Suppose you do a lift and shift from virtual machines to containers, even though you are dealing with very few variables, and you can go into containers quickly. In that case, you don't get any benefits out of it – especially if your application is a massive monolith. Instead, you would add some application overhead because of the container layer. So, rearchitecting your application to fit in the container landscape is the key to going ahead.

Are we there yet?

So, you might be wondering, are we there yet? Not really! Virtual machines are to stay for a very long time. They have a good reason to exist, and while containers solve most problems, not everything can be containerized. Many legacy systems are running on virtual machines that cannot be migrated to containers.

With the advent of the cloud, *virtualized infrastructure* forms its base, and virtual machines are at its core. Most containers run on virtual machines within the cloud, and though you might be running containers in a cluster of nodes, these nodes would still be virtual machines.

However, the best thing about the container era is that it sees virtual machines as part of a standard setup. You install a container runtime on your virtual machines and do not need to distinguish between them. You can run your applications within containers on any virtual machine you wish. With a container orchestrator such as *Kubernetes*, you also benefit from the orchestrator deciding where to run the containers while considering various factors – resource availability is among the most critical.

This book will look at various aspects of modern DevOps practices, including managing cloud-based infrastructure, virtual machines, and containers. While we will mainly cover containers, we will also look at config management with equal importance using Ansible and learn how to spin up infrastructure with Terraform.

We will also look into modern CI/CD practices and learn how to deliver an application into production efficiently and error-free. For this, we will cover tools such as **Jenkins** and **Argo CD**. This book will give you everything you need to undertake a modern DevOps engineer role in the cloud and container era.

Summary

In this chapter, we understood modern DevOps, the cloud, and modern cloud-native applications. We then looked at how the software industry is quickly moving toward containers and how, with the cloud, it is becoming more critical for a modern DevOps engineer to have the required skills to deal with both. Then, we took a peek at the container architecture and discussed some high-level steps in moving from a virtual machine-based architecture to a containerized one.

In the next chapter, we will look at source code management with **Git**, which will form the base of everything we will do in the rest of this book.

Questions

Answer the following questions to test your knowledge of this chapter:

1. Cloud computing is more expensive than on-premises. (True/False)
2. Cloud computing requires more **Capital Expenditure** (**CapEx**) than **Operating Expenditure** (**OpEx**). (True/False)
3. Which of the following is true about cloud-native applications? (Choose three)

   A. They typically follow the microservices architecture

   B. They are typically monoliths

   C. They use containers

D. They use dynamic orchestration

E. They use on-premises databases

4. Containers need a hypervisor to run. (True/False)
5. Which of the following statements regarding containers is *not* correct? (Choose one)

   A. Containers are virtual machines within virtual machines

   B. Containers are simple OS processes

   C. Containers use cgroups to provide isolation

   D. Containers use a container runtime

   E. A container is an ephemeral workload

6. All applications can be containerized. (True/False)
7. Which of the following is a container runtime? (Choose two)

   A. Docker

   B. Kubernetes

   C. Containerd

   D. Docker Swarm

8. What kind of applications should you choose to containerize first?

   A. APIs

   B. Databases

   C. Mainframes

9. Containers follow CI/CD principles out of the box. (True/False)
10. Which of the following is an advantage of breaking your applications into multiple parts? (Choose four)

    A. Fault isolation

    B. Shorter release cycle time

    C. Independent, fine-grained scaling

    D. Application architecture simplicity

    E. Simpler infrastructure

11. While breaking an application into microservices, which aspect should you consider?

    A. Breaking applications into as many tiny components as possible

    B. Breaking applications into logical components

12. What kind of application should you containerize first?

    A. Stateless

    B. Stateful

13. Which of the following are examples of CaaS? (Choose three)

    A. Azure Functions

    B. Google Cloud Run

    C. Amazon ECS

    D. Azure ACS

    E. Oracle Functions

Answers

1. False
2. False
3. A, C, D
4. False
5. A
6. False
7. A, C
8. A
9. True
10. A, B, C, E
11. B
12. A
13. B, C, D

2

Source Code Management with Git and GitOps

In the previous chapter, we looked at the core concepts of modern DevOps, had an introduction to the cloud, and got a fair understanding of containers. In this chapter, we will understand source code management and one of the modern ways of enabling DevOps with **GitOps**.

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

- What is source code management?
- A crash course on Git
- What is GitOps?
- The principles of GitOps
- Why GitOps?
- Branching strategies and GitOps workflow
- Git versus GitOps

Technical requirements

To follow this chapter, you will need access to a Linux-based command line. If you are using macOS, you can use the inbuilt Terminal for all tasks. If you're a Windows user, you must install **GitBash** from `https://git-scm.com/download/win`. We will cover the installation instructions for this in the following sections.

Now, let's start by understanding source code management.

What is source code management?

Software development involves writing code. Code is the only tangible aspect of the software, allowing the software to function. Therefore, you need to store code somewhere to write and make changes to existing software. There are two kinds of code – **source code**, which is written in a high-level language, and **binaries**, which are compiled from the source code. Generally, binaries are nothing but functional applications that execute when we run the software, and source code is the human-readable code written to generate the binary, which is why source code is named as such.

A software development team has multiple members writing software features, so they must collaborate on code. They cannot just write code on silos without understanding how the application works. Sometimes, more than one developer works on the same feature, so they need some place to share their code with their peers. Source code is an asset in itself; therefore, we want to store it securely in a central location while still readily providing access to developers without hampering their work. You will also want to track changes and version them as you might want to know what caused a problem and immediately roll them back. You will also need to persist the history of code to understand what changes were made by whom, and you will want to have a mechanism for source code peer reviews.

As you can see, you would want to manage multiple aspects of source code, and therefore you would use a source code management tool to do so.

A source code management tool helps you manage all aspects of source code. It provides a central location to store your code, version changes and allows multiple developers to collaborate on the same source code. It also keeps a record of all changes through a version history and everything else that we've talked about before. Effective source code management practices improve collaboration; enable efficient development workflows; provide version control, repository management, branching and merging, change tracking, and auditing; and enhance the overall quality and maintainability of software projects. Some popular SCM tools are **Git**, **Subversion**, **Mercurial**, and **CVS**. However, the most popular and de facto standard for SCM is Git. So, let's go ahead and learn about it in the next section.

A crash course on Git

Git is the most popular source code management system available these days, and it has now become mandatory for all developers to learn Git, at least the basic stuff. In this crash course, we will learn about all basic Git operations and build from them in the subsequent chapters.

Git is a distributed version control system. This means that every Git repository is a copy of the original, and you can replicate that to a remote location if needed. In this chapter, we will create and initialize a local Git repository and then push the entire repository to a remote location.

A Git repository in a remote central location is also known as a **remote repository**. From this central repository, all developers sync changes in their local repository, similar to what's shown in the following diagram:

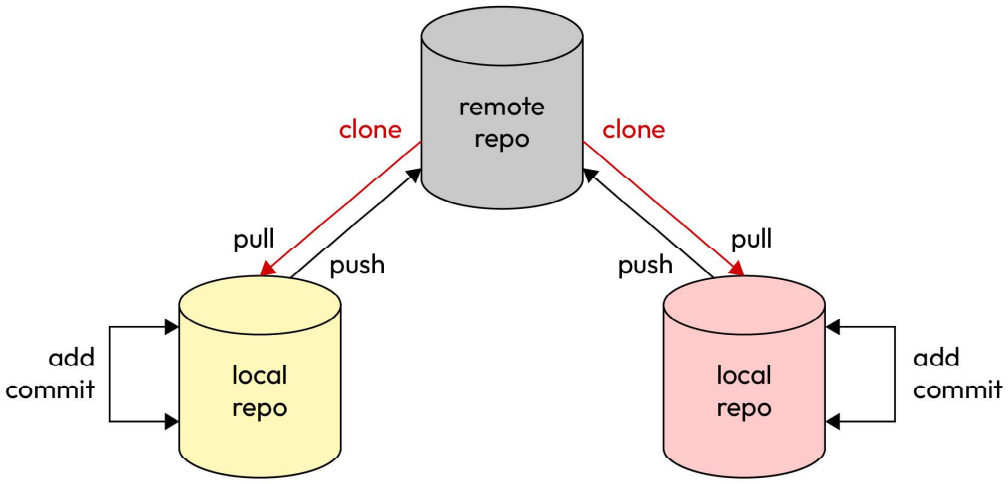

Figure 2.1 – Git distributed repository model

First, let's install Git locally and initialize a local repository. We will look at a remote repository later.

Installing Git

Depending on your platform and workstation, there are different ways to install Git. To install Git on **Ubuntu**, run the following command:

```
$ sudo apt install -y git-all
```

For other OSs and platforms, you can follow the steps at the following link: `https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`.

To check if Git has been installed successfully, run the following command:

```
$ git --version
git version 2.30.2
```

Now, let's initialize our first Git repository.

Initializing your first Git repository

To create a Git repository, you need to create a directory and run the `git init` command, as shown here:

```
$ mkdir first-git-repo && cd first-git-repo/
$ git init
Initialized empty Git repository in ~/first-git-repo/.git/
```

You are now ready to use your Git repository. You can also see that when you initialized the Git repository, Git created a hidden directory, `.git`, which it uses to keep track of all changes and commits. Whatever changes you make in your repo, Git keeps them as a delta of changes, which it depicts using + and - signs. We will look at these in detail in the subsequent sections. For now, let's create a new file within our Git repository and stage it for changes.

Staging code changes

Git allows developers to stage their changes before they commit them. This helps you prepare what you want to commit to the repository. The staging area is a temporary holding area for your changes, and you can add and remove files from the staging area by using the `git add` and `git restore` commands.

Let's create our first file within the local Git repository and stage the changes:

```
$ touch file1
```

Alternatively, you can create a blank file in the `first-git-repo` directory.

Now, we will check if Git can detect the new file that we've created. To do so, we need to run the following command:

```
$ git status
On branch master
No commits yet
Untracked files: (use "git add <file>..." to include in what will be committed)
    file1
nothing added to commit but untracked files present (use "git add" to track)
```

So, as we can see, Git has detected `file1` and is telling us that it is not tracking the file currently. To allow Git to track the file, let's run the following command:

```
$ git add file1
```

Now, let's run `git status` again to see what has changed:

```
$ git status
On branch master
No commits yet
Changes to be committed: (use "git rm --cached <file>..." to unstage)
    new file:   file1
```

As we can see, Git now shows `file1` as a new file in the staging area. You can continue making changes, and when you are done, you can commit the changes using the following command:

```
$ git commit -m "My first commit"
[master (root-commit) cecfb61] My first commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1
```

Git has now recorded a commit with your changes. Now, let's look at its status again using the following command:

```
$ git status
On branch master
nothing to commit, working tree clean
```

Git is now reporting that the working tree is clean, and there is nothing to commit. It also shows that there are no untracked files. Now, let's change `file1` and add some text to it:

```
$ echo "This is first line" >> file1
$ cat file1
This is first line
```

`file1` now contains the first line. Let's go ahead and commit this change:

```
$ git add file1
$ git commit -m "My second commit"
[master 4c55cf5] My second commit
 1 file changed, 1 insertion(+)
```

As we can see, Git is now reporting that one file has changed, and there is one insertion. Remember when we discussed that Git only tracks the delta changes between commits? That is what is happening here.

In the introduction, we mentioned that Git provides a history of all commits. Let's look at how we can display this history.

Displaying commit history

Git keeps a history of all commits. To see a list of all commits that you've done, you can use the following command:

```
$ git log
commit 275d24c62a0e946b8858f562607265c269ec5484 (HEAD -> master)
Author: Gaurav Agarwal <example@gmail.com>
Date:   Wed Apr 19 12:27:13 2023 +0530
    My second commit
commit cecfb61b251f9966f50a4d8bb49742b7af014da4
Author: Gaurav Agarwal < example@gmail.com>
Date:   Wed Apr 19 12:20:02 2023 +0530
    My first commit
```

As we can see, Git has displayed the history of both our commits. Notice that Git marks every commit with a commit ID. We can also delve into what changes were made in the commit by using the `git diff <first_commit_id> <second_commit_id>` command, as follows:

```
$ git diff cecfb61b251f9966f50a4d8bb49742b7af014da4 \
275d24c62a0e946b8858f562607265c269ec5484
diff --git a/file1 b/file1
```

```
index e69de29..0cbcf32 100644
--- a/file1
+++ b/file1
@@ -0,0 +1 @@
+This is first line
```

The output clearly shows that the second commit has added `This is first line` within `file1`.

You've suddenly realized that you needed to add another line to `file1` and wanted to do so in the same commit. We can do this by amending the commit. We'll look at this in the next section.

Amending the last commit

It is a best practice to have a single commit for your changes to a particular feature. This helps you track the changes better and makes it easier for the reviewer to review them. In turn, it is cleaner to visualize and manage. However, committing frequently is also a best practice so that your changes are not lost. Fortunately, Git allows you to add changes to the last commit.

To demonstrate this, let's change `file1` and add another line:

```
$ echo "This is second line" >> file1
$ cat file1
This is first line
This is second line
```

Now, let's add the changes to the previous commit using the following commands:

```
$ git add file1
$ git commit --amend
```

Once you run this command, Git will show you a prompt, allowing you to amend the commit message if you like. It will look something like the following:

```
My second commit
# Please enter the commit message for your changes. Lines
# starting with # will be ignored and an empty message aborts the commit
# Date Wed Apr 19 12:27:13 2023 +0530
# on branch master
# Changes to be commited
#   modified: file1
#
```

Save this file (use `ESC:wq` for Vim). This should amend the last commit with the changes. You should get the following output:

```
Date: Wed Apr 19 12:27:13 2023 +0530
 1 file changed, 2 insertions(+)
```

When Git amends a commit, you can no longer refer to the previous commit with the same commit ID. Instead, Git generates a separate `SHA-1 id` for the amended commit. So, let's look at the logs to see this for ourselves:

```
$ git log
commit d11c13974b679b1c45c8d718f01c9ef4e96767ab (HEAD -> master)
Author: Gaurav Agarwal <gaurav.agarwal@example.com>
Date:   Wed Apr 19 12:27:13 2023 +0530
    My second commit
commit cecfb61b251f9966f50a4d8bb49742b7af014da4
Author: Gaurav Agarwal < example@gmail.com>
Date:   Wed Apr 19 12:20:02 2023 +0530
    My first commit
```

Now, let's run the `diff` command again and see what it is reporting:

```
$ git diff cecfb61b251f9966f50a4d8bb49742b7af014da4 \
d11c13974b679b1c45c8d718f01c9ef4e96767ab
diff --git a/file1 b/file1
index e69de29..655a706 100644
--- a/file1
+++ b/file1
@@ -0,0 +1,2 @@
+This is first line
+This is second line
```

The output clearly shows that the second commit has added `This is first line`, as well as `This is second line`, within `file1`. With that, you've successfully amended a commit.

Local repositories are as good as keeping files on your system. However, since you need to share your code with others and keep it secure from laptop OS crashes, theft, physical damage, and more, you need to push your code into a remote repository. We'll look at remote repositories in the next section.

Understanding remote repositories

Remote repositories are replicas of the Git repository at a central location for multiple people to access. This allows your developers to work on the same code base simultaneously and provides you with a backup of your code. There are various tools you can use to host your remote repositories. Notable ones include **GitHub**, **Bitbucket**, and **Gerrit**. You can install them on your on-premises or cloud servers or use a **Software-as-a-Service** (**SaaS**) platform to store them online. In this book, we are going to focus on GitHub.

GitHub is a web-based platform that helps developers collaborate on code. It is based on Git and allows you to host remote Git repositories. It was founded in 2008 and was acquired by Microsoft in 2018. It is one of the most popular open-source SaaS-based Git repository services and contains almost all open-source code available worldwide.

Before we can create our first remote repo, we must go to `https://github.com/signup` to create an account.

Once you've created an account, we can go ahead and create our first remote Git repository.

Creating a remote Git repository

Creating a remote Git repository is simple on GitHub. Go to `https://github.com/new`, set **Repository Name** to `first-git-repo`, keep the rest of the fields as-is, and click the **Create Repository** button.

Once you've done that, GitHub will provide you with some steps that you can follow to connect with your remote repository. Before we go into any of that, we want to configure some authentication for our local Git command line to interact with the remote repository. Let's take a look.

Setting up authentication with the remote Git repository

Some of the ways you can authenticate with your remote Git repository are as follows:

- **HTTPS**: In this mode, Git uses HTTPS to connect with the remote Git repository. We need to create an HTTPS token within our GitHub account and use this token as a password to authenticate with the remote repository. This process requires you to key in your token every time you authenticate with Git; therefore, it is not a convenient option.
- **SSH**: In this mode, Git uses the SSH protocol to connect with the remote Git repository. While using SSH, we do not need to use a password to authenticate; instead, we must add the **public key** of an **SSH key pair** we can generate from the Linux (or Windows if you're using Git Bash) command line to the GitHub account. This process is more secure as well as convenient.

So, let's set up SSH-based authentication with our remote Git repository.

First, we must generate the SSH key pair within our local system. Go to your Terminal and run the following command to generate an SSH key pair:

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
```

You will be prompted for other details. Keep pressing *Enter* until you reach the prompt again.

Once the key pair has been generated, copy the public key present in the `~/.ssh/id_rsa.pub` file.

Then, go to `https://github.com/settings/ssh/new`, paste the public key in the **Key** field, and click the **Add SSH Key** button. We are now ready to connect with the remote Git repository. Now, let's look at the configurations we must do on our local repository to connect with the remote repository.

Connecting the local repository to the remote repository

You will need to add a remote entry using the following command to connect with the remote repository from the local repository:

```
$ git remote add origin git@github.com:<your-github-username>/first-git-repo.git
```

You can also find these details on the **Quick Setup** page of your GitHub repository.

Now that we've set up the connection, let's look at how we can push our changes to the remote repository.

Pushing changes from the local repository to the remote repository

To push the changes from the local repository to the remote repository, use the following command:

```
$ git push -u origin master
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 474 bytes | 474.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com: <your-github-username>/first-git-repo.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Now, refresh the page on your remote repository. You should see that the code was synced, as shown in the following screenshot:

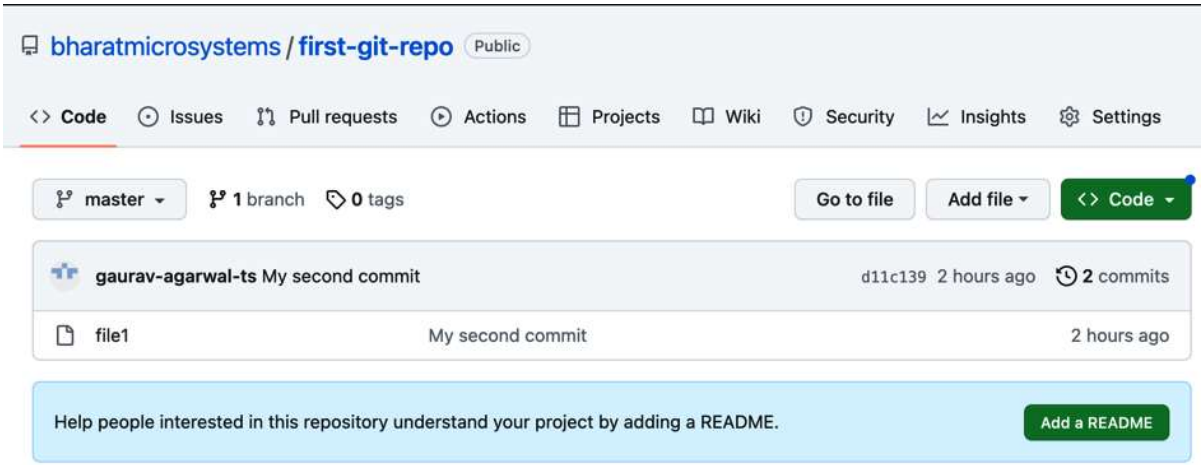

Figure 2.2 – Code synced in the remote repository

You can also use the inline editor to make further changes to the file using the GitHub web portal. While this is not recommended, we'll do this to simulate a situation where another developer changed the same file you were working on.

Click on **file1** and then click on the **pencil** icon to edit the file, as shown in the following screenshot:

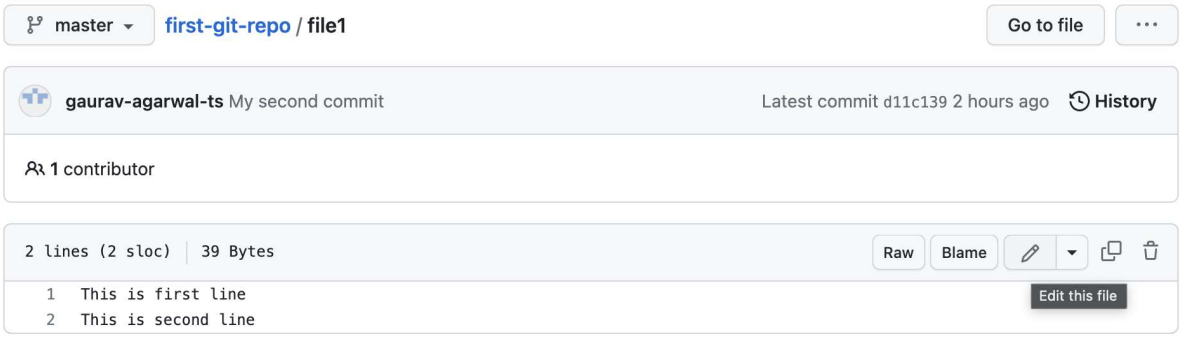

Figure 2.3 – Editing the file in the remote repository

Upon doing this, an editing window will open where you can make changes to the file. Let's add `This is third line` within the file, as shown in the following screenshot:

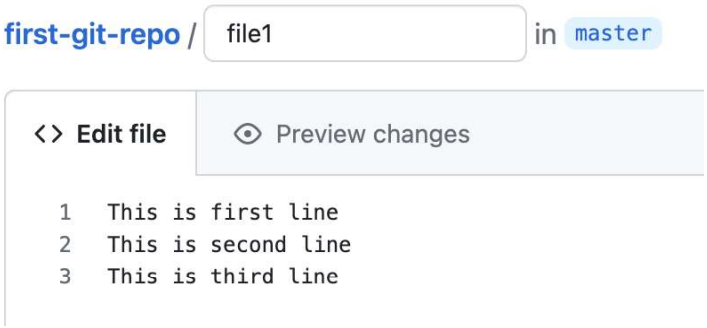

Figure 2.4 – Adding a new line

Scroll down – you should see a **Commit changes** section, where you can add a commit message field and click on the **Commit** button, as shown in the following screenshot:

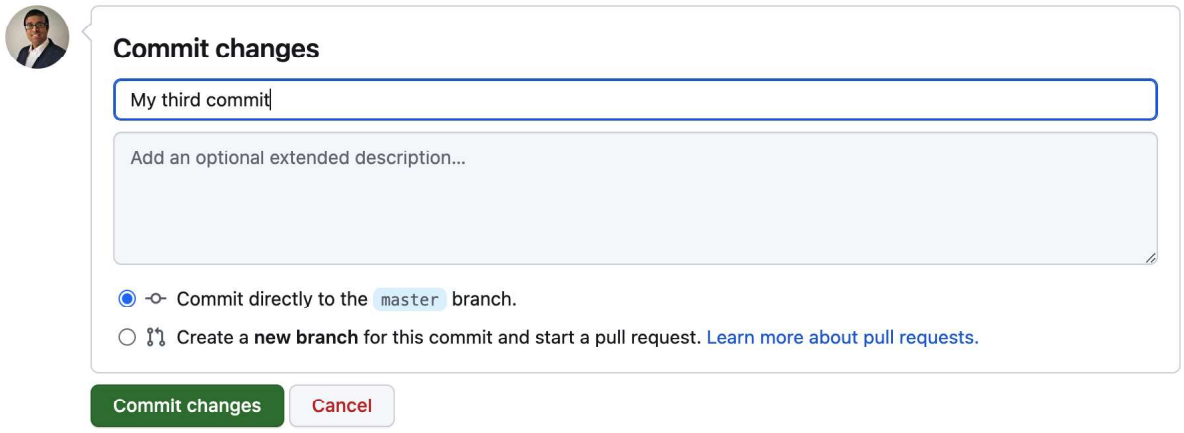

Figure 2.5 – The Commit changes section

Once you've clicked on that button, you should see the third line, as shown in the following screenshot:

Figure 2.6 – Changes committed remotely

At this point, changes have been made to the remote repository, but you have also been working on your changes. To simulate this, let's make a change in the local repository as well using the following commands:

```
$ echo "This is fourth line" >> file1
$ cat file1
This is first line
This is second line
This is fourth line
$ git add file1
$ git commit -m "Added fourth line"
[master e411e91] Added fourth line
 1 file changed, 1 insertion(+)
```

Now that we've committed the changes in our local repository, let's try to push the code to the remote repository using the following commands:

```
$ git push
To github.com:<your-github-username>/first-git-repo.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'github.com:<your-github-username>/first-git-repo.git'
hint: Updates were rejected because the remote contains work that you do not have locally.
This is usually caused by another repository pushing to the same ref. You may want to
first integrate the remote changes.
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Wait, what happened? Well, the remote repository rejected the changes as we tried to push changes while someone else made some commits in the remote repository, and our changes are not current. We would need to pull the changes in our local repository first to apply our changes on top of the existing ones in the remote repository. We'll look at this in the next section.

Pulling and rebasing your code

Pulling code involves downloading up-to-date code from the remote to your local repository. **Rebasing** means applying your changes on top of the latest remote commit. It is a best practice to pull and rebase your changes on top of what already exists in the remote repository.

Let's do so using the following command:

```
$ git pull --rebase
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0),pack-reused 0
Unpacking objects: 100% (3/3), 652 bytes | 130.00 KiB/s, done.
From github.com:<your-github-username>/first-git-repo
   d11c139..f5b7620  master     -> origin/master
Auto-merging file1
CONFLICT (content): Merge conflict in file1
error: could not apply e411e91... Added fourth line
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply e411e91... Added fourth line
```

Now, we have another issue: we are unable to rebase our commit as we've tried to modify a file that has been modified by someone else. Git wants us to check the file and make appropriate changes so that the changes are applied correctly. This situation is known as a **merge conflict**. Git also provides us with the file that contains the conflict. Let's open the file with a text editor and make the appropriate changes.

The current file looks like this:

```
This is first line
This is second line
<<<<<<< HEAD
This is third line
=======
This is fourth line
>>>>>>> e411e91 (Added fourth line)
```

The portion depicted by `HEAD` is the line in the remote repository and shows the recent changes made remotely. The `e411e91` commit shows the changes that we made locally. Let's change the file to the following and save it:

```
This is first line
This is second line
This is third line
This is fourth line
```

Now, let's add the file to the staging area and continue the rebase using the following commands:

```
$ git add file1
$ git rebase --continue
[detached HEAD 17a0242] Added fourth line
 1 file changed, 1 insertion(+)
Successfully rebased and updated refs/heads/master.
```

Now that we've rebased the changes, let's look at the status of the Git repo by running the following command:

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
nothing to commit, working tree clean
```

As we can see, we've added a single commit that we need to push to the remote repository. Let's do that now using the following command:

```
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 297 bytes | 148.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:<your-github-username>/first-git-repo.git
   f5b7620..17a0242  master -> master
```

And this time, the push worked successfully.

In most cases, you would normally need to take a copy of the main code and make changes to it to test new features. You might also want someone to review your changes before they are merged into the main code base. Git allows you to manage that by using Git branches. We'll look at Git branches in the next section.

Git branches

A **Git branch** is a copy of the code base (from where the branch is created) that you can independently modify and work on without affecting the main code base. You will want to create branches while working on new features to ensure that you are not affecting the main branch, which contains reviewed code. Most technology companies normally have several environments where you have code deployed in various stages. For example, you might have a **development environment** where you test your features, a **staging environment** where you integrate all features and test the complete application, and a **production environment** where the application that your end users access resides. So, there would be a possibility that you might have additional environment-specific branches where code deployed on those branches reside. In the following sections of this chapter, we will talk about GitOps, which works on this fundamental principle. For now, let's look at how we can create and manage Git branches.

Creating and managing Git branches

To create a Git branch, you must be on the branch from where you want to branch your code. As in our example repo, we were working on the master branch. Let's stay there and create a feature branch out of that.

To create the branch, run the following command:

```
$ git branch feature/feature1
```

As we can see, the feature branch has been created. To check what branch we are on now, we can use the `git branch` command. Let's do that now:

```
$ git branch
```

And as we see by the `*` sign over the `master` branch, we are still on the `master` branch. The good thing is that it also shows the `feature/feature1` branch in the list. Let's switch to the feature branch now by using the following command:

```
$ git checkout feature/feature1
Switched to branch 'feature/feature1'
```

Now, we are on the `feature/feature1` branch. Let's make some changes to the `feature/feature1` branch and commit it to the local repo:

```
$ echo "This is feature 1" >> file1
$ git add file1
```

```
$ git commit -m "Feature 1"
[feature/feature1 3fa47e8] Feature 1
 1 file changed, 1 insertion(+)
```

As we can see, the code is now committed to the `feature/feature1` branch. To check the version history, let's run the following command:

```
$ git log
commit 3fa47e8595328eca0bc7d2ae45b3de8d9fd7487c (HEAD -> feature/feature1)
Author: Gaurav Agarwal <gaurav.agarwal@example.com>
Date:   Fri Apr 21 11:13:20 2023 +0530
    Feature 1
commit 17a02424d2b2f945b479ab8ba028f3b535f03575 (origin/master, master)
Author: Gaurav Agarwal <gaurav.agarwal@example.com>
Date:   Wed Apr 19 15:35:56 2023 +0530
    Added fourth line
```

As we can see, the `Feature 1` commit is shown in the Git logs. Now, let's switch to the `master` branch and run the same command again:

```
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
$ git log
commit 17a02424d2b2f945b479ab8ba028f3b535f03575 (HEAD -> master, origin/master)
Author: Gaurav Agarwal <gaurav.agarwal@example.com>
Date:   Wed Apr 19 15:35:56 2023 +0530
    Added fourth line
commit f5b7620e522c31821a8659b8857e6fe04c2f2355
Author: Gaurav Agarwal <<your-github-username>@gmail.com>
Date:   Wed Apr 19 15:29:18 2023 +0530
    My third commit
```

As we can see here, the `Feature 1` commit changes are absent. This shows that both branches are now isolated (and have now diverged). Now, the changes are locally present and are not in the remote repository yet. To push the changes to the remote repository, we will switch to the `feature/feature1` branch again. Let's do that with the following command:

```
$ git checkout feature/feature1
Switched to branch 'feature/feature1'
```

Now that we've switched to the feature branch, let's push the branch to the remote repository using the following command:

```
$ git push -u origin feature/feature1
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 286 bytes | 286.00 KiB/s, done.
```

```
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'feature/feature1' on GitHub by visiting:
remote:      https://github.com/<your-github-username>/first-git-repo/pull/new/feature/
feature1
remote:
To github.com:<your-github-username>/first-git-repo.git
 * [new branch]      feature/feature1 -> feature/feature1
Branch 'feature/feature1' set up to track remote branch 'feature/feature1' from 'origin'.
```

With that, we've successfully pushed the new branch to the remote repository. Assuming the feature is ready, we want the changes to go into the master branch. For that, we would have to raise a pull request. We'll look at pull requests in the next section.

Working with pull requests

A **pull request** is a request for merging a **source branch** to a **base branch**. The base branch is the branch where the reviewed code resides. In this case, our base branch is `master`. Pull requests are generally useful for developers to get their code peer reviewed before they merge it with the *fair* version of the code. The reviewer generally checks the quality of the code, whether best practices are being followed, and whether coding standards are appropriate. If the reviewer is unhappy, they might want to flag certain sections of the changes and request modifications. There are normally multiple cycles of reviews, changes, and re-reviews. Once the reviewer is happy with the changes, they can approve the pull request, and the requester can merge the code. Let's take a look at this process:

1. Let's try to raise a pull request for merging our code from the `feature/feature1` branch to the `master` branch. To do so, go to your GitHub repo, select **Pull requests**, and click on the **New pull request** button, as shown in the following screenshot:

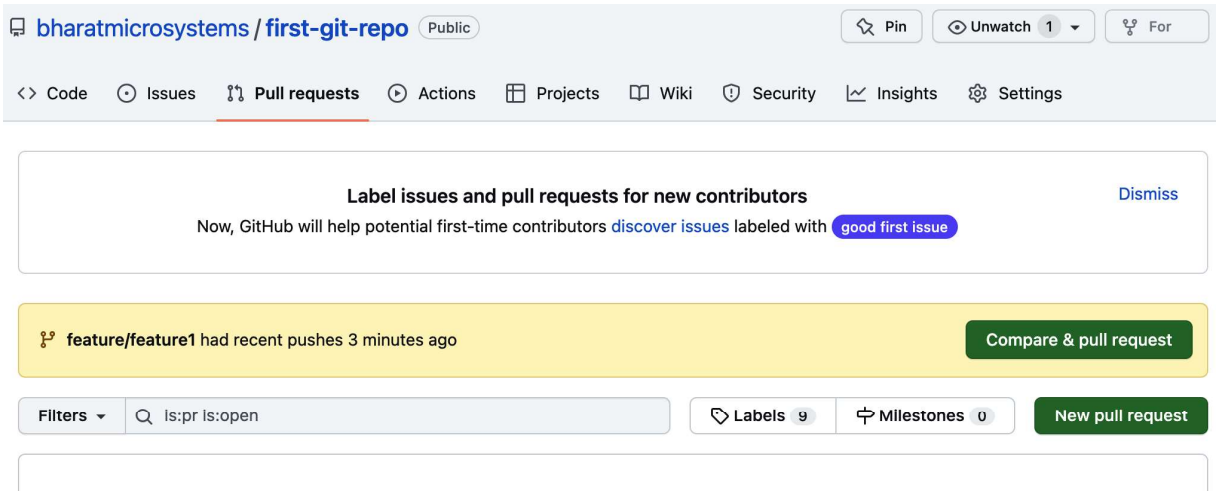

Figure 2.7 – New pull request

2. Keep **base** set to `master` and, in the **compare** dropdown, select `feature/feature1`:

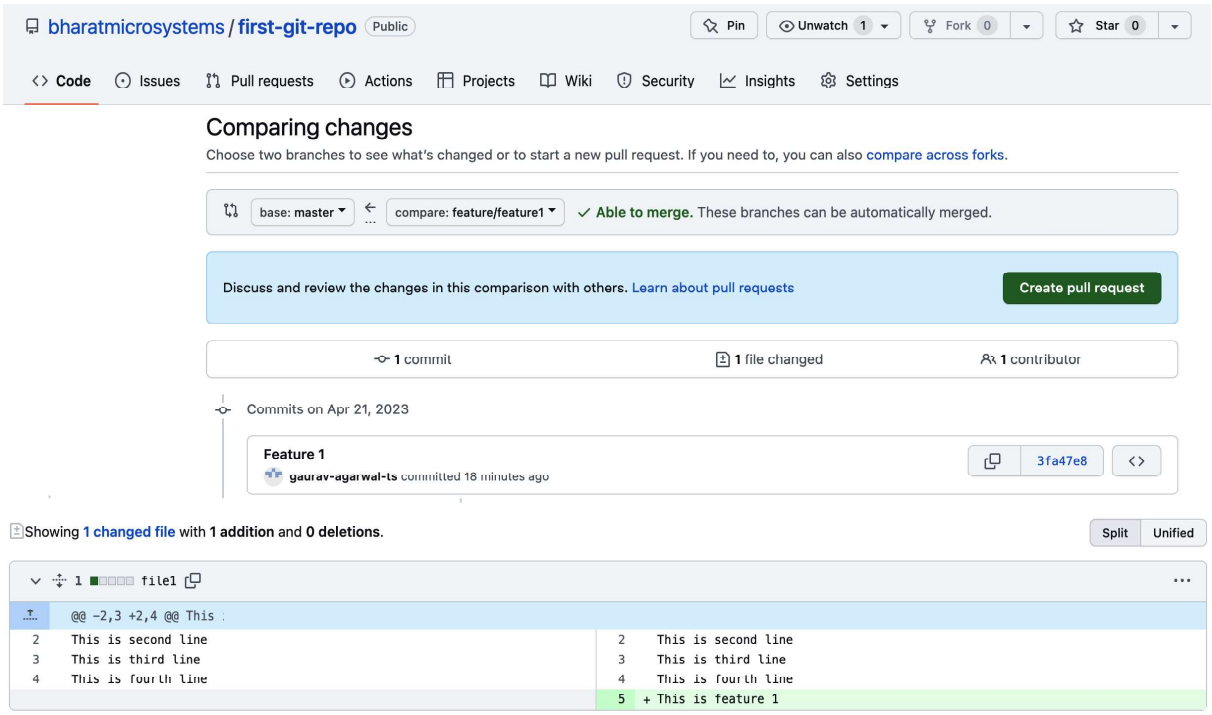

Figure 2.8 – Comparing changes

3. As you can see, it shows you all the changes we've made on the `feature/feature1` branch. Click on the **Create pull request** button to create the pull request. On the next page, stick to the defaults and click on the **Create pull request** button:

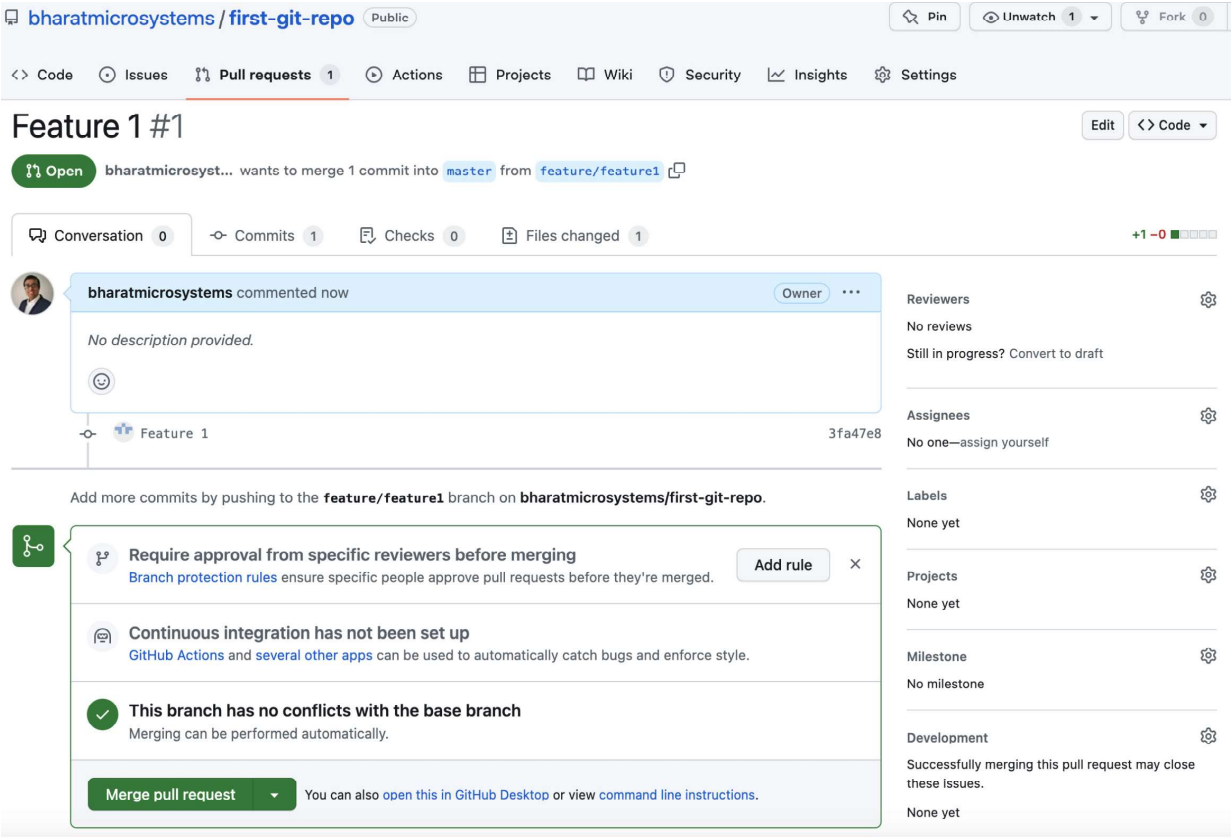

Figure 2.9 – Pull request created

4. As you can see, the pull request was created successfully. Here, you can assign a reviewer and get the code reviewed. Once the reviewer approves the code, you can merge the code to the master branch. For now, let's click on the **Merge pull request** button, followed by the **Confirm merge** button, which should merge the pull request.
5. Now, check if the `master` branch shows the changes within GitHub. If it does, switch to the `master` branch and pull the changes into your local repository. You should see the changes in your local repository as well.

I leave this to you as an exercise.

This was a crash course on Git to help you get started. Now, let's move on and understand GitOps, which uses Git as a single source of truth to spin up virtually anything within your application and infrastructure.

What is GitOps?

GitOps is a method that involves implementing DevOps so that Git forms the single source of truth. Instead of maintaining a long list of scripts and tooling to support this, GitOps focuses on writing declarative code for everything, including the infrastructure, configuration, and application code. This means you can spin anything out of thin air by simply using the Git repository. The idea is that you declare what you need in your Git repository, and there is tooling behind the scenes that ensures the desired state is always maintained in the running application and infrastructure surrounding it. The code to spin up the tooling also resides in Git, and you don't have anything outside of Git. This means everything, including the tooling, is automated in this process.

While GitOps also enables DevOps within the organization, it primarily focuses on using Git to manage infrastructure provisioning and application software deployments. DevOps is a broad term that contains a set of principles, processes, and tools to enable developers and operations teams to work seamlessly and shorten the development life cycle, with an end goal to deliver better software more quickly using a CI/CD cycle. While GitOps relies heavily on Git and its features and always looks to Git for versioning, finding configuration drift, and only applying deltas, DevOps is, as such, agnostic of any tool and focuses more on the concepts and processes. Therefore, you can implement DevOps without using Git, but you cannot implement GitOps without Git. Put simply, GitOps implements DevOps, but the reverse may not always be true.

Why GitOps?

GitOps provides us with the following benefits:

- **It deploys better software more quickly**: GitOps offers simplicity in delivering software. You don't have to worry about what tool you need for the deployment type. Instead, you can commit your changes in Git, and the behind-the-scenes tooling automatically deploys it.
- **It provides faster recovery from errors**: If you happen to make an error in deployment (for example, a wrong commit), you can easily roll it back using `git revert` and restore your environment. The idea is that you don't need to learn anything else apart from Git to do a rollout or a rollback.
- **It offers better credential management**: With GitOps, you don't need to store your credentials in different places for your deployments to work. You simply need to provide the tooling access to your Git repository and the binary repository, and GitOps will take care of the rest. You can keep your environment completely secure by restricting your developers' access to it and providing them access to Git instead.
- **Deployments are self-documenting**: Because everything is kept within Git, which records all commits, the deployments are automatically self-documenting. You can know exactly who deployed what at what time by simply looking at the commit history.

- **It promotes shared ownership and knowledge**: As Git forms the single source of truth for all code and configurations within the organization, teams have a single place to understand how things are implemented without ambiguity and dependency on other team members. This helps promote the shared ownership of the code and knowledge within the team.

Now that we know about the benefits of GitOps, let's look at its key principles.

The principles of GitOps

GitOps has the following key principles:

- **It describes the entire system declaratively**: Having declarative code forms the first principle of GitOps. This means that instead of providing instructions on how to build your infrastructure, applying the relevant configuration, and deploying your application, you declare the end state of what you need. This means that your Git repository always maintains a single source of truth. As declarative changes are idempotent, you don't need to worry about the state of your system as this will eventually become consistent with the code in Git.
- **It versions desired system state using Git**: As Git forms an excellent version control system, you don't need to worry too much about how to roll out and roll back your deployments. A simple Git commit means a new deployment, and a Git revert means a rollback. This means you do not need to worry about anything apart from ensuring that the Git repository reflects what you need.
- **It uses tooling to automatically apply approved changes**: As you've stored everything within Git, you can then use tooling that looks for changes within the repository and automatically applies them to your environment. You can also have several branches that apply changes to different environments, along with a pull request-based approval and gating process so that only approved changes end up in your environment.
- **It uses self-healing agents to alert and correct any divergence**: We have the tooling to automatically apply any changes in Git to the environment. However, we also require self-healing agents to alert us of any divergence from the repository. For example, suppose someone deletes a container manually from the environment but doesn't remove it from the Git repository. In that scenario, the agent should alert the team and recreate the container to correct the divergence. This means there is no way to bypass GitOps, and Git remains the single source of truth.

Implementing and living by these principles is simple with modern DevOps tools and techniques, and we will look at practically implementing them later in *Chapters 11* and *12*. In this chapter, however, we'll examine their design principles using a branching strategy and GitOps workflow.

Branching strategies and the GitOps workflow

GitOps requires at least two kinds of Git repositories to function: the **application repository**, which is from where your builds are triggered, and the **environment repository**, which contains all of the infrastructure and **configuration as code** (**CaC**). All deployments are driven from the environment repository, and the changes to the code repository drive the deployments. GitOps follows two primary kinds of deployment models: the **push model** and the **pull model**. Let's discuss each of them.

The push model

The push model pushes any changes that occur within your Git repository to the environment. The following diagram explains this process in detail:

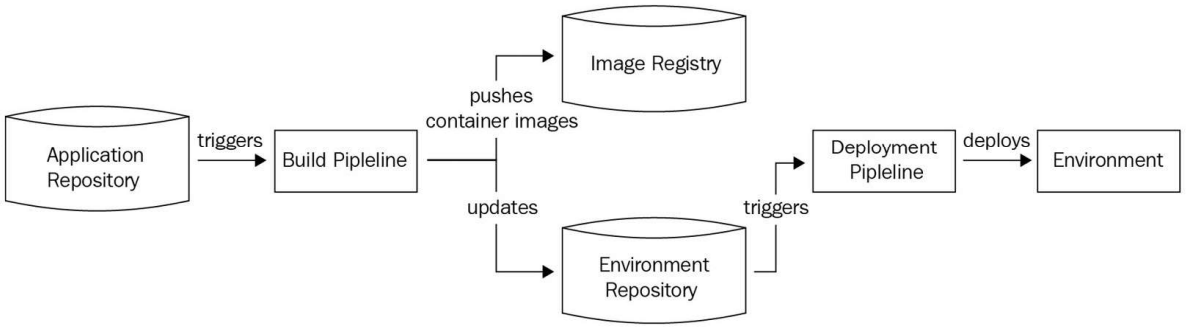

Figure 2.10 – The push model

The push model is inherently unaware of the existing configuration and reacts only to changes made to your Git repositories. Therefore, you will need to set up some form of monitoring to understand whether there are any deviations. Additionally, the push model needs to store all environment credentials within the tools. This is because it interacts with the environment and has to manage the deployments. Typically, we use **Jenkins**, **CircleCI**, or **Travis CI** to implement the push model. While the push model is not recommended, it becomes inevitable in cloud provisioning with **Terraform**, or config management with **Ansible**, as they are both push-based models. Now, let's take a closer look at the pull model.

The pull model

The pull model is an *agent-based deployment model* (also known as an *operator-based deployment model*). An *agent* (or *operator*) within your environment monitors the Git repository for changes and applies them as and when needed. The operator constantly compares the existing configuration with the configuration in the environment repository and applies changes if required. The following diagram shows this process in detail:

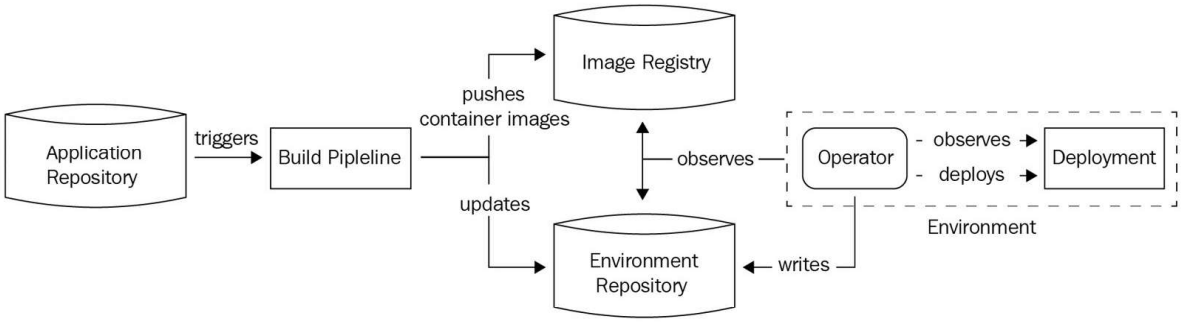

Figure 2.11 – The pull model

The advantage of the pull model is that it monitors and reacts to environment changes alongside repository changes. This ensures that any changes that do not match the Git repository are reverted from the environment. It also alerts the operations team about anything it could not fix using mail notifications, ticketing tools, or Slack notifications. Because the operator lives within the same environment where the code is deployed, we do not need to store credentials within the tools. Instead, they live securely within the environment. You can also live without storing any credentials at all with tools such as Kubernetes, where you can employ **role-based access control** (**RBAC**) and service accounts for the operator managing the environment.

> **Tip**
>
> When choosing a GitOps model, the best practice is to check whether you can implement a pull-based model instead of a push-based model. Implement a push-based model only if a pull-based model is not possible. It is also a good idea to implement polling in the push-based model by scheduling something, such as a `cron` job, that will run the push periodically to ensure there is no configuration drift.

We cannot solely live with one model or the other, so most organizations employ a **hybrid model** to run GitOps. This hybrid model combines push and pull models and focuses on using the pull model. It uses the push model when it cannot use the pull model. Now, let's understand how to structure our Git repository so that it can implement GitOps.

Structuring the Git repository

To implement GitOps, we require at least two repositories: the **application repository and the environment repository**. This does not mean that you cannot combine the two, but for the sake of simplicity, let's take a look at each of them separately.

The application repository

The application repository stores the application code. It is a repository in which your developers can actively develop the product that you run for your business. Typically, your builds result from

this application code, and they end up as containers (if we use a container-based approach). Your application repository may or may not have environment-specific branches. Most organizations keep the application repository independent of the environment and focus on building semantic code versions using a branching strategy. Now, there are multiple branching strategies available to manage your code, such as **Gitflow**, **GitHub flow**, and any other branching strategy that suits your needs.

Gitflow is one of the most popular branching strategies that organizations use. That said, it is also one of the most complicated ones as it requires several kinds of branches (for instance, master, hotfixes, release branches, develop, and feature branches) and has a rigid structure. The structure of Gitflow is shown in the following diagram:

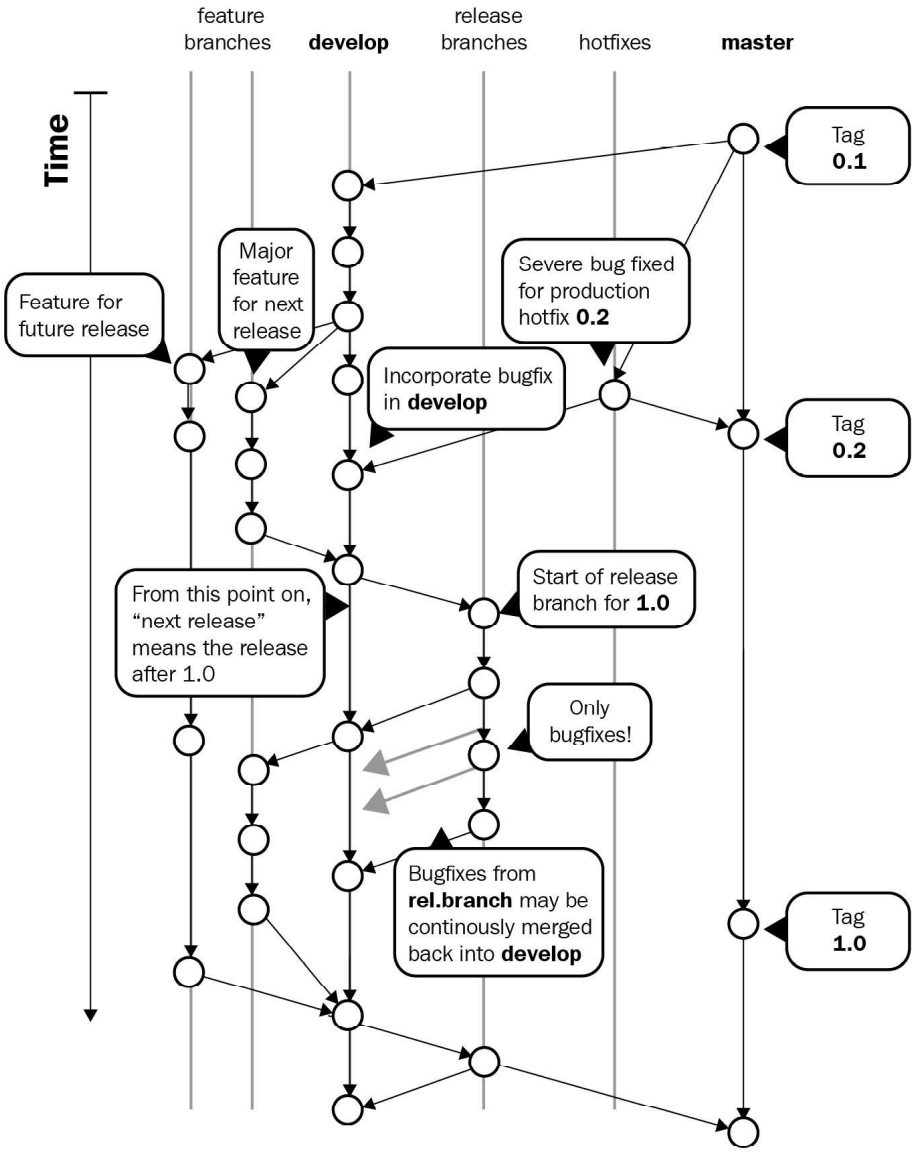

Figure 2.12 – Gitflow structure

A simplified way of doing things is using GitHub flow. It employs fewer branches and is easier to maintain. Typically, it contains a single master branch and many feature branches that eventually merge with the master branch. The master branch always has software that is ready to be deployed to the environments. You tag and version the code in the master branch, pick and deploy it, test it, and then promote it to higher environments. The following diagram shows GitHub flow in detail:

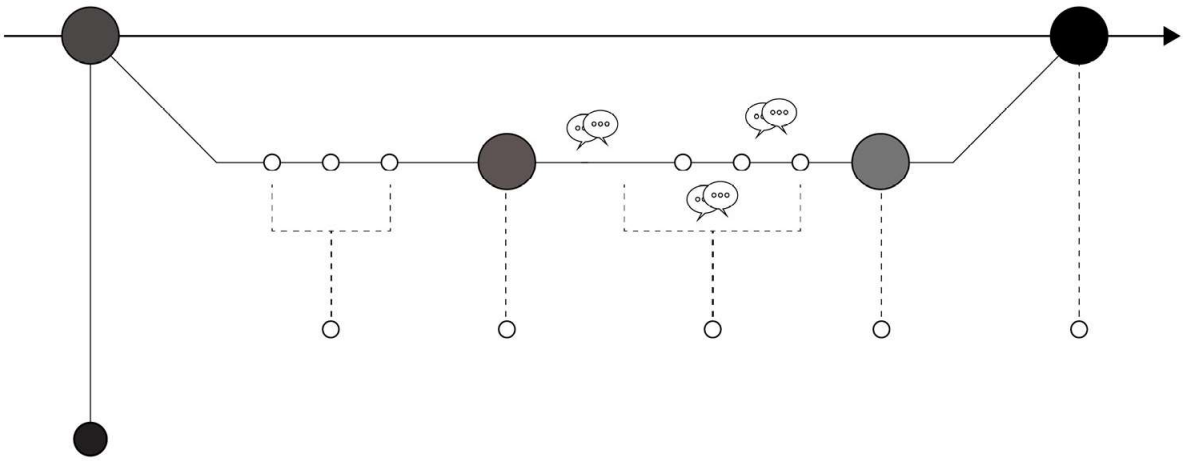

Figure 2.13 – GitHub flow

Note that you are free to create your branching strategy according to your needs and what works for you.

> **Tip**
> Choose Gitflow if you have a large team, a vast monolithic repository, and multiple releases running in parallel. Choose GitHub flow if you work for a fast-paced organization that releases updates several times a week and doesn't use the concept of parallel releases. GitHub flow also typically works for microservices where changes are minor and quick.

Typically, application repositories do not have to worry too much about environments; they can focus more on creating deployable software versions.

The environment repository

The environment repository stores the environment-specific configurations needed to run the application code. Therefore, they will typically have **Infrastructure as Code** (**IaC**) in the form of Terraform scripts, CaC in the form of Ansible playbooks, or Kubernetes manifests that typically help deploy the code we've built from the application repository.

The environment repository should follow an environment-specific branching strategy where a branch represents a particular environment. You can have pull request-based **gating** for these kinds of scenarios. Typically, you build your **development environments** from a development branch and then raise a

pull request to merge the changes to a staging branch. From the staging branch to production, your code progresses with environments. If you have 10 environments, you might end up with 10 different branches in the environment repository. The following diagram showcases the branching strategy you might want to follow for your environment repository:

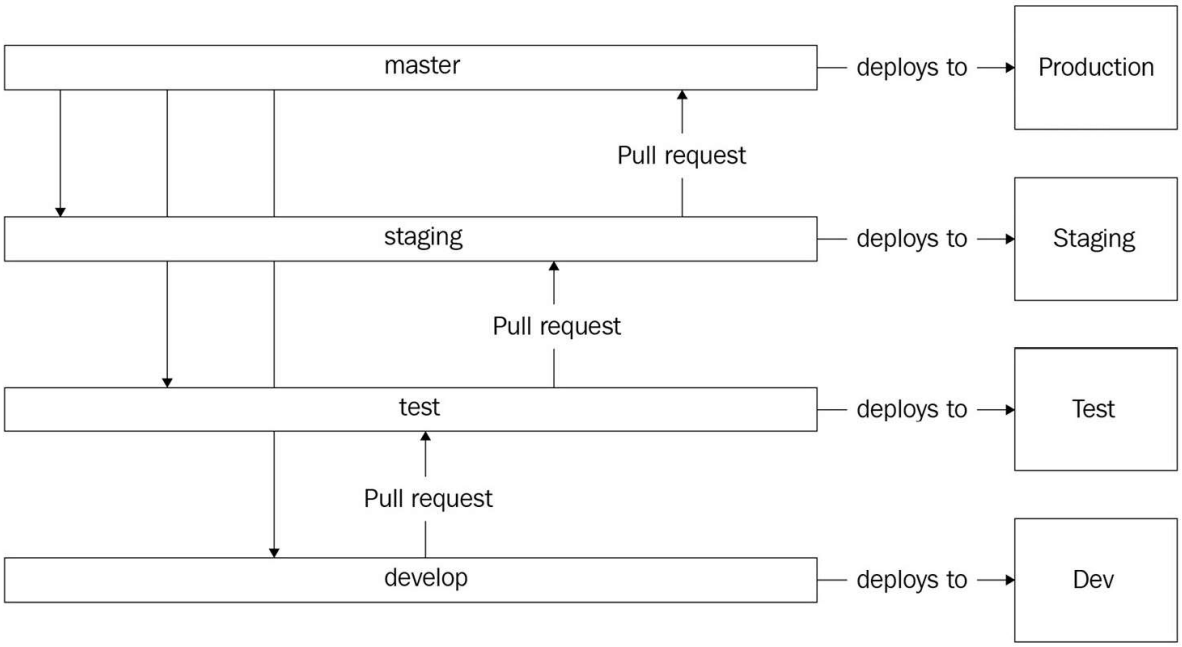

Figure 2.14 – The environment repository

The environment repository aims to act as the single source of truth for your environments. The configuration you add to the repository is applied directly to your environments.

> **Tip**
> While you can combine the environment and application repository into one, the best practice is to keep them separate. GitOps offers a clear separation between the CI and CD processes using the application and environment repositories, respectively.

Now that we've covered Git and GitOps in detail, let's look at why Git and GitOps are related but different concepts.

Git versus GitOps

The following table summarizes the differences between Git and GitOps:

	Git	GitOps
Definition	Git is a distributed version control system that tracks changes to source code and other files. It allows multiple developers to collaborate and manage code revisions efficiently.	GitOps is a set of practices and principles that leverage Git as the single source of truth for managing and automating the deployment and operation of infrastructure and applications.
Focus	Primarily focused on version control and collaboration for source code.	Focused on automating and managing the deployment and operation of infrastructure and applications through Git-based DevOps workflows.
Usage	Widely used for version control and collaboration in software development projects. Developers use Git to track changes, manage branches, and merge code.	Used for declaratively defining and managing infrastructure and application configurations. Git repositories serve as a central hub for defining desired states and driving automation.
Core Components	Repositories, branches, commits, and pull requests.	Git repositories, declarative configuration files (such as YAML), Kubernetes manifests, CI/CD pipelines, and deployment tools such as Argo CD or Flux.
Workflow	Developers clone, modify, commit, and push changes to a remote repository. They collaborate through pull requests and branch merges.	Infrastructure and application configurations are stored in Git repositories. Changes to these configurations trigger automated processes, such as CI/CD pipelines or reconciliation loops, to apply those changes to the target environment.
Benefits	Enables efficient version control, collaboration, and code management for software development teams.	Promotes infrastructure and application as code, versioning of configurations, and declarative management. It simplifies infrastructure deployment, provides consistency, and enables automated workflows.
Focus Area	Source code management.	Infrastructure and application deployment and management.
Examples	GitHub, Bitbucket, GitLab.	Argo CD, Flux, Jenkins X, Weave Flux.

Remember that while Git is a version control system, GitOps extends this concept by utilizing Git as a central source of truth for infrastructure and application configurations, allowing for automated deployment and management of DevOps workflows.

Summary

This chapter covered Git, GitOps, why we need it, its principles, and various GitOps deployments. We also looked at different kinds of repositories that we can create to implement GitOps, along with the branching strategy choices for each of them.

You should now be able to do the following:

- Understand what source code management is and how it is necessary for many activities with modern DevOps
- Create a Git repository and play around with the `clone`, `add`, `commit`, `push`, `pull`, `branch`, and `checkout` commands
- Understand what GitOps is and how it fits the modern DevOps context
- Understand why we need GitOps and how it achieves modern DevOps
- Understand the salient principles of GitOps
- Understand how to use an effective branching strategy to implement GitOps based on the org structure and product type

In the next chapter, we will develop a core understanding of containers and look at Docker.

Questions

Answer the following questions to test your knowledge of this chapter:

1. Which of the following is true about Git? (Choose three)

    A. It is a distributed SCM platform

    B. It is a centralized SCM platform

    C. It allows multiple developers to collaborate

    D. It has commits and branches

2. In Git terms, what does Git checkout mean?

    A. Sync code from remote to local

    B. Switch from one branch to another

    C. Review and approve a pull request

3. In GitOps, what forms a single source of truth

   A. The Git repository

   B. The configuration stored in a datastore

   C. The secret management system

   D. The artifact repository

4. Which of the following options are deployment models for GitOps? (Choose two)

   A. The push model

   B. The pull model

   C. The staggering model

5. Should you use Gitflow for your environment repository?

6. For monolithic applications with multiple parallel developments in numerous releases, what is the most suitable Git branching strategy?

   A. Gitflow

   B. GitHub flow

   C. Hybrid GitHub flow

7. Which is the recommended deployment model for GitOps?

   A. The push model

   B. The pull model

   C. The staggering model

Answers

1. A,C,D
2. B
3. A
4. A,B
5. No
6. A
7. B