1ST EDITION

CI/CD Design Patterns

Design and implement CI/CD using proven design patterns

GARIMA BAJPAI | MICHEL SCHILDMEIJER
MUKTESH MISHRA | PAWEL PIWOSZ

Foreword by Helen Beal, Head of Ambassador Program, DevOps Institute (PeopleCert)

CI/CD Design Patterns

Design and implement CI/CD using proven design patterns

Garima Bajpai

Michel Schildmeijer

Muktesh Mishra

Pawel Piwosz

‹packt›

CI/CD Design Patterns

The authors acknowledge the use of cutting-edge AI, specifically Grammarly writing assistant, with the sole aim of enhancing text flow, performing plagiarism checks, and correcting certain sentences within the book, thereby ensuring a smooth reading experience for readers. It is important to note that the content itself has been crafted by the authors and edited by a professional publishing team.

Group Product Manager: Preet Ahuja

Publishing Product Manager: Suwarna Patil

Book Project Manager: Ashwini Gowda

Senior Editor: Mohd Hammad

Technical Editor: Arjun Varma

Copy Editor: Safis Editing

Proofreader: Mohd Hammad

Indexer: Rekha Nair

Production Designer: Gokul Raj S.T

DevRel Marketing Coordinator: Rohan Dobhal

First published: December 2024

Production reference: 1151124

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK

ISBN 978-1-83588-964-0

www.packtpub.com

I would like to thank the Canada DevOps Community of Practice, where I have been able to brainstorm a lot of ideas with chapter leads and senior community members. I would also like to thank the open-source community, which provides us with a platform to steer thought leadership in the DevOps and continuous delivery space, especially the Continuous Delivery Foundation ambassadors. My gratitude toward my co-authors and the entire team at Packt; without them, I would not have been able to bring this book to life. Lastly, I would like to express my gratitude toward my husband, Amit Mishra, and my daughter, Akira, for their relentless support and encouragement.

– Garima Bajpai

Writing this book and sharing my learnings with others has been one of the most rewarding experiences of my life. I would like to extend my heartfelt thanks to my co-authors, Garima, Michel, and Pawel—you all are awesome! I am also deeply grateful to the Packt team, the Continuous Delivery Foundation, and the incredible open source community for their invaluable support.

A special thanks to my wife, Anita, and my family for their love, patience, and unwavering support, without which this book would not have been possible. You are my constant source of strength and inspiration.

– Muktesh Mishra

Foreword

When DevOps emerged in 2009, and for several years after, teams in organizations adopted **continuous integration/continuous delivery** (**CI/CD**) in a haphazard manner. It was mostly tactical, often an experiment in version control or build automation. It was developers trying to avoid merge hell, trying to deliver something better to their friends in IT operations. It was the dawn of DevOps.

More teams in more organizations tried out more tools, hung together DevOps toolchains by accident, bolting on bit after bit, striving for test coverage, test categories, and all the tests needed. Security and performance got involved. People started to wonder what continuous deployment was and what they could do to limit blast radius using canary testing and other progressive deployment techniques such as blue-green. Infrastructure as Code was part of it. It was all a bit chaotic. Heterogeneous pipelines spawned across departments. Arguments broke out over tool standardization. Platform engineering emerged.

And now times have changed. This book is a fantastic example of what we have learned along these paths we have traveled, the paths all the authors have taken with their twists and turns and dead-ends and lights at the ends of tunnels. Together, they bring their wide and varied expertise of what's possible when designing CI/CD solutions for your teams in your organization and how to pick the right design pattern for the outcome you seek to achieve, for the constraints that exist for you.

You will benefit from all the learnings that have come before, distilled into these patterns, abstracted to make it easy for you to collaborate with your colleagues on what's going to work best for you. They address the most common design problems and are adaptable and flexible, so you can make them fit your unique requirements and environments. You truly are standing on the shoulders of giants.

Helen Beal

Head of Ambassador Program, DevOps Institute (PeopleCert)

Contributors

About the authors

Garima Bajpai is an industry leader in DevOps and cloud technologies. She is the founder of the Canada DevOps Community of Practice. She leads the ambassador program for the Continuous Delivery Foundation. Some might know her as a course contributor and instructor for various DevOps courses from leading certification bodies. She has over two decades of experience leading large-scale R&D with a variety of different teams and has helped them adapt DevOps to be able to increase team productivity when it comes to cloud resource deployment. Furthermore, she has collaborated on and contributed to many international conference talks, written several technical blog posts, and published white papers.

Michel Schildmeijer started his career in pharma and then switched to IT, where he increased his multiple-industry knowledge in his role as solutions or IT architect in several industries. At the time of writing this book, he fulfills the role of enterprise architect at the Dutch government. He has received the IT industry-recognized title of Oracle ACE for being an ambassador and community leader in his area of expertise. Michel speaks regularly about technology and the impact of innovation at national and international conferences. He contributes to the open source community and solutions regarding containerization, CI/CD, and DevOps.

Muktesh Mishra is a passionate engineer and open source enthusiast with expertise in cloud computing, DevOps, software architecture, distributed systems, and artificial intelligence. He loves programming across multiple languages and solving complex software problems.

Beyond his professional pursuits, Muktesh is a keen photographer and cook. He has presented at numerous global conferences on topics such as CI/CD, developer productivity, platform engineering, and open source technologies.

Committed to giving back to the community, Muktesh actively contributes to Apache Software Foundation, Microsoft TEALS, and open source forums. He is a proud ambassador for the Continuous Delivery Foundation and the founder of the Sunnyvale Java User Group.

Pawel Piwosz has a long career in SysOps and DevOps areas. With experience in small start-ups and huge, global organizations, he has unique knowledge of processes and executions of DevOps workloads. He is deeply interested in Infrastructure as Code and CI/CD. He authored a CI/CD design framework, which helps organizations create and assess CI/CD processes. He is a DevOps Institute Ambassador, Continuous Delivery Foundation ambassador, and AWS Community Builder. He speaks about DevOps and DevSecOps at multiple conferences and meetups, and he led the DevOps Academy where he was responsible for introducing new engineers to the DevOps world. He is a mentor in the Tech Leaders mentoring program.

About the reviewers

Roshan Mahant is a seasoned senior J2EE/AMANDA solution architect with over 20 years of experience in developing enterprise applications for various industries, including finance, gaming, and e-governance. He has expertise in Java, J2EE, Spring Boot, microservices, and cloud computing (AWS). Roshan has worked on digital transformation projects for state agencies, customizing platforms such as AMANDA for efficient governance. His background also encompasses academic roles in programming and engineering education.

I would like to express my gratitude to my family and friends for their unwavering support throughout this journey. Special thanks to my colleagues for their collaboration and insights, which have greatly contributed to my professional growth.

Mayank Jadhav is a seasoned DevOps engineer with over five years of experience in crafting and optimizing mission-critical deployments. His expertise lies in automating processes, streamlining CI/CD pipelines, and ensuring robust application and infrastructure monitoring. Mayank's DevOps expertise enables him to deliver business-driven solutions. His focus on automation and monitoring consistently drives positive results in his projects. His dedication to staying updated with industry trends is reflected in his ongoing pursuit of knowledge through technical book reviews and contributions to the DevOps community through his LinkedIn presence, where he is recognized as a *Top Cloud Computing Voice*.

Table of Contents

2

Understanding Types of CI/CD Design Patterns and Their Components 33

3

Advancing on CI/CD Design Patterns – from Testing to Deployment 57

4

Business Outcome Alignment with CI/CD Design Patterns 79

Part 2: Structural Design Patterns for CI/CD

5

Exploring Structural CI/CD Design Patterns 95

6

Deployment Strategies for Structural Design Patterns for CI/CD 119

Part 3: Behavioral and Domain-Driven Design Patterns for CI/CD

7

8

Part 4: Creational CI/CD Design Patterns

9

Applying Creational CI/CD Design Patterns 183

10

Understanding Deployment Strategies – Creational CI/CD with Cloud Providers 219

Preface

Hello there! In the evolving landscape of software development, **continuous integration** and **continuous delivery** (**CI/CD**) have become the lifeblood of innovation and efficiency. CI/CD practices empower teams to bring new features or update existing software in order to market in a faster and more reliable way with improved quality.

For organizations of all sizes, adopting CI/CD practices can mean the difference between thriving in a competitive market or lagging behind. CI/CD has become a transformative approach that every organization can benefit from, whether they're a start-up building their first product or an enterprise with established systems.

This book aims to provide a roadmap through the complex landscape of CI/CD, bridging foundational principles with practical design patterns. We'll explore how CI/CD can be optimally structured to meet various organizational needs while being scalable and resilient to change.

The need for CI/CD is evident, but implementing it effectively is often challenging, especially on an organization's scale. This book is designed to fill that gap, offering a pattern-driven approach that takes into account the nuanced requirements of various organizational sizes and development methodologies. You'll find practical advice, industry insights, and pattern-based solutions that can be readily applied to real-world CI/CD challenges.

Whether you're a developer, DevOps engineer, architect, or technology leader, this book will serve as both a reference and a guide. Together, we'll uncover how CI/CD, underpinned by these structural, behavioral, and creational patterns, can elevate your development processes, reduce risk, and increase delivery speed—ultimately, helping your organization move forward with confidence in a fast-paced world.

Who this book is for

The following persona types will specifically benefit from the book:

- **Senior/principal developers and software architects**: Learn about reusable patterns and practices while delivering the software
- **Site reliability engineering architects, DevOps architects, and cloud architects**: Learn about reliable and quality-driven software delivery and how to design complex pipelines with reusable components utilizing different design patterns
- **Engineering managers**: Learn how to gauge the overall quality of the pipeline and delivery of the software along with best practices

This book highlights the relationships and interactions between different tools and practices within CI/CD. The objective is to speed up the development process by providing well-tested, proven development/design paradigms when it comes to CD and its adoption.

Overall, It can be considered a model that you can use to implement CD in specific environments with similar needs.

What this book covers

Chapter 1, *Foundations of CI/CD Design Patterns*, explains the significance of efficient CI/CD in modern organizations. It also provides an overview of the delivery pipeline and its components.

Chapter 2, *Understanding Types of CI/CD Design Patterns and Their Components*, talks about key considerations while designing pipelines and also explains the significance of Infrastructure as Code and pipeline as code in the deployment pipeline.

Chapter 3, *Advancing on CI/CD Design Patterns – from Testing to Deployment*, focuses on pipeline components and stages around artifact management and test automation. It also talks in depth about testing strategies and their significance while designing the delivery pipeline.

Chapter 4, *Business Outcome Alignment with CI/CD Design Patterns*, talks about how efficient CI/CD practices and architecture can be aligned with business goals such as improved user experiences. It also provides insights to users on some of the best practices to align implementation and adoption of CI/CD design patterns toward bigger business outcomes.

Chapter 5, *Exploring Structural CI/CD Design Patterns*, provides an overview of structural components in the pipeline and their interaction, especially in terms of monorepo or polyrepo source code repositories. It also talks about how to consider constructs such as the modularity of the pipelines and components while designing a pipeline.

Chapter 6, *Deployment Strategies for Structural Design Patterns for CI/CD*, talks about different deployment strategies and their trade-offs. It also introduces concepts such as data-driven DevOps where data/feedback loops are integrated into different components by default.

Chapter 7, *Understanding Behavioral Design Patterns for CI/CD*, discusses the pipeline components and their behavior. It also explains the importance of designing observability into the pipeline and its benefits, such as enhanced debuggability and reduced downtimes. More importantly, it also highlights using observability as a mechanism to improve the pipeline behavior and performance.

Chapter 8, *Domain-Driven Design Patterns for Regulated Sectors*, discusses the importance and challenges of applying the CI/CD practices in a regulated sector such as banking/finance or healthcare. This chapter also talks about security and access paradigms such as access controls and approval workflows and their importance and implementations.

Chapter 9, *Applying Creational CI/CD Design Patterns*, focuses on applying creational design patterns such as built-in resilience and scalability into the pipeline design considerations. It provides their implementation in terms of pipeline design for complex and modern software and different workloads (serverless to containers).

Chapter 10, *Understanding Deployment Strategies – Creational CI/CD with Cloud Providers*, covers the various implementations from different cloud providers and their examples. This chapter also talks about implementing CI/CD from a team topologies perspective and considerations.

Chapter 11, *Auditing and Assessment of Design Patterns*, discusses a maturity model for design patterns and also running audits while implementing one or multiple patterns for defining the pipeline.

Chapter 12, *Advanced CI/CD Design Patterns and Use Cases*, covers pipeline design for advanced use cases such as machine learning-based workflows or real-time utility-based workflows. It also talks about various challenges such as performance or implementation and how to solve them with different strategies.

Chapter 13, *Exploring Anti-Patterns for CI/CD Design Pattern Deployments*, talks about common mistakes to avoid while designing the pipelines.

Chapter 14, *Appendix – Knowledge Test and Case Studies*, talks about some real-world case studies and examples for designing pipelines.

To get the most out of this book

Here, we will give you some specifications for trying the examples out yourself.

Software/hardware covered in the book	**Operating system requirements**
Jenkins	Windows, macOS, or Linux (Any)
OpenTofu	Windows, macOS, or Linux (Any)
Kubernetes	Windows, macOS, or Linux (Any)

Download the example code files

There are a few code examples that are also supplemented at multiple places in the book. You can refer to or download the example code files for this book from GitHub at `https://github.com/PacktPublishing/CI-CD-Design-Patterns`. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter/X handles. Here is an example: "The `path` field in the present instance directs toward the `overlays/dev` directory, which houses the configuration settings tailored to the developmental environment."

A block of code is set as follows:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: my-app-dev
spec:
```

Any command-line input or output is written as follows:

```
$ docker build -t name:tag -t name:tag -t tag .
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Go to **Administration | System | Pipelines as Code** and check the **Enable Pipelines as Code** box."

> **Tips or important notes**
>
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share Your Thoughts

Once you've read *CI/CD Design Patterns*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/978-1-83588-964-0`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: Introduction to CI/CD Design Patterns

In this part, you will get an idea of how design patterns can be applied to the CI/CD pipeline. You will also learn about the importance of CI/CD and its impact on the overall organization. Moreover, you will be diving into various components of the pipeline and exploring how applying real-world patterns can enable quality delivery and help in achieving organizational goals.

This part has the following chapters:

- *Chapter 1*, *Foundations of CI/CD Design Patterns*
- *Chapter 2*, *Understanding Types of CI/CD Design Patterns and Their Components*
- *Chapter 3*, *Advancing on CI/CD Design Patterns – from Testing to Deployment*
- *Chapter 4*, *Business Outcome Alignment with CI/CD Design Patterns*

1
Foundations of CI/CD Design Patterns

In this first chapter, we will delve into the significance of **Continuous Integration**/**Continuous Delivery** (**CI/CD**) **design patterns** and principles and how they impact the entire software development process from start to finish. The chapter concludes by summarizing the transformative effect of CI/CD design patterns on software development, highlighting that they are not just a set of tools but also a cultural and operational shift that enables organizations to thrive in the rapidly evolving world of modern software development.

This chapter lays the groundwork for the book, preparing you for a detailed exploration of the specific design patterns and best practices that are fundamental to achieving CI/CD excellence. It also emphasizes the holistic importance of CI/CD as a strategic priority for organizations looking to maintain their innovative edge in the digital age.

In this chapter, we will cover the following topics:

- An overview of design patterns
- Where design patterns meet CI/CD
- Understanding CI/CD and design patterns
- The key components of design patterns – pipelines and infrastructures
- The deployment of CI/CD

An overview of design patterns

To gain a better understanding of specific CI/CD and its relationship with design patterns, it is important to first understand the origin of design patterns and get an overview of the various types that are available. This will help you to understand the purpose and usage of design patterns in general.

Design patterns are general, reusable solutions to common problems that occur during the design and development of software. They represent best practices to solve specific types of issues and provide developers with a way to communicate effective design solutions. These patterns contain the best practices that have evolved over time by experienced software developers to address specific challenges in designing robust, maintainable, and scalable software systems.

The book *Design Patterns – Elements of Reusable Object-Oriented Software*, published in 1994 by four authors known as the *Gang of Four*, popularized the term *design patterns*. However, the concept of design patterns can be traced back to earlier works, such as the book *A Pattern Language* (1977) by Christopher Alexander and his colleagues. This book proposed patterns for architecture and urban planning.

Design patterns help developers to create software architectures that are flexible, modular, and easy to understand. They are not specifically about coding but, rather, about providing solutions to recurring questions, issues, and problems that engineers come across when coding a software solution. You can think of them as a prescription for ensuring a higher quality of code.

There are a few key features of design patterns that are worth noting:

- **Reusability**: Firstly, they promote reusability by encapsulating solutions to common design problems. This means that developers can apply proven solutions to similar problems in various parts of their applications.
- **Abstraction**: Secondly, patterns provide a level of abstraction, which makes it easier for developers to explain the design of a system. They serve as a common vocabulary for discussing and documenting software architectures.
- **Maintainability**: Using design patterns can lead to more maintainable code by making it easier for developers to understand and modify code that follows established and documented patterns.
- **Flexible and adaptable**: Finally, design patterns help create flexible and adaptable software systems. They enable developers to build systems that can evolve and accommodate changes without requiring a complete overhaul.

In the next section, we will discuss types of common design patterns to provide a foundational understanding of the design patterns.

Types of design patterns

In software development, various design patterns are utilized for specific use cases. These patterns allow for the reuse of proven techniques, thereby avoiding the need to reinvent the wheel. Consistent principles and conventions are followed in the design patterns, making them reliable and easy to implement. Design patterns provide a systematic approach to software design and can be beneficial for developers. They can broadly be categorized into creational, structural, and behavioral patterns.

Creational patterns

Creational patterns refer to a group of design patterns that primarily deal with the process of creating objects. These patterns are particularly useful in managing and maintaining object-creation processes by providing various techniques and strategies to create objects in a structured and efficient manner. Some examples of creational patterns include Singleton, Factory Method, and Abstract Factory, each of which offers unique solutions to create objects in different contexts.

A commonly used design pattern is the **Singleton pattern**, which is a creational pattern that ensures only one instance of a class exists and provides a global point of access to that instance. This pattern is useful when only a single object is needed to coordinate actions across a system, such as a configuration manager, logging service, database connection, or connection pool. Here is a diagram containing an example of a Singleton pattern for a logger class.

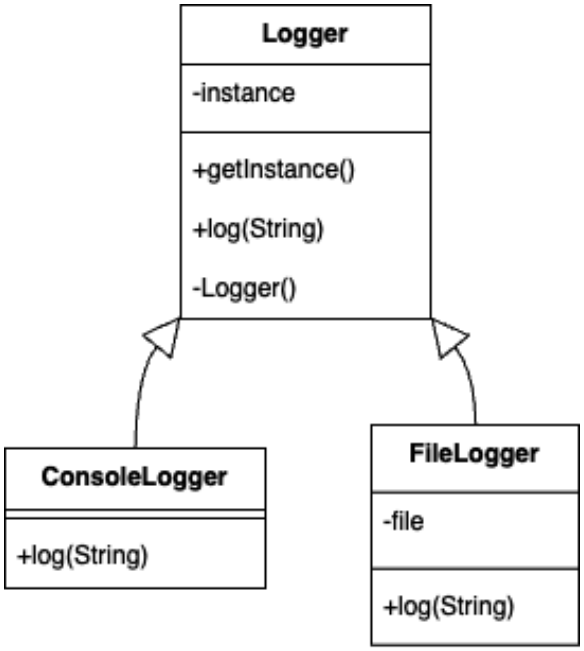

Figure 1.1 – An example of a Singleton pattern for a logger class

The `Logger` class in the example uses the Singleton design pattern to ensure that only one instance of the class is created, which can then be accessed globally. The class has a private static variable, `instance`, which holds the single instance of the class. Whenever the `getInstance()` method is called, it initializes the instance if it's null.

Another common creational pattern is the **Factory Method pattern**. The Factory Method pattern is a fundamental design pattern in object-oriented programming that encapsulates an object's instantiation

process. It defines an interface for object creation, allowing subclasses to specify the type of objects to be created. Essentially, it involves a creator class with a factory method that is either abstract or has a default implementation to create objects. Subclasses have the option to override this method in order to modify the class of objects that will be created. This pattern is especially helpful when a class cannot predict the type of objects it needs to create, or when a class wants its subclasses to specify the objects it creates. The Factory Method pattern promotes loose coupling and enhances scalability, as new classes can be introduced without altering the creator's code.

Let's have a look at a graphical representation of this pattern:

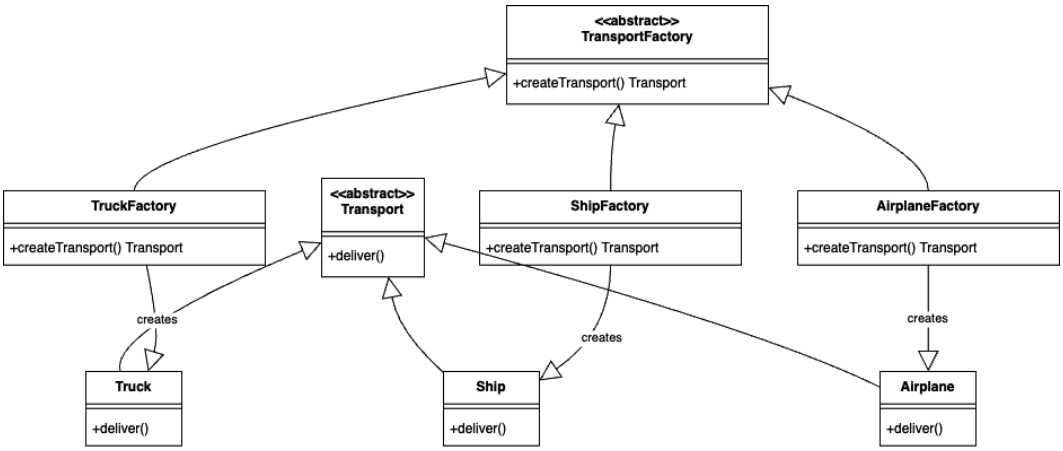

Figure 1.2 – The Factory Method pattern

Imagine a logistics management application that needs to create different types of vehicles such as trucks, ships, and airplanes. We can define an abstract class called `Transport` with a method, `deliver()`. Then, we have subclasses such as `Truck`, `Ship`, and `Airplane` that implement the `deliver()` method. The `TransportFactory` is an abstract class with a `createTransport()` method, which is the factory method. Subclasses of `TransportFactory`, such as `TruckFactory`, `ShipFactory`, and `AirplaneFactory`, implement the `createTransport()` method to instantiate and return an object of the respective type. When the application needs a vehicle, it simply calls the `createTransport()` method on the relevant factory without worrying about the instantiation details. This way, the Factory Method pattern allows the application to remain flexible and scalable, as new types of vehicles can be added without modifying the existing codebase.

Finally, there is the **Abstract Factory pattern**. This pattern is essential in scenarios where a system needs to be independent for object creation, composition, and representation. It acts as a super-factory, creating other factories; this pattern is also known as a factory of factories. This pattern is particularly useful when a system must be configured with one of multiple families of products that are only known at runtime. The pattern provides interfaces to create families of related objects without specifying their concrete classes. By doing so, it encapsulates the creation of objects in a separate object, allowing for the interchangeability of concrete implementations without altering the code that uses them. Despite

its advantages, the Abstract Factory pattern can introduce extra complexity and abstraction, which might not be necessary in simpler applications. It's a powerful tool in the developer's toolkit, but like all tools, it should be used appropriately. This is explored in the following diagram, based on a real-life example:

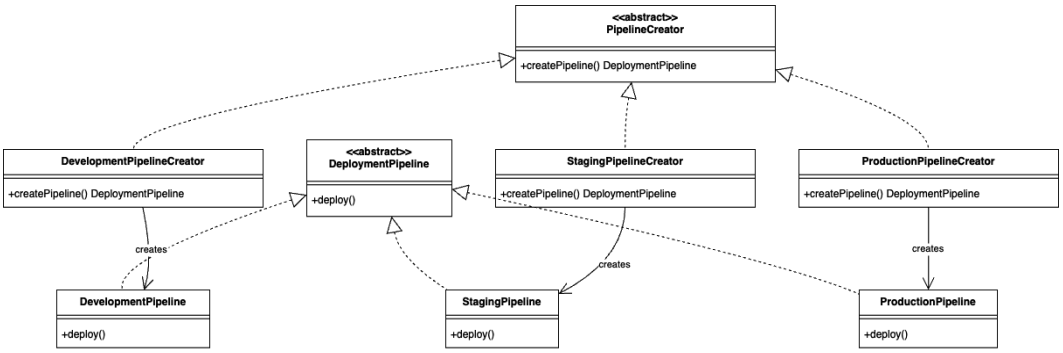

Figure 1.3 – Abstract Factory pattern

When building a CI/CD system for a software project, the Factory Method pattern can be applied to create different types of deployment pipelines, each with specific steps, tools, and configurations. This involves creating an abstract class or interface called `PipelineCreator` with a factory method, `createPipeline()`, and implementing concrete subclasses of `PipelineCreator` for each type of pipeline (e.g., `DevelopmentPipelineCreator`, `StagingPipelineCreator`, and `ProductionPipelineCreator`). Additionally, an abstract class or interface called `DeploymentPipeline` represents the common interface for all deployment pipelines, and concrete classes such as `DevelopmentPipeline`, `StagingPipeline`, and `ProductionPipeline` are created to implement the specific steps to deploy to different environments.

Next, we will take a look at Structural patterns.

Structural patterns

A **structural pattern** is a type of design pattern that aims to simplify the arrangement of classes or objects in a software system. Structural patterns provide well-defined and effective methods for classes and objects to collaborate, resulting in improved flexibility, maintainability, and reusability. These patterns specifically deal with the arrangement of classes and objects and how they can be merged to create more complex structures. Structural patterns also provide guidelines to create a more flexible and maintainable structure in your software design. A few examples of structural patterns are as follows:

- **Adapter pattern**: This pattern helps you to use an already existing class in place of another class. For example, it can be used to make an old version of a class work with a new interface.
- **Bridge pattern**: This pattern separates a concept from its implementation so that it can be changed independently. It can be used to connect different database implementations to a

system. For example, the Bridge pattern separates abstraction from implementation, allowing different shapes to be created using different drawing APIs, without changing the shape or drawing API classes. The `shape` class delegates drawing operations to a drawing API object, which has different subclasses that implement different drawing methods. Clients can create any shape with any drawing API by passing appropriate objects to the shape constructor.

- **Composite pattern**: This pattern is used to create tree-like structures to represent hierarchical relationships between parts and wholes. It can be used to create complex graphical user interfaces where both individual elements and composite elements are treated equally. For example, the composite pattern is a useful design pattern that treats a group of objects as a single unit. It comprises three main components – the **component interface**, the **leaf class**, and the **composite class**. The component interface defines shared behavior for all components, the `leaf` class represents individual objects with no children, and the `composite` class represents complex objects with one or more children.

Let's talk about behavioral patterns now.

Behavioral patterns

Behavioral patterns in software development are a type of design pattern that focus on the interactions and communication between different objects. They help to define the roles and responsibilities of each component in a system and how they work together to achieve a common goal. Types of behavioral patterns include Observer, Strategy, Command, Mediator, and Iterator, explained as follows:

- **Observer pattern**: The Observer pattern, which is widely used in software development, defines a relationship between a subject and multiple observers. The subject maintains a list of observers and notifies them of any state changes. This allows the observers to react accordingly based on their own logic.

  To give an example of an implementation scenario, the Observer pattern is useful when you want to decouple the components of a system that need to be informed of certain events or changes. For example, the Observer pattern can be used to implement a notification system that sends alerts to different channels (email, SMS, push notification, etc.) whenever a new order is placed or a payment is received. Another example is a dashboard that displays various metrics and charts that update automatically when the data source changes.

The following figure shows you a representation of the Observer pattern:

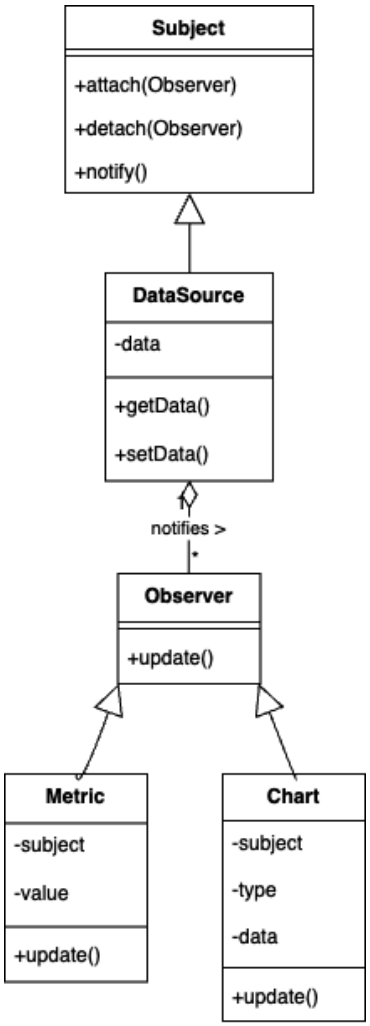

Figure 1.4 – An example of the Observer pattern

The preceding diagram shows how the `Observer` class receives metric and chart updates of a few observers (`Metric` and `Chart`) for `DataSource`. When `DataSource` has new metrics data, it calls `notifyObservers()`, which in turn calls `update()` on each registered observer, such as `MetricsObserver`, to inform them of the changes. This enables a decoupled and flexible design where observers can be added or removed without modifying `DataSource`.

- **Strategy patterns**: A Strategy pattern is a type of design pattern that enables a family of algorithms to be defined and switched out easily. Strategy patterns allow for the algorithms

to be enclosed and varied independently from the clients that utilize them. Essentially, these patterns facilitate the separation of an algorithm's strategy from its execution, enhancing the code's adaptability and reusability.

The best way to describe it is by this example. Suppose you have a class called `Animal` that has a method called `makeSound()`. Different animals make different sounds, so you want to implement this method differently for each subclass of `Animal`. However, you don't want to hardcode the sound logic in each subclass, as that would violate the open-closed principle and make your code less flexible and maintainable. One way to solve this problem is to use a Strategy pattern. The following figure illustrates this example that applies to the Strategy pattern:

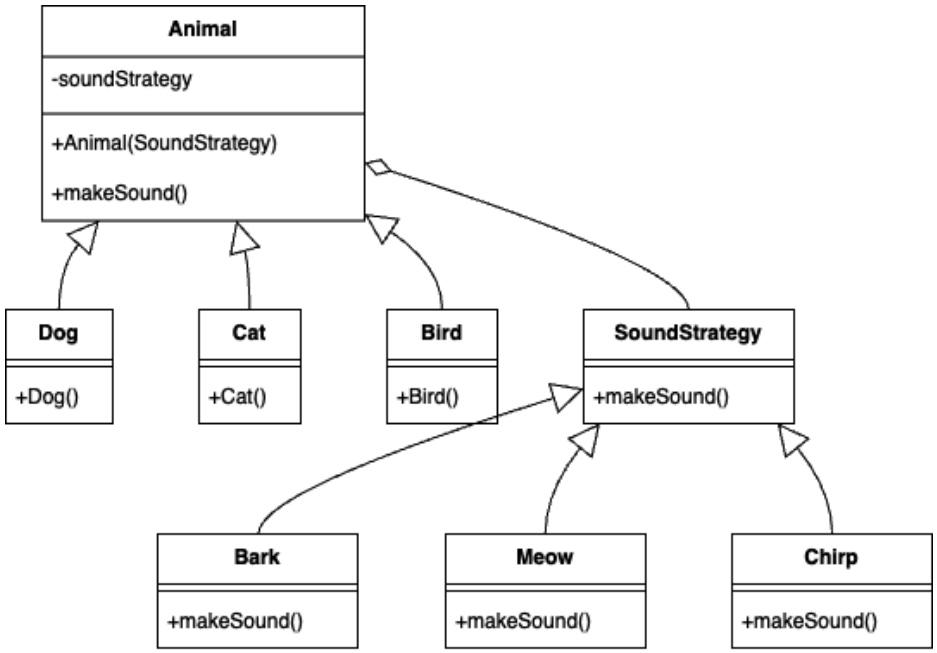

Figure 1.5 – An example of a strategy design pattern

Here, you can change the behavior of the `makeSound()` method at runtime by passing different strategy objects that represent various animal sounds, such as meow or bark. This makes the code more flexible and maintainable, as you can add new behaviors without altering existing code.

- **Command patterns**: A Command pattern is a type of design pattern that encapsulates a request as an object. This allows a request to be parameterized, queued, logged, or undone. Command patterns are useful for implementing user interfaces, network protocols, and transactional systems. These patterns decouple the sender of a request from the receiver, making the code more modular and reusable. For example, undo/redo operations (implemented by a created `command` object), macro recording (to record a sequence of commands), and menu actions

(also implemented as a command object) are all executed through a method that performs the specific action.

- **Mediator patterns**: The Mediator pattern is designed to simplify and streamline communication between objects while reducing dependencies and complexity. This is achieved by encapsulating the interactions between objects in a mediator object, which acts as an intermediary and defines an interface for communication between the objects. Using this pattern, objects don't need to know each other's identities or references, and they can vary their interactions independently. Mediator patterns can help make code maintenance easier, improve the modularity and testability of a system, and facilitate code reuse.

  For example, using a Mediator object makes it possible to allow other objects to be loosely coupled, as the mediator coordinates interactions between the objects. It's useful when you have a large number of objects that need to communicate but you want to avoid that same large number of references between them. In the following diagram, the `Mediator` class defines an interface to communicate with colleagues:

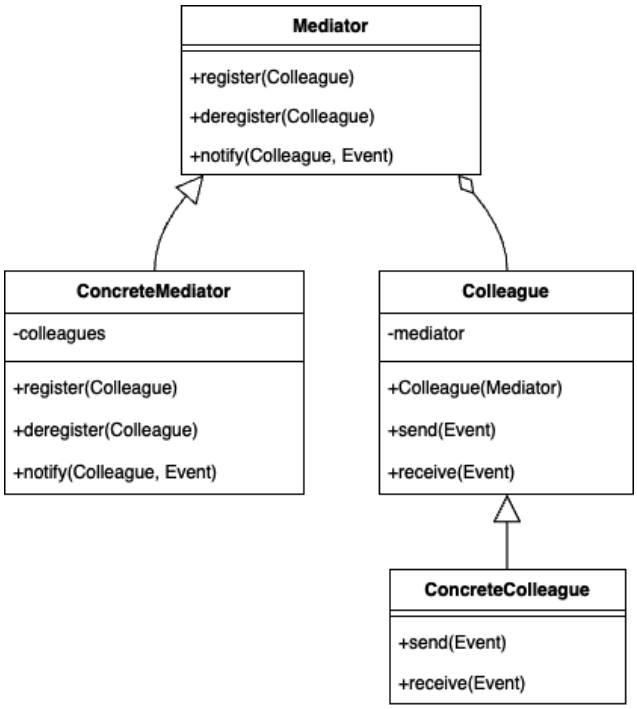

Figure 1.6 – An example of the Mediator pattern

- **Iterator patterns**: The Iterator pattern is a common design technique in software engineering that allows sequential access to the elements of a collection, without exposing its underlying representation. Iterator patterns can be implemented in various ways, such as using cursors, generators, coroutines, or closures. The main benefits of Iterator patterns are that they decouple

the client code from the collection implementation, support multiple simultaneous traversals, and enable lazy evaluation of the elements.

In Java, the `Iterator` interface provides methods to iterate over any collection of objects, regardless of how they are stored internally. Here is an example diagram of an Iterator pattern to read a file line by line.

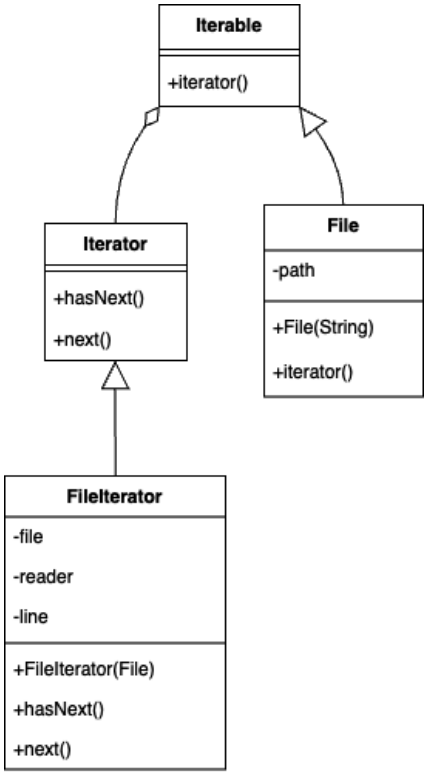

Figure 1.7 – An example of the Iterator design pattern

This concludes this section. Next, we will see how all these design patterns relate to CI/CD.

Where design patterns meet CI/CD

In this section, we will explore the basics of design patterns in general, which gives you a good foundation for the following sections of this chapter, where we will focus more on the CI/CD part of design patterns.

Design patterns can help enforce CI/CD pipelines by providing guidance on how to structure code, manage dependencies, handle errors, and perform software testing. Design patterns and CI/CD serve the same purpose of improving the efficiency and quality of software development. For example,

the adapter pattern can be used to combine different tools or services in the pipeline, the Observer pattern can be used to notify various stakeholders of the pipeline status, and the Strategy pattern can be used to encapsulate different deployment algorithms, such as blue-green, canary, or rolling, and select them at runtime.

Conversely, CI/CD may also influence the selection of design patterns by imposing specific code constraints or requirements. The code should be modular, testable, and easy to deploy to comply with CI/CD's standards. In the next section, we will take a deeper look at it.

Understanding CI/CD and design patterns

In this section, we will start with the basic definition and purpose of CI/CD, and their related processes, practices, and tools. The most widely referred definition of CD is provided by Jez Humble (and can be found at `https://continuousdelivery.com/`):

> *"Continuous Delivery is the ability to get the changes of all types of new features, config changes, bug fixes, and experiments into production or into the hands of the user safely and quickly in a sustainable way."*

According to the CD Foundation, continuous integration is defined as follows (which can be found at `https://bestpractices.cd.foundation/learn/ci/`):

> *"Continuous Integration, the CI in CI/CD, is the practice of combining code changes frequently where each change is verified on check-in."*

Over time, CI/CD practices and their application have evolved, providing practitioner tools to implement CI/CD efficiently. The evolution has also resulted in exponential growth in tools, operational overheads, and cognitive load for the practitioners. There are many aspects of learning and people are falling behind due to increasing complexity, continuous evolution, and a lack of time for self-development. To fully realize the potential of CI/CD, it is important to balance investment in tools and standardization of posture for CI/CD vis-à-vis rapid innovation.

As CI/CD comes of age, the factors that are important to consider in evolving the CI/CD posture are making CI/CD simple to adopt, scale, and manage. In this section, we will attempt to define the design patterns for CI/CD, keeping in mind the aforementioned factors.

First, we will look at the CI/CD structure itself to understand the big picture of its design, which involves explaining the build cycle and manual deployments.

The build cycle

Understanding the build cycle is essential for implementing CI/CD effectively. The build cycle includes the following phases:

- **Initialization**: During the initialization phase, the build system prepares the necessary environment and dependencies.

- **Compilation**: Compilation is where the source code is converted into executable code.
- **Testing**: Testing involves running automated tests to ensure the quality of the build.
- **Packaging**: Packaging is the process of assembling the code into a deployable format.
- **Reviewing**: Reviewing involves checking the build for any potential issues before it is released.

The second cycle to elaborate is manual deployment; let's take a look at what it consists of.

Manual deployment

Manual deployment, while less common in a fully automated CI/CD pipeline, is still relevant in certain contexts, particularly in production environments where manual oversight is desired. The manual deployment process typically involves a series of steps that need to be performed by an operator, such as the following:

- Pulling the latest code from the repository
- Configuring the environment
- Executing the deployment scripts
- Verifying that the deployment was successful

In conclusion, CI/CD practices, when implemented with a clear understanding of the build cycle and deployment processes, can significantly enhance the efficiency and reliability of software development projects. By automating as much of the process as possible and maintaining manual control where necessary, teams can ensure that their software is always ready for production, with minimal risks and delays. In the upcoming chapters, we will take a deeper dive into CI/CD best practices and explore how to tailor these processes to your organization's needs.

Next, we will take a closer look at the evolution of CI/CD.

The evolution of CI/CD

Before we jump into exploring specific patterns that can be applied in CI/CD, we should remember that CI/CD can also be considered a pattern. Some practitioners consider CI as a pattern of collaborative work and a way of delivering and storing artifacts. How we build CI depends on many factors. The starting point is a branching strategy, which influences the way CI is triggered. But this is just the beginning. In fact, we should remember that a branching strategy can impact the delivery method at the end of the process.

For the CD part, the factors become even more complicated. Two patterns hide under this single abbreviation – **continuous delivery** and **continuous deployment**. Although both are interchangeably positioned in the same place, these two patterns are very different.

A continuous delivery pattern introduces some manual actions. It might be a test, an approval, and so on. Continuous deployment, however, doesn't contain any manual element. The whole process is automated. Moreover, if CI contains any manual step, continuous deployment is not achievable. The only option is continuous delivery. We can clearly see that these two patterns (CI and CD) have a very huge impact on each other.

We will discuss how to design CI and CD patterns in the next chapters of this book; however, one element is important at this point. This element declares strictly where CI ends and CD starts. It is a connection point between these two patterns. This connection point relates to publishing the artifact, which is the final step of CI, and collecting this artifact, which is the first phase of CD. In the next section, we will discuss the evolution of patterns for CI/CD during the last decade, which will pave the way for the exploration of design patterns and the subsequent structural components.

The maturity journey of CI/CD

Many attempts have been made to simplify, refine, organize, and implement CI/CD. These range from a broader attempt to build a common vocabulary for CI/CD practices to reference architectures, and finally, design patterns, which are existing implementations, often tagged as recurrent solutions.

The reference architecture of CI/CD

A reference architecture often involves high-level guidelines and principles. It refers to an abstract concept that can outline a structure or a construct for CD. Some examples of reference architecture are as follows:

- An example of a reference architecture and associated service offering in the cloud is Open Source on AWS: `https://aws.amazon.com/blogs/devops/choosing-a-well-architected-ci-cd-approach-open-source-on-aws`.
- Another example of a CD reference architecture is the one being worked on by the **Continuous Delivery Foundation** (**CDF**): `https://bestpractices.cd.foundation/architecture/`.

Reference architecture paves the way for the maturity of CI/CD and leads us to the next steps toward understanding the differences between patterns and reference architecture.

The differences between patterns and reference architectures

A reference architecture is a broader abstraction; in some cases, it is more appropriate to compare such architecture to a set of patterns instead of one. Another important difference between a pattern and a reference architecture is that a pattern must be a recurrent solution that was previously used on existing implementations.

A reference architecture is often helpful in establishing the structure for a new, innovative approach whose usage might not have been implemented or tested yet in real life. So, we can clearly see that

the design pattern for CI/CD is the next step in the evolution of establishing a common vocabulary, interaction points, and proven implementation. In the next section, we will discuss the first steps in the evolution of implementation for CI/CD, which will pave the way for establishing design patterns for CI/CD.

The evolution of CI/CD deployments – push and pull-based implementation

Now, we understand the evolution of CI/CD and the efforts to create standard vocabulary and guidelines for practitioners, by outlining reference architecture and how it is associated with CI/CD at a conceptual level. Now, it is time to understand the recent evolution of the deployment of CI/CD. The deployment and implementation of CI/CD pave the way to establish design patterns that can be considered solutions to recurrent problems while trying to implement CI/CD in specific domains.

With the emergence of new technologies, the implementation of CI/CD has changed, sometimes radically. From the launch of **Kubernetes** to the emergence of **GitOps**, the implementation and deployment approach has evolved. The primary change is in the method of triggering the CI/CD pipelines, approaches toward quality control and testing approaches, and finally, deployment of infrastructure. For example, GitOps redefined the perspective of the execution model and introduced a pattern that was the opposite of the most popular one.

Let's have a look at these two patterns, called push and pull models.

- **The push deployment approach** is the most popular CI/CD triggering deployment and was introduced to CI/CD tooling from the beginning, when CI/CD was fairly new in the field. In this approach, a change in code, committed to the **version control system** (**VCS**), triggers the pipeline. This means that when the change is committed, the execution starts immediately. There is no waste of the trigger's time.
- **The pull deployment approach**, introduced as a concept implemented with Kubernetes-specific tools, such as Argo CD or Flux, works differently. As Kubernetes is a complicated but coherent system that can manage itself, GitOps provided the pull model. GitOps not only redefined this aspect, but it also revolutionized the entire approach to delivering applications. In the pull approach, execution is not triggered by the code change itself. The system uses a special controller that constantly manages the state of the system. In general, this management is done by comparing the **desired state** and the **actual state**. The desired state is stored in the VCS. The actual state is the current state of the system itself. It means that this controller manages not only the number of instances of the application but also the version of the code used by it. When the controller realizes that there is a new version of the code in the **VCS**, it will trigger the deployment.

These patterns serve the same purpose, the delivery of code to an environment, but they are different under the hood – the way the trigger is defined, how the changes should be calculated, and so on.

Both the aforementioned patterns should contain elements that are the same. Testing is one of these elements.

Implementing testing in CI/CD

How tests are run in the CI/CD pipelines affects the implementation and structure we will use to build and execute them. What approach we choose can impact or even block specific CI/CD implementation. For example, introducing manual testing in the process blocks CD, as it is fully automated.

In this section, we will focus on general testing rather than specific test types. It is important to select the proper type of test, especially when scale is a factor. In this chapter, we will describe the most popular pattern-test pyramid and the emerging one-test diamond, which targets distributed systems.

When we design a testing approach, it is very important to understand how the system works and how it is architected. We have many architectural approaches at our disposal today, ranging from monoliths to distributed systems, such as microservices. A proper testing strategy has crucial areas to ensure that quality attributes (known previously as non-functional requirements) are met.

Designed tests must be implemented into a CI/CD process. They are part of automating the process and following DevOps principles. Proper implementation of tests in CI/CD not only ensures automation but also enables shortened feedback loops, measurements, and so on.

The test pyramid

The **test pyramid** is a well-established pattern of testing software and incorporating these tests into the CI/CD process. The following diagram shows a test pyramid:

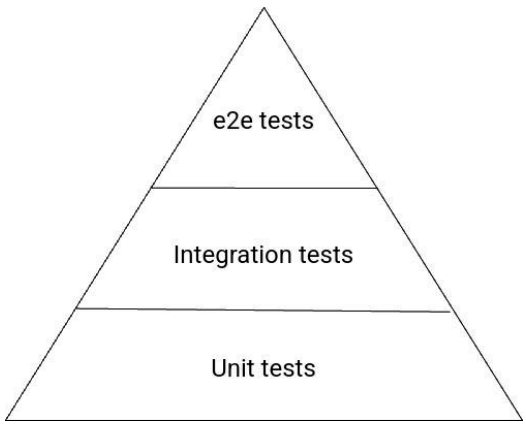

Figure 1.8 – The test pyramid

In this approach, we establish tests based on their impact, the complexity of writing specific test cases, isolation, integration, and so on. This book is not about tests, so you will not find explanations of

every test level here. For now, it is enough to understand that the bottom of the pyramid is defined by unit tests. **Unit tests** are the simplest, fastest, and least expensive in development and execution. The goal of unit tests is to catch the most common issues as early as possible in the development process by having the smallest testable procedures. For example, through unit tests, developers can check the behavior of the `if` statement in a function. These tests can be applied at every stage of a project's life.

The next layers in the pyramid cover more and more complex tests – integration, End-to-End (e2e), service, system, UI, and so on. The idea is to not run complicated tests if easier ones weren't run.

The test diamond

The **test diamond** is a recent approach to tests in the DevOps world. While the test pyramid is great for testing many applications, it works best for monolithic applications. In the distributed services world, microservices rule the roost, and the testing approach must reflect that change too.

This change is significant, as the whole communication layer between parts of an application is now removed from the application logic itself. This means that the focus on tests should be different.

What changes in the test diamond is not the order of the tests executed but, rather, the focus and weights of the tests. Microservices are distributed and highly rely on the integrations between them, so the tests must reflect that. That is why the diamond pattern focuses on integration tests, as shown in the following figure:

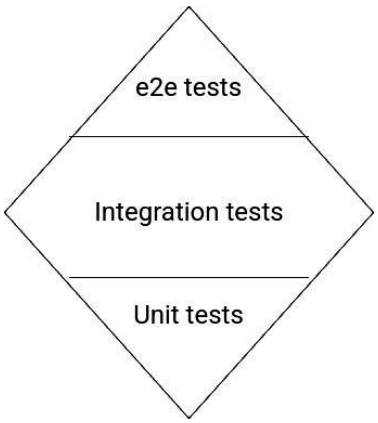

Figure 1.9 – The test diamond

Although the testing pattern doesn't sound like a design pattern for CI/CD, it has huge implications for the execution of pipelines, scale, and performance.

These two patterns cover the same types of tests. The difference is in the emphasis on specific tests, which is enforced by the architecture approach. For example, in monolithic applications, the integration tests are less extensive than in microservices, where we must ensure that an application not only works

with third-party services or a database but also with other parts of the application, which are detached and encapsulated as separate microservices.

The evolution of CI/CD deployments – infrastructure

Working with infrastructure and its deployment is different than with an application's code. For engineers, an **Infrastructure as Code** (**IaC**) approach opens the same doors as delivery. So, the same approach is used – generic CI/CD tools. However, we should remember that infrastructure works under different laws than application code. It is subjected to different kinds of tests and has different artifacts, and every change might have a significant impact on the stability, security, and behavior of an entire system.

While considering IaC in CI/CD, we need to cover a few concepts and apply proper patterns to manage these concepts:

- **State file management**: Some tools, such as OpenTofu and Terraform, allow engineers to use **state files**. State files can be considered as an intermediate state between code in a template and created infrastructure. This state consists of mapping all resources created with a specific version of the template.

  To determine the state, it is important to not only keep the state file secure (by secure, we mean well-known security means ensuring that the state file is protected by clearly defined access controls, specifying exactly who can access it to prevent unauthorized access, removal, or damage) but also ensure that it will not be removed or damaged. Additionally, the state file should be used to create an artifact from the CI/CD execution.

  Let's explore an example with OpenTofu here. Imagine a situation where a pipeline is run against existing infrastructure. In order to find out what Tofu will do during the actual execution, we run the following command:

  ```
  $ tofu plan
  ```

  This command will create a so-called *plan* of changes.

  The pipeline should run a similar command; however, this plan should be stored as an artifact for this execution. Therefore, we run the following command:

  ```
  $ tofu plan -out <filename>
  ```

  This will create a file with the `plan` data. After this artifact is created, it can be subjected to inspection and testing, stored in artifact storage, and later used to execute changes.

- **Pull requests (merge requests)**: Infrastructure changes are very sensitive. Change in one resource can not only impact the security of a whole platform but also its behavior and stability. It is crucial to ensure that changes deployed to infrastructure are checked. Pull requests allow organizations to control the changes at scale, not only by providing guardrails and tests for the quality of the changes prepared but also by keeping these changes auditable.

- **Cost analysis**: With pull request functionalities come options where cost estimation can be performed. Tools such as Infracost or Terracost should be implemented as gates in pipelines to control cloud expenses.
- **Golden images**: If an organization uses virtual machines, the creation of a golden image is a way to ensure the repeatability and stability of the infrastructure update processes. A golden Image is an image of a virtual machine, where all recent changes are installed and tested. This approach ensures that the component (in this case, the virtual machine) is ready to be used very quickly. This way, we eliminate the time needed to install all elements on the machine, eliminate risks that some source is not available or a component cannot be installed, and so on.
- **Ship a configuration down**: In a complicated infrastructure, which is built from many templates and even based on many tools (e.g., OpenTofu, Kubernetes, and Ansible), it is critical to ensure a proper flow of configuration.

  Terraform, CloudFormation, and OpenTofu can evaluate the identifiers of expected resources by using a special syntax, such as in this example for OpenTofu:

```
data "aws_vpc" "this" {
  id = <vpc_id>
}
```

  The additional work that is needed here is a need to know the ID of the AWS **Virtual Private Cloud** (**VPC**), the virtual network used in AWS to separate the network. This might break the automated process and force engineers to provide the data manually.

  This can be overcome using another approach:

```
data "aws_vpc" "this" {
  filter {
    name = "tag:Name"
    values = ["<vpc_name>"]
  }
}
```

  This snippet will evaluate the data by the name of the VPC, so it is easier to keep it as a variable in the project.

  Although this is a popular approach, it is risky. All templates must follow very strict rules, and it is very easy to break the chain by leaving a typo in the VPC's name, a missing tag, and so on.

  That is why tools such as Spacelift offer the functionality of passing data between executions. This ensures consistency between different template executions and allows us to build scaled pipelines.

- **Drift detection and remediation**: Drift is one of the biggest risks for infrastructure. Simply put, drift is the difference between a template (which is a desired state) and the actual deployed infrastructure. The manual change done on the system is the most often cause of drift.

Any deviation from a desired state is a security risk and must be remediated. The detection must be as quick as possible to minimize the potential risk.

This leads us to a specific pattern, a pipeline, which is solely responsible for detecting the drift and remediating it when needed.

Of course, having separate pipelines for drift leads to overcomplicated management, especially at scale. We will cover the ways of managing this later in the book.

In the next section, we will further elaborate on the implementation and basic building blocks of CI/CD design patterns, covering pipelines and infrastructure.

The key components of design patterns – pipelines and infrastructures

In this section, we will discuss the next step after deployment approaches, which is introducing the building blocks for the CI/CD design pattern. We will focus on the crucial aspects of design patterns, which are associated with pipelines and infrastructures. We will do this while keeping in mind the implementations discussed in the *Understanding CI/CD and design patterns* section. It's important to understand that design patterns are reusable solutions that help software developers solve common programming problems. These patterns make it easier to write code that can be maintained, extended, and tested effectively.

Let's outline the key components of the CI/CD design patterns:

- **Pipelines**: These are a series of steps that process data to convert raw input into meaningful output. Design patterns such as the Chain of Responsibility, Iterator, Pipeline, and Pipes and Filters can be used to implement pipelines. These patterns help to decouple the logic of each step from the flow of data, allowing the composition of different pipelines from reusable components.
- **Infrastructures**: These refer to the systems that support the execution of pipelines. They can include hardware, software, networks, databases, cloud services, and so on. Infrastructures can be built using various design patterns, including Abstract Factory, Builder, Factory Method, and Singleton. These patterns help to abstract away the details of creating and managing infrastructures and provide consistent interfaces to access them.

By utilizing pipelines and infrastructures, we can improve the performance and quality of our software systems. We can also leverage the collective knowledge and experience of other developers who have faced similar challenges.

Pipelines – definition and characteristics

Pipelines are a common architectural construct for CI/CD that aim to improve software quality and speed by automating the software development life cycle. They consist of a series of stages, from source control and integration to monitoring and feedback.

In the realm of CI/CD, pipelines are a series of actions that must be taken to release a software update. Each action within the pipeline serves a specific purpose, which handles compiling the code to conduct tests and finally deploying it. These pipelines are adaptable and can handle various forms of data, such as source code, configuration settings, and build artifacts.

Pipelines can be configured in different ways, depending on the needs and preferences of the developers and the organization. Some of the common types of pipelines are as follows:

- **Basic pipelines**: These run all the tasks in each stage concurrently and move on to the next stage once all tasks in the previous stage have been completed. For example, a basic pipeline might have stages to build, test, and deploy.
- **Directed Acyclic Graph (DAG) pipelines**: A DAG pipeline is designed based on the dependencies between jobs. For example, a DAG pipeline might have jobs for linting, unit testing, integration testing, and code coverage, which can run in parallel or sequence, depending on their dependencies. In the following diagram, you can see an example of a simple DAG pipeline:

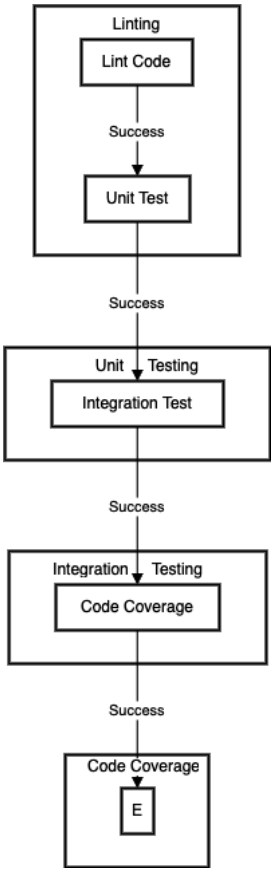

Figure 1.10 – An example of a DAG pipeline

- **Merge request pipelines**: These pipelines only run for merge requests, not every commit. For instance, a merge request pipeline might run tests and code analysis on the changes proposed by the developer before they are merged into the main branch.
- **Merged results pipelines**: These are pipelines for merge requests that assume that changes from the source branch are already merged into the target branch. For example, a merged results pipeline might run tests and code analysis on the merged code to check for any conflicts or errors.
- **Merge trains**: Merge trains use merged results pipelines to queue merge requests one after another. This ensures that only one merge request is merged at a time and that each merge request is tested against the latest version of the main branch. This approach allows developers to reduce manual errors, improve collaboration, and enhance customer satisfaction through the use of pipelines.

Pipelines are an essential part of CI/CD, as they enable developers to deliver software faster, more reliably, and more securely. By using pipelines, developers can reduce manual errors, improve collaboration, and increase customer satisfaction. They can be implemented using various tools and frameworks, which will be discussed in the following section.

Pipelines – techniques and tools

CI/CD pipelines have become integral to modern software development, offering a systematic approach to building, testing, and deploying applications. These pipelines automate the software delivery process, enabling faster releases, increased collaboration, and improved code quality.

Pipeline structure and components

A typical CI/CD pipeline comprises several stages, each representing a specific phase in the software delivery life cycle; they are as follows:

- **Source control integration**: The pipeline starts with integration with VCSs (e.g., Git), triggering the pipeline on code changes.
- **Build stage**: The compilation of source code and creation of executable artifacts.
- **Testing stage**: Automated testing to ensure that code meets quality standards.
- **Deployment stage**: Automated deployment of an application to development, staging, or production environments.
- **Monitoring and feedback**: Continuous monitoring and feedback loops to track performance and identify issues.

Now that we have looked at the definition and basic structures of pipelines, we'll move on to look at the orchestration of pipelines.

Orchestration – coordinating pipeline execution

Orchestration is the technique of coordinating and scheduling the execution of various pipeline stages. Effective orchestration ensures a smooth flow of activities, from code commit to deployment. Some popular orchestration tools include the following:

- **Jenkins**: An open source automation server that supports building, testing, and deploying code. Jenkins provides extensive plugin support for integrations with various tools and technologies.
- **GitLab CI/CD**: Integrated CI/CD capabilities within the GitLab platform, offering seamless pipeline configuration and execution directly from the VCS.
- **Travis CI**: A cloud-based CI/CD service that integrates with GitHub repositories, making it easy to set up and manage CI/CD pipelines for open source projects.
- **GitHub Actions**: GitHub Actions is a useful feature of GitHub that enables you to automate your software development workflows. With GitHub Actions, you can conveniently build, test, and deploy your code directly from GitHub. Additionally, you can integrate GitHub Actions with other tools and services to streamline your workflow further. Moreover, you can even create and share your own actions or use actions made by the GitHub community.
- **Azure DevOps**: Azure DevOps is a comprehensive set of cloud-based services provided by Microsoft that support the entire development life cycle. It includes a variety of tools and services for planning, developing, testing, and deploying software applications efficiently. The cloud-based nature of Azure DevOps enables team members to collaborate, workflows to be automated, and seamless integration with other Azure services and third-party tools. Whether you're working on a small project or a large enterprise application, Azure DevOps provides a scalable and efficient platform for software building and deployment.
- **Argo CD**: It is a robust continuous delivery tool that is specifically designed for Kubernetes. It uses the GitOps pattern to ensure that the Git repositories serve as the only source of truth to define the desired state of an application. It automates the deployment process of these states in the target environments. This makes it easier and more efficient to manage Kubernetes applications. It can also be used as an orchestration tool for CI/CD by integrating with other tools and platforms, such as source control, testing, monitoring, or logging. It is compatible with different pipeline models, deployment strategies, and feedback loops.
- **Tekton Pipelines**: Tekton Pipelines is an open source tool, part of the CD Foundation, which enables you to build and execute CI/CD pipelines on Kubernetes. This platform utilizes custom resources to establish the tasks, steps, and parameters of your pipelines. Additionally, it can integrate with other tools and platforms, for tasks such as source control, testing, monitoring, and logging. Red Hat OpenShift uses it as the default tool for supporting CI/CD pipelines.

Orchestration is important, but it's also very important to test to ensure code quality and reliability, which we'll dive into next.

Testing – ensuring code quality and reliability

Testing is a crucial component of CI/CD pipelines. It ensures that any changes made to software do not introduce any defects or regression issues. Various tools and techniques are used to test the functionality, performance, security, and usability of software products. Testing can be done at different levels of granularity, such as unit testing, integration testing, system testing, and acceptance testing. The complexity and scope of a software project determine whether testing can be done manually or automatically. Let's have a look at the different methods of testing:

- **Unit testing**: Testing individual components or functions to ensure that they behave as expected. Tools such as Pytest, JUnit, and Mocha are commonly used for unit testing.
- **Integration testing**: Verification of interactions between components or services is facilitated by tools such as Selenium (for web applications) and Postman (for APIs), which aid in integration testing.
- **Acceptance testing**: This is the process of testing whether a software system meets the requirements and expectations of the end users or stakeholders, using tools such as Cucumber, SpecFlow, and Robot Framework.
- **End-to-end testing**: Validating an entire application's functionality by simulating real user scenarios is commonly done using tools such as Cypress and TestCafe.

Now that we have covered the testing components, we will next look at deployment strategies.

Deployment strategies – efficiently releasing software

Deployment strategies define how new code changes are released to different environments. Some common deployment strategies include the following:

- **Blue-green deployment**: Involves maintaining two identical production environments, with one handling live traffic (green) and the other being prepared with the new release (blue). The switch is made once testing is successful.
- **Canary deployment**: The gradual release of new features to a subset of users before a full rollout. This allows you to monitor and detect issues early.
- **Rolling deployment**: A rolling deployment is a gradual deployment strategy that replaces the previous version of an application with the new version by updating one instance or node at a time.
- **Feature toggles**: Enabling or disabling specific features during runtime allows for greater flexibility when releasing and rolling back changes. Although not technically a deployment strategy, this approach is worth mentioning, as it enables system behavior to be modified without altering the underlying code.

These strategies will be handled in more detail in the *The deployment of CI/CD* section coming up later in this chapter. Next, we will discuss logging and monitoring.

Logging and monitoring

Effective logging frameworks and monitoring tools are essential for tracking pipeline steps' execution, identifying issues, and debugging errors. In the industry, there are multiple tools available. Some of the most well-known ones are the following:

- **ELK** (**Elasticsearch, Logstash, and Kibana**) **stack**: This combines Elasticsearch for log storage, Logstash for log processing, and Kibana for log visualization. It offers real-time search, analytics, and centralized log management.
- **Prometheus and Grafana**: A popular combination for monitoring and alerting. Prometheus collects metrics, while Grafana provides a visually appealing dashboard for analysis.
- **New Relic and Datadog**: These are comprehensive monitoring solutions that offer application performance monitoring, infrastructure monitoring, and real-time analytics.

CI/CD pipelines are an essential part of software delivery. When implemented with the right techniques and tools, they can play a crucial role in streamlining the process. A robust CI/CD ecosystem is made up of orchestration tools, testing frameworks, deployment strategies, and logging and monitoring solutions. As software development continues to evolve, it is increasingly important for teams to leverage these techniques and tools to deliver high-quality software efficiently and consistently.

Pipelines and Infrastructures

Infrastructure includes servers, networks, storage, databases, containers, and cloud services that enable pipeline operations. Tools such as Terraform, Ansible, Chef, and Puppet automate infrastructure setup and management. Monitoring tools such as Prometheus, Grafana, and Nagios provide insights into infrastructure health and status for better control and management. Pipelines and infrastructures work together to enable CI/CD practices. By using pipelines and infrastructures, developers can ensure that code changes are integrated and delivered consistently and efficiently. Pipelines and infrastructures also help to improve the quality, security, and performance of software products. Infrastructure includes servers, networks, storage, databases, containers, and cloud services that enable pipeline operations. Tools such as Terraform, Ansible, Chef, and Puppet automate infrastructure setup and management. Monitoring tools such as Prometheus, Grafana, and Nagios provide insights into infrastructure health and status for better control and management.

Some of the benefits of integrating infrastructures into pipelines are as follows:

- **Faster feedback**: Developers can receive prompt feedback on their code changes through automated testing and code reviews, helping them to identify and resolve errors early in the development process.

- **Higher reliability**: Developers can deploy software updates more frequently and with less risk of breaking the existing functionality. This helps to reduce downtime and improve customer satisfaction.
- **Better collaboration**: Developers can work together more effectively by using common tools and standards. This helps to avoid conflicts and ensure alignment across teams.
- **Lower costs**: Developers can optimize the use of resources by using cloud services and containers. This helps to reduce operational expenses and increase the scalability of the software products.

In conclusion, pipelines and infrastructures are essential components of CI/CD practices. They enable developers to automate the testing, building, and deployment of software applications, ensuring faster delivery and higher quality. Pipelines and infrastructures also facilitate collaboration, feedback, and monitoring throughout the software development life cycle. Therefore, it is important to design and maintain pipelines and infrastructures that are reliable, scalable, and secure.

The deployment of CI/CD

Deployment strategies are a set of practices and methods that are used to deliver software to different environments, such as development, testing, staging, and production, with minimal downtime and risk. The advantages and disadvantages of different deployment strategies depend on the type, complexity, and requirements of the software. CI/CD processes can improve software delivery in terms of quality, reliability, and efficiency, as they ensure that code meets the required standards and requirements before it is released. This section discusses some common deployment strategies.

Rolling deployments

Rolling deployments are a strategy for updating software applications without causing downtime or disrupting the user experience. They involve gradually replacing the old version of the application with the new one, usually by deploying the update to a subset of servers or users at a time. This way, if there are any issues with the new version, they can be detected and fixed before affecting the entire system. Rolling deployments also allow you to test the performance and compatibility of the new version in a live environment, as well as collect feedback from users.

Rolling deployments have several advantages over other deployment strategies, such as blue-green deployments or canary deployments. They are more efficient, as they do not require duplicating the entire infrastructure or maintaining two versions of an application at the same time. They are also more flexible, as they allow you to adjust the pace and scope of the deployment based on the results and feedback. They are less risky, as they minimize the impact of potential errors or bugs on a system and its users. They are more user-friendly, as they avoid interrupting a service or forcing users to switch between versions. The following figure represents a basic implementation of a rolling deployment:

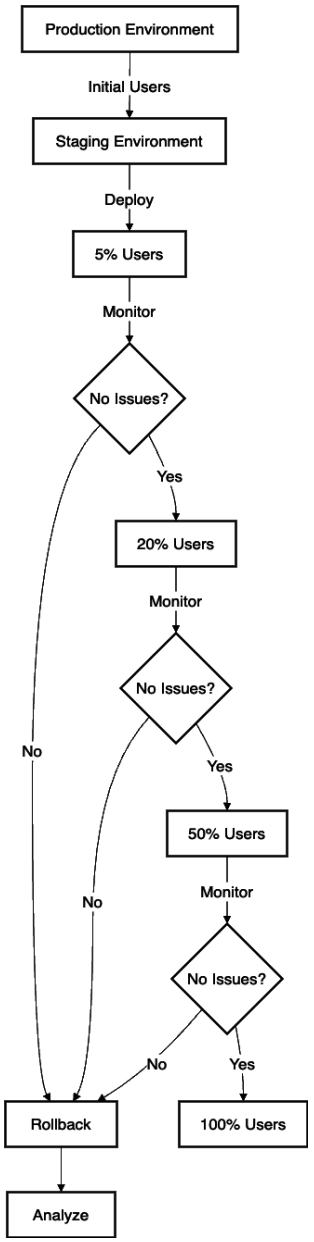

Figure 1.11 – A rolling deployment

In this figure, you can see a rolling deployment; throughout, there are steps to monitor how the new version of the application is doing, after every analysis. Depending on failures, it can be rolled back or rolled out as the new production version. Let's understand this with an example. Suppose you

want to add a new feature to your e-commerce website that allows users to rate and review products. To ensure a smooth deployment, you can use a rolling deployment strategy. This involves deploying the new feature to a small percentage of your servers or users first and observing how it performs. If everything goes well, you can gradually increase the percentage until all your servers or users have the new feature. However, if you encounter any issues, you can quickly revert to the previous version or fix the issues before continuing with the deployment.

Rolling deployments are best suited when new code is released in an incremental roll-out. Old code is phased out as new code takes over.

Another well-known deployment strategy is a blue-green deployment; let's take a look at it.

Blue-green (red-black) deployments

This deployment strategy involves creating two identical environments, one running the old version of the software (blue) and the other running the new version (green). Once the new version is ready and tested, the traffic is switched from the blue environment to the green one, resulting in a fast and reliable deployment with minimal downtime and rollback risk. However, this strategy requires more resources and infrastructure to maintain two environments. When the new version is tested and verified, the traffic is redirected to the green or black environment, making it the new active one. The blue or red environment can then be used for further testing or rollback if needed. The next figure provides a schematic view of a blue-green deployment scenario:

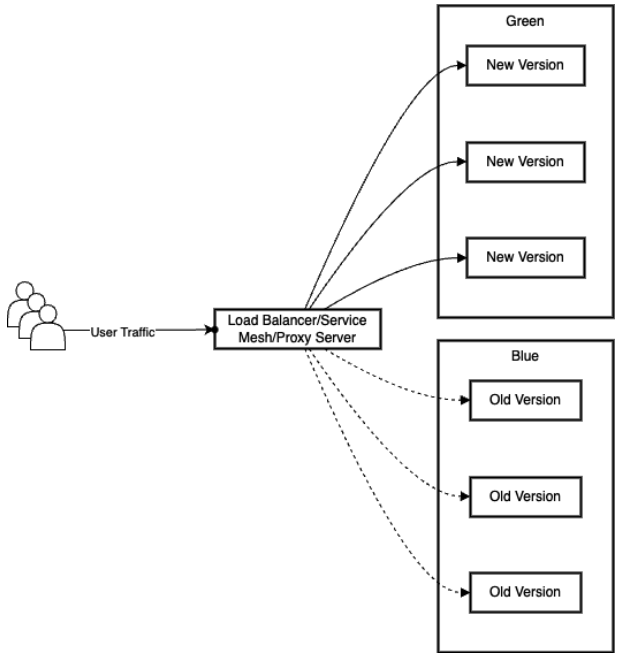

Figure 1.12 – A blue-green deployment

This figure shows you how user traffic will be redirected from the old (blue) version to the new (green) version of the application, with an option to roll back everything to the blue version if the green version fails.

In order to utilize this method, a load balancer, service mesh, or proxy server is required that can direct the traffic to either environment, depending on a configuration switch. Additionally, you must establish a way to synchronize the data between the environments. This can be achieved through the use of a shared database or replication mechanism.

Some of the benefits of using blue-green deployments are as follows:

- **Zero downtime**: The switch between the environments is instantaneous and does not require any downtime for the application.
- **Reduced risk**: If there are any problems with the new release, it is easy to roll back to the previous version by switching back to the blue environment.
- **Faster feedback**: The new release can be tested in the green environment before being switched to live traffic, which can help identify and fix any bugs or issues before they affect users.
- **Simplified deployment**: The deployment process is simplified by having two identical environments that can be switched easily, by changing the router configuration.

A use case of blue-green (red-black) deployments is when a software company wants to update its web application without causing any downtime or disruption to its users. Blue-green (red-black) deployments are a technique that allows the company to deploy two identical versions of the application, one in production (blue or red) and one in staging (green or black), and switch between them seamlessly (using a load balancer or a proxy server). This way, the company can test the new version of the application in staging, and if everything works fine, they can switch the traffic to the new version in production. If there are any issues or bugs, they can quickly switch back to the old version without affecting the user experience.

Blue-green deployments and red-black deployments are often considered to be the same strategy, with just a difference in naming. However, some sources claim that there is a slight difference between these two terms. In blue-green deployment, both versions may receive requests temporarily, while in red-black, only one of the versions gets traffic at any point in time. The difference between these two deployment strategies may depend on how the traffic is routed and how the sessions are handled between the two environments. But generally speaking, they are conceptually the same.

Next, let's take a look at the third deployment strategy, canary deployments.

Canary deployments

This deployment strategy involves launching the latest software version to a small group of users, known as *canaries*, before deploying it to the rest of the user base. The name refers to coal mining, where mineworkers used to send in a bird to see whether it was safe inside the mine. The main objective is to test the new version in a real-world situation and gather feedback from the canaries. However, there are

some possible downsides to this strategy, such as difficulties in selecting and monitoring the canaries and a risk of exposing them to bugs or errors in the new version. Despite these potential issues, this strategy reduces the risk of introducing bugs or breaking changes that might affect the entire user base.

Canary deployments also enable you to gather feedback and metrics from real users and environments, which can help evaluate the performance and usability of the new version or feature. To execute a canary deployment, an application must be able to route traffic between the old and new versions, either by using load balancers, proxies, or service meshes. The portion of traffic directed to the new version can be gradually increased until it reaches 100%, or rolled back to the old version if any issues are detected.

The following figure shows you a representation of a canary deployment scenario:

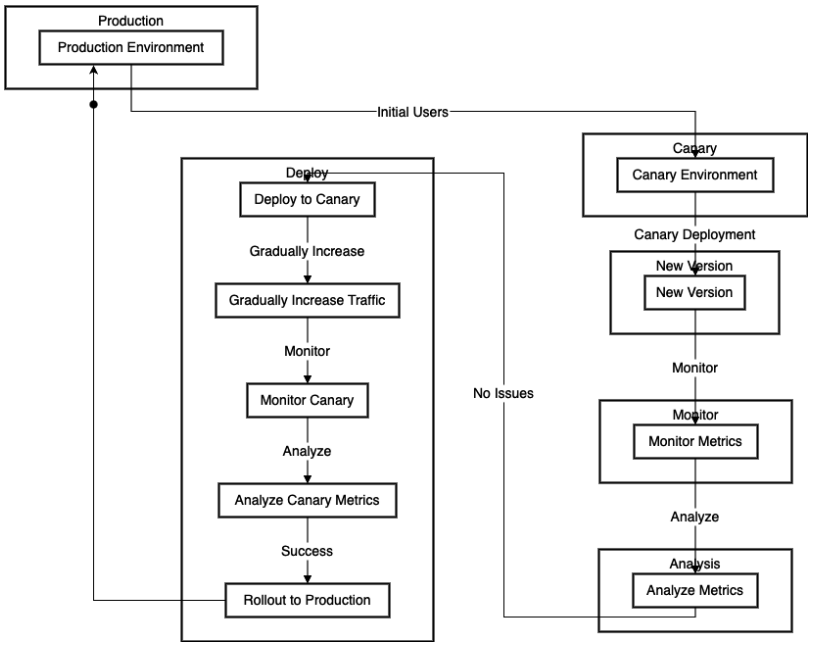

Figure 1.13 – A canary deployment

The preceding figure shows how a few users are redirected gradually to the new version, and which ones are monitored and evaluated. After a certain amount of time, when no errors occur, more users can be redirected to the new version until all use it. At that stage, the new version is promoted to production.

Some examples of use cases for canary deployments are as follows:

- Testing a new feature or functionality that might have different effects on different segments of users, such as the geographic location, device type, and browser version.
- Evaluating the impact of a performance improvement or optimization on the user behavior, engagement, or retention metrics.

- Monitoring the feedback and sentiment of users who receive the new version, adjusting or rolling back the deployment accordingly.

In this section, we learned about different deployment strategies that can help us deliver software updates to our users with minimal downtime and risk. We compared the advantages and disadvantages of blue-green, rolling, and canary deployments and how they can be implemented, using various tools and platforms. We also discussed some best practices and challenges when choosing and executing a deployment strategy that suits our needs and goals. By applying these concepts, we can improve the reliability, availability, and performance of our software systems, ensuring a positive user experience.

Summary

In this chapter, we introduced the concept of CI/CD design patterns and their benefits in designing and developing reliable, efficient, and adaptable software systems. These patterns require a shift in mindset from traditional software development approaches, as they are not just technical solutions but also cultural and operational ones. CI/CD design patterns have evolved over time, influenced by software engineering principles, agile methodologies, DevOps practices, and cloud computing.

By applying these patterns, we can achieve faster feedback loops, higher quality standards, lower costs, improved collaboration, and continuous innovation. These benefits enable us to deliver value to our customers and stakeholders more frequently and effectively, while also ensuring that our software systems are secure, scalable, and resilient.

This chapter has set the context for the rest of the book, where we will explore specific CI/CD design patterns that can help address common challenges and scenarios in software development. We will also discuss how to implement these patterns using various tools and technologies, as well as how to measure their impact and outcomes. By the end of this book, you will have a comprehensive understanding of CI/CD design patterns and how to apply them to your own projects and organizations.

In the next chapter, we will dig deeper into understanding the different types of CI/CD design patterns and components.

Further reading

Here, you can find the sources to expand your knowledge about the specific concepts in this chapter:

- *Design Patterns – Elements of Reusable Object-Oriented Software* (1994) by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.
- *A Pattern Language* (1977) by Christopher Alexander.
- *Continuous Delivery – Reliable Software Releases through Build, Test, and Deployment Automation* (2010) by Jez Humble and David Farley.
- *Strategizing Continuous Delivery in the Cloud* (2023) by Garima Bajpai and Thomas Schuetz.

2

Understanding Types of CI/CD Design Patterns and Their Components

In the previous chapter, we discovered the foundations of CI/CD, various types of design patterns in general, and a set of practices that automate the software development life cycle. In this chapter, we'll dive deeper into design patterns that apply to CI/CD. From code changes to testing to deployment, CI/CD workflows can be implemented in various ways, such as branching, feature toggles, and trunk-based development. The tools and technologies that enable these processes are known as CI/CD key components. Here are some examples:

- Version control systems (VCSs)
- Infrastructure, such as build servers

Understanding the different types of CI/CD design patterns and components available can help developers choose the best options for their projects and optimize their software quality and delivery speed.

In this chapter, we'll cover the following topics:

- Common CI/CD design patterns
- Design pattern principles – sequential and parallel execution
- Design pattern components – Pipeline as Code
- Design pattern components – Infrastructure as Code

Common CI/CD design patterns

Knowing CI/CD patterns involves knowing about the most common types of CI/CD patterns that are used in the industry. In this section, we discuss the most common types to give you a broad view of these design patterns.

Design patterns aim to simplify the arrangement of classes or objects in a software system. We'll introduce design patterns that can serve as a base for understanding design patterns and integrating key components, practices, and techniques to create an end-to-end CI/CD capability.

The build and deploy model

This model is a process that involves building the application code from the repository where the source code is stored and then running unit tests and code quality checks. Finally, the artifact is deployed to different environments using automation tools. The build and deploy model is a process that combines creating, testing, and delivering software applications to end users. The **build and deploy model** may differ depending on the type of software, the development methodology, the target platform, and the organizational culture. However, there are some common steps in the build and deploy model:

- **Coding**: The developers write the source code of the software using a programming language and tools.
- **Building**: The source code is compiled, linked, and packaged into an executable file or a set of files that can run on a specific platform.
- **Testing**: The software undergoes testing to ensure proper functioning, performance, security, and quality. The testing process involves various methods and tools, including unit testing, integration testing, system testing, acceptance testing, regression testing, load testing, stress testing, security testing, and usability testing. Developers may use tools such as code analyzers, debuggers, test frameworks, and test management tools. Testing can be done by the developers, a dedicated testing team, or external parties such as customers or users.
- **Deploying**: The software is deployed to a production environment where it can be accessed by the end users. The deployment can be manual or automated and may involve copying, installing, configuring, and updating the software.
- **Monitoring**: The software is monitored for errors, bugs, crashes, performance issues, user feedback, and usage statistics using various tools and techniques.
- **Maintaining**: The software is maintained by fixing bugs, adding features, improving performance, updating dependencies, and applying patches using steps similar to or the same as those mentioned previously.

The following figure represents the build and deploy model in practice:

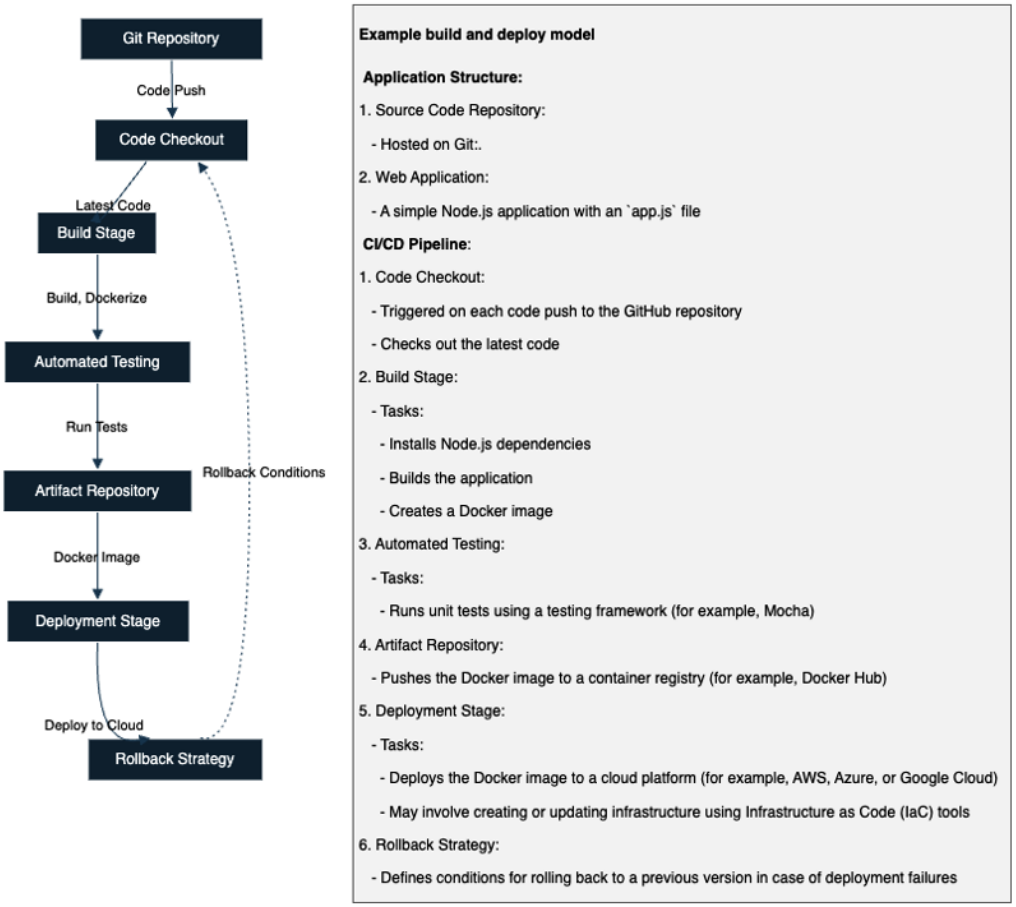

Figure 2.1 – Example of a pipeline based on the build and deploy model

This is a simplified representation; the actual CI/CD pipeline might include additional stages or tasks based on the specific requirements of the project. Each stage in the pipeline contributes to the build and deployment process, ensuring that the application is tested and deployed reliably.

In a real-world scenario, you would use CI/CD tools such as Jenkins, GitLab CI, GitHub Actions, or others to implement and automate this pipeline. Additionally, you might need to integrate with cloud services for deployment and infrastructure management purposes.

Moving on from the build and deploy model, we'll look deeper into key components that can shed light on aspects that make up a robust and efficient pipeline.

Key component – Pipeline as Code

Pipeline as Code (**PaC**) is considered a key component and is often referred to as the practice of managing the workflow of a software delivery pipeline using code. This means that instead of configuring the pipeline through a graphical user interface or command-line tool, the pipeline is written as code in a file that can be versioned, tested, and deployed along with the application code.

Here are some of the benefits of PaC:

- **Consistency and reliability**: The pipeline code can be executed in the same way every time, reducing the risk of human errors or inconsistencies.
- **Reusability and maintainability**: The pipeline code can be modularized and reused across different projects, environments, or stages. It can also be easily updated or modified as the requirements change.
- **Visibility and collaboration**: The pipeline code can be stored in a source control repository, making it visible and accessible to all stakeholders. It can also be reviewed, commented on, or approved by other developers or managers using pull requests or code reviews.
- **Automation and integration**: The pipeline code can be triggered by events such as code commits, pull requests, or scheduled tasks. It can also integrate with other tools or services using APIs or plugins.

The following are some examples of tools that support PaC:

- **Jenkins**: Allows users to define pipelines using a **domain-specific language** (**DSL**) called Jenkinsfile (Groovy)
- **GitHub Actions**: A feature of GitHub that enables users to create workflows using YAML files that run on GitHub's servers or self-hosted runners
- **Azure DevOps**: Offers various services for software development and delivery, including pipelines that can be defined using YAML files or a graphical editor
- **GitLab CI/CD**: A part of GitLab that provides CI/CD features using YAML files that describe the stages and jobs of a pipeline

Now that we've explored the key components of PaC, it's time to understand how this impacts team feedback. In the next section, we'll discuss how these components enable quicker and more efficient feedback within teams.

Faster team feedback/feedback loops

The **faster team feedback** CI/CD pattern is based on the idea that receiving feedback from various stakeholders as early and often as possible is vital for delivering software that satisfies the users' expectations and needs. By automating the testing and deployment processes, the team can decrease the time and effort required for software delivery while also minimizing the risk of errors and bugs.

The faster team feedback CI/CD pattern involves the following steps:

1. The developer writes code and commits it to a feature branch in the shared repository.
2. The code is automatically tested by a CI server, which runs unit tests, code quality checks, security scans, and other validations.
3. If the tests pass, the code is merged into the main branch and deployed to a staging environment where it can be tested by other developers, testers, or product owners.
4. If the tests fail, the developer is notified and must fix the issues before merging the code.
5. The staging environment is continuously monitored for performance, availability, and user feedback.
6. If the staging environment is stable and meets the acceptance criteria, the code is deployed to the production environment, where it can be used by real users or customers.
7. If the production environment shows any problems or negative feedback, the team can quickly roll back to a previous version or fix the issues.

Working with this method can provide numerous benefits:

- You can deliver software faster and more frequently, which increases customer satisfaction and business value. The team can detect and fix errors and bugs earlier in the development cycle, which reduces costs and improves quality.
- You can receive feedback from different stakeholders at every stage of the delivery process, which helps the team align the work with user needs and expectations.
- You can improve collaboration and communication skills so that the team works together to ensure that the software is reliable and usable.

However, there are some challenges to using the faster team feedback approach:

- You must invest in setting up and maintaining a CI/CD pipeline, which requires technical skills and resources.
- You must implement continuous improvement and a culture of adopting continuous improvement and learning, which requires a shift in mindset and organizational support.
- You must deal with frequent changes and uncertainty, which requires flexibility and adaptability.

Now that we've looked at some common patterns, let's delve into some design pattern principles.

Design pattern principles – sequential and parallel execution

This section delves into the principles of sequential and parallel execution and their correlation with design patterns. **Sequential execution** is when a program carries out one task after another in a predetermined order. **Parallel execution**, on the other hand, is when a program carries out multiple

tasks simultaneously without waiting for each other. Both types of execution have their pros and cons, which depend on the context and the problem at hand. Let's briefly look at the distinction in their application:

- **Sequential execution**: It is straightforward to understand since it follows a linear flow of logic. Besides, its outcome is consistent and predictable. However, it can be slow and inefficient, especially when you're dealing with tasks that are independent or have varying levels of priority. For instance, processing a large amount of data sequentially can take a long time to complete.
- **Parallel execution**: It's quick and efficient since it can leverage the resources of multiple processors or cores. It can also handle tasks that are independent or have varying levels of priority by assigning them to different threads or processes. However, parallel execution can be tricky and challenging to comprehend since it involves concurrency and synchronization issues. Additionally, its outcome may vary, depending on the timing and order of execution. For example, updating a shared variable in a parallel manner can lead to race conditions or deadlocks.

Design patterns can help you choose between sequential and parallel execution, depending on the problem's nature and requirements. Some design patterns are more suited to sequential execution, while others are more suited to parallel execution.

For instance, the **Chain of Responsibility** pattern is a **behavioral pattern** that enables an object to pass a request along a chain of handlers until one of them handles it. This pattern is ideal for sequential execution since it follows a linear flow of logic and avoids concurrency issues. The Chain of Responsibility pattern can be utilized to execute logging, authentication, validation, or error handling. Let's have a look at a simple but clear example of a sequential execution using a login sequence:

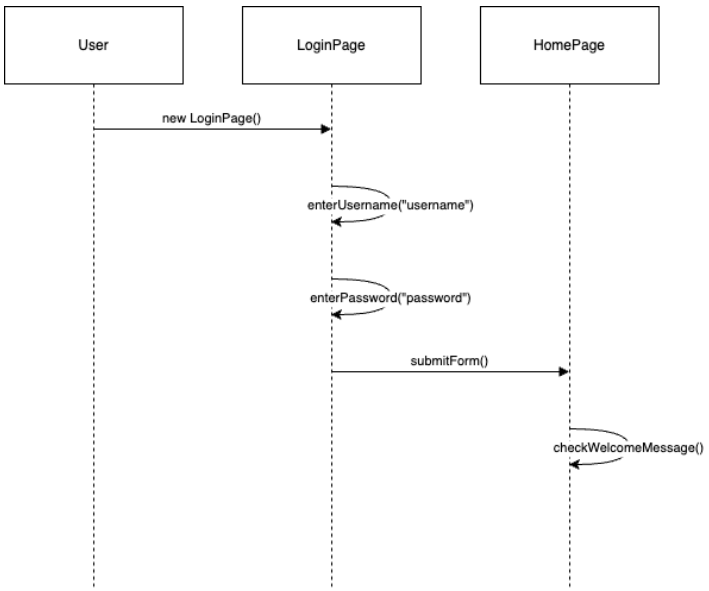

Figure 2.2 – Sequential execution of the Fluent Page Object Model pattern

In this diagram, `User`, `LoginPage`, and `HomePage` represent different components or objects in your software. The arrows represent method calls, with each call triggering the next sequentially.

Here, `User` creates a new instance of `LoginPage`, enters the username and password, submits the form, which returns an instance of `HomePage`, and then checks the welcome message in `HomePage`.

Another example is the **Executor pattern**, which is a **concurrency pattern** that disentangles the initiation and execution of tasks. This pattern is ideal for parallel execution because multiple tasks can be executed concurrently by a pool of threads or processes. The Executor pattern can be utilized to execute load balancing, caching, scheduling, or asynchronous operations. The following figures shows how this works:

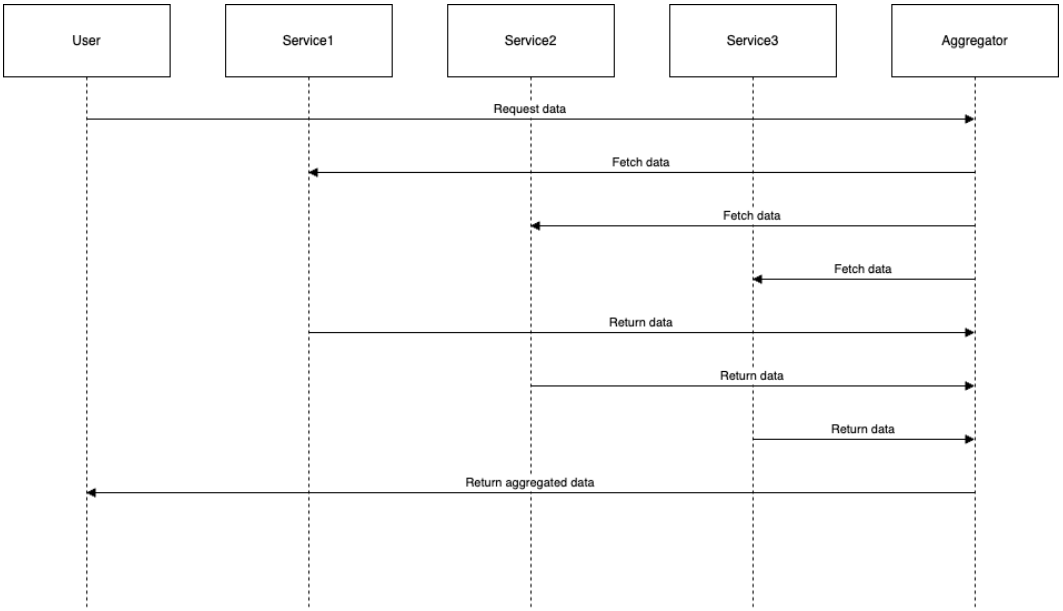

Figure 2.3– Parallel execution example flow

In this diagram, `User`, `Service1`, `Service2`, `Service3`, and `Aggregator` represent different components or objects in your software. The arrows represent method calls, with `Aggregator` starting tasks in `Service1`, `Service2`, and `Service3` simultaneously. Once the tasks are completed, `Service1`, `Service2`, and `Service3` notify `Aggregator`, which then returns the aggregated data to `User`.

Next, you'll learn how to choose between sequential and parallel execution.

How to choose between sequential and parallel execution

When designing software, it's important to choose between sequential and parallel execution based on the problem's nature and requirements. Some problems are inherently sequential with a clear order of steps, while others can be executed simultaneously, making them parallel. Most problems are a mix of both. It's essential to consider design patterns so that you can decide whether to use sequential or parallel execution since some patterns are more suitable for sequential execution and avoid concurrency issues.

Some design patterns are flexible enough to support both sequential and parallel execution, depending on the context and the configuration. For example, the **Strategy pattern** can be used to implement encryption, compression, sorting, or filtering, supporting both sequential and parallel execution.

In summary, choosing between sequential and parallel execution depends on the nature and requirements of the problem, and the design patterns we want to use. We need to consider the trade-offs between speed and complexity and use the appropriate design patterns to implement our solution. It's important to know about these design patterns and make the right choice for your scenario or use case.

Now that we've explained design patterns principles, it's good to have a look at which components serve these principles.

Design pattern components – Pipeline as Code

PaC is a specific component of design patterns that utilizes a methodology for creating a **continuous integration** (**CI**) pipeline. The pipeline is defined entirely in computer code and stored in version control as a single script or program code, which can be executed with a single command line. It allows developers to define and execute CI and delivery pipelines using code, rather than graphical user interfaces or configuration files.

PaC has several advantages. It allows for version control and collaboration on pipeline definitions, making maintenance and review easier. It also enables the reuse and sharing of pipeline components such as stages, steps, and actions across different projects or teams. Additionally, it simplifies the process of testing and debugging pipelines and facilitates automated validation and verification of pipeline quality and compliance. PaC also supports dynamic and complex pipeline logic, such as conditional branching, looping, parallelism, and error handling.

PaC has many benefits for CI pipelines:

- **Version control**: The code that defines the pipeline can be tracked, reviewed, and rolled back using version control tools. This ensures that the pipeline is always consistent and reliable and that any changes are documented and auditable.
- **Reusability**: The code that defines the pipeline can be modularized and shared across different pipelines or projects. This reduces duplication and improves consistency and quality. For example, common stages, steps, or actions can be extracted into functions or libraries and reused in multiple pipelines.

- **Testability**: The code that defines the pipeline can be tested and debugged using code analysis tools or testing frameworks. This helps with identifying and fixing errors or bugs in the pipeline logic before it's deployed to production. It also enables automated validation and verification of the pipeline's functionality and compliance with best practices or standards.
- **Flexibility**: The code that defines the pipeline can support complex and dynamic pipeline logic, such as conditional branching, looping, parallelism, and error handling. This allows the pipeline to adapt to different scenarios or requirements, such as different environments, branches, or triggers.

PaC is a powerful component for design patterns as you can create and manage CI pipelines using code. It offers many advantages over traditional methods of defining pipelines using graphical tools or configuration files. It also requires some skill and practice to ensure that the code is readable, maintainable, and secure. In the next section, we'll look at the implementation side of PaC.

How do you implement PaC?

To start using PaC, you need to choose a tool that supports it. There are many tools available, such as Jenkins, GitLab CI, Travis CI, and more. Each tool has its own syntax and features for defining pipelines as code. You also need to set up a VCS, such as Git, so that you can store and manage your pipeline code. You can use any VCS that works with your chosen tool.

Once you've chosen a tool and a VCS, you can start writing your pipeline code. The following subsections will discuss some of the basic steps for several solutions using various tools.

Jenkins

Jenkins is a very popular open source automation server that helps developers build, test, and deploy software projects. It supports CI/CD, which are practices that aim to improve the quality and speed of software development. Jenkins can run on various platforms, such as Windows, Linux, macOS, and Unix, and can integrate with many tools, such as VCSs, testing frameworks, code analysis tools, and cloud services. Jenkins is written in Java and can be easily installed and configured via its web interface. Jenkins also has a plugin architecture to extend its functionality with numerous plugins available in the Update Center.

When you want to implement PaC in Jenkins, you need to follow these basic steps:

1. Create a file called `Jenkinsfile`. It should be located in the root directory of your project.

   If you want to automate the creation of the `Jenkinsfile` file itself, you can use one of two things:

   - **The template-based approach**: Create a template `Jenkinsfile` that includes placeholders for project-specific information. Then, use a script or tool to replace these placeholders with actual values based on your project.

    - **Pipeline domain-specific language (DSL)**: The pipeline DSL allows you to define Jenkins jobs programmatically. You can use this plugin to create and configure Jenkins jobs, including pipeline jobs.

2. Write the code that defines the stages, steps, and actions of your pipeline. The code should follow the syntax and conventions of your chosen tool. For example, in Jenkins, the code is written in Groovy and it uses a declarative or scripted style.
3. Commit and push your pipeline code to your VCS. This will cause your tool to run your pipeline according to the code you wrote.
4. Monitor and troubleshoot your pipeline execution using the graphical interface or logs provided by your tool. You can also make changes to your pipeline code and push them to your VCS to update your pipeline.

To achieve the highest level of automation, almost every step or action should be included so that you can rule out any manual intervention. By following the preceding steps, you'll come close to that level.

Gitlab Runners

GitLab Runners are agents that execute your pipeline jobs on a machine of choice. They can be used to run your pipelines on different operating systems, architectures, and environments, depending on your needs and preferences. This can be configured by adjusting `config-toml`, which is located on the Gitlab Runners machine.

Some of the most common configuration options are as follows:

- **Concurrent**: The number of jobs that can run concurrently on the runner.
- **Limit**: The maximum number of jobs that can be assigned to the runner.
- **Executor**: The method that's used to run the jobs on the runner. There are different types of executors, such as `shell`, `docker`, `ssh`, and others.
- **Environment**: A list of environment variables that are set for all jobs on the runner.
- `pre_clone_script`, `pre_build_script`, `post_build_script`: Scripts that are executed before or after you clone the repository or run the job.

You can assign tags to runners that specify which runner runs which job. For example, if you have a runner with the `linux` tag, you can use the tag in your job definition to ensure that only that runner can run that job.

Azure Pipelines (from Azure DevOps)

For Azure Pipelines, PaC enables you to version control, review, and test your pipeline configurations. It also allows you to reuse and share them across different projects and teams. PaC automates the creation and deployment of your pipelines using the Azure CLI or REST API. This gives you more

control and flexibility over your Azure Pipelines workflows. Within Azure Pipelines, you can use YAML to specify the stages, jobs, steps, and tasks that your pipeline should execute, as well as the triggers, variables, parameters, and conditions that affect its behavior.

To use PaC with the Azure CLI, you need to install the Microsoft Power Platform CLI. This is a developer CLI that empowers you to perform various operations in Microsoft Power Platform related to your environment life cycle, authentication, Dataverse environments, solution packages, portals, code components, and more. You can install the Power Platform CLI using Visual Studio Code or PowerShell. To update the Power Platform CLI to the latest version, you can use the `pac install latest` command.

Once you've installed the Power Platform CLI, you can use the `pac solution` command group to work with Dataverse solution projects. For instance, you can use `pac solution clone` to create a solution project based on an existing solution in your organization, `pac solution pack` to package solution components on the local filesystem into a `solution.zip` file, `pac solution import` to import the solution into Dataverse, and `pac solution export` to export a solution from Dataverse. You can also use `pac solution check` to upload a Dataverse solution project to run against the Power Apps Checker service.

Implementing PaC in Azure Pipelines involves performing the following steps:

1. Create a YAML file that contains a pipeline definition, then define stages, jobs, and steps.
2. Use parameters and variables for flexibility.
3. Set triggers to run the pipeline automatically.
4. Use templates for common sets of tasks.
5. Manage artifacts and use environments for targeted deployments.
6. Lastly, review and run the pipeline to ensure it's working as expected.

By implementing PaC in Azure Pipelines, you can create a flexible, automated, and reusable CI/CD process that manages artifacts and uses targeted deployments, ensuring your pipeline works as expected.

Travis CI

Travis CI is a well-known cloud-based service that facilitates CI/CD for a wide range of programming languages and platforms. To establish PaC for Travis CI, you must create a file called `.travis.yml` in the main directory of your project repository. This file comprises instructions on how to build, test, and deploy your code using Travis CI. In this file, you can specify the language, environment, dependencies, scripts, stages, jobs, notifications, and other options for your pipeline. Moreover, you can make use of YAML anchors, aliases, and references to avoid repetition and simplify your configuration. For instance, you can set up a common set of environment variables or commands that apply to multiple jobs or stages. Additionally, you can leverage matrix expansion to create various versions of a job with different parameters.

Let's look at some of the capabilities of Travis CI. It's important to highlight its robust features as they streamline the software development process and enhance productivity:

- **Version control**: You can store your `.travis.yml` file in the same repository as your source code, and track its changes over time. This way, you can ensure that your pipeline is always in sync with your code and that you can easily revert or roll back to a previous version if needed.
- **Collaboration**: You can share your `.travis.yml` file with other developers, and use pull requests and code reviews to improve its quality and functionality. You can also use branching and merging strategies to manage different versions of your pipeline for different environments or features.
- **Automation**: You can automate the creation and execution of your pipeline, and integrate it with other tools and services that you use in your development process. For example, you can trigger your pipeline when you push code to GitHub, or when you create a pull request. You can also use webhooks or APIs to communicate with other platforms, such as Slack, Jira, or AWS.
- **Customization**: You can customize your pipeline to suit your specific needs and preferences, and use the features and options that Travis CI provides. For example, you can use different languages, frameworks, or tools for your jobs, or run them in parallel or sequentially. You can also use different environments or operating systems for your stages, or run them on different machines or containers.

Now that we've explored the capabilities of Travis CI, let's transition to another powerful tool in the realm of CI – GitHub Actions.

GitHub Actions

You can use PaC with GitHub Actions by creating a YAML file that defines the workflow. The YAML file specifies the events that trigger the workflow, the jobs that run in parallel or sequentially, and the steps that each job performs. You can also use variables, expressions, and secrets to customize your workflow.

To get started, you need to store the YAML file in your repository, under the `.github/workflows` directory, or in a separate repository that you reference from your main repository. This way, you can keep your workflow organized and version-controlled.

By using PaC with GitHub Actions, you can ensure that your workflows are consistent, repeatable, and shareable. You can easily modify and update your workflows to suit your project's needs, making it a great tool for automating your software development process.

Here are some best practices for using PaC for GitHub Actions:

- Use descriptive names for your workflows, jobs, and steps to make them easy to understand and maintain.

- Use self-contained actions or containers whenever possible to avoid dependencies on external tools or environments.
- Use secrets to store sensitive data such as passwords, tokens, or keys, and avoid exposing them in your workflow code or logs.
- Use environments to define protection rules and approval policies for your workflows, especially when deploying to production or critical systems.
- Use the matrix strategy to run your workflows on multiple combinations of operating systems, programming languages, or software versions.

Having discussed the features of GitHub Actions, let's shift our focus to another significant player in the CI/CD arena – Bamboo.

Bamboo

Bamboo is a product of Atlassian that provides support for defining pipelines as code using Bamboo Specs. Bamboo Specs is a DSL that enables you to define your build and deployment plans programmatically. You can use either YAML or Java to create your Bamboo Specs files. You can store these files in the same repository as your application code, which helps you to version control your build plans and keep them synchronized with your application code.

PaC for Bamboo Atlassian provides numerous benefits, including simplifying the management of build configurations and minimizing human errors. It enhances the visibility and traceability of build changes and history, enabling faster feedback and delivery cycles through automated build plan creation and updates. It also supports collaboration and promotes best practices among developers and teams.

To enable PaC in Bamboo's settings, follow these steps:

1. Go to **Administration | System | Pipelines as Code** and check the **Enable Pipelines as Code** box.
2. Locate the root directory of your Git repository and create a YAML file named `.bamboo.yml`. This file should contain the definition of your build plan; please refer to the Bamboo documentation for the correct syntax and structure.
3. Push the YAML file to your Git repository. Bamboo will detect the file and automatically create or update the corresponding build plan.
4. If you wish to customize your build plan further, you can use variables, triggers, conditions, and notifications in your YAML file. You can also use multiple YAML files to define different stages or branches of your build plan.

> **Tip**
> Your YAML file can be further customized using variables, triggers, conditions, and notifications. You can define different stages or branches of your build plan by using multiple YAML files.

There are some extra considerations to think about. Version control is essential to keep track of changes and enable collaboration when working with Bamboo Specs code. With the Bamboo Specs documentation, you can refer to detailed information on the available DSL elements and configuration options so that you can manage your build and deployment plans effectively. Bamboo Specs provides customizable options so that you can tailor your pipeline to meet specific requirements. You can leverage the integration of Bamboo with other Atlassian products such as Jira and Bitbucket to enhance your pipeline's functionality and efficiency.

As we wrap up this section, we've seen how it streamlines the software development process. Now, let's turn our attention to another crucial aspect of design patterns – **Infrastructure as Code** (**IaC**). This approach allows us to manage and provision our technology stack efficiently.

Design pattern components – Infrastructure as Code

IaC is another important component of design patterns that refers to the process of managing and provisioning IT resources using human-readable configuration files instead of relying on manual hardware configuration or interactive configuration tools. IaC aims to automate the deployment, configuration, and maintenance of IT infrastructure, reducing human error and increasing efficiency and reliability.

One of the main benefits of IaC is that it enables the use of design patterns. It can help improve the quality, scalability, and performance of IT systems, as well as facilitate collaboration and communication among developers and operators.

The components and design patterns contribute to creating a robust, scalable, and maintainable infrastructure solution through IaC practices. The specific tools and implementations may vary based on the IaC framework you're using (Terraform, AWS CloudFormation, Ansible, and so on) and the cloud provider or platform you're working with.

In terms of IaC design pattern components, we can identify the following:

- **Declarative configuration files**: Use declarative configuration files to define the desired state of the infrastructure without specifying the step-by-step procedure to reach that state. This makes your code more readable, maintainable, and portable. Some of the common formats for configuration files are YAML, JSON, and **HashiCorp Configuration Language** (**HCL**).
- **Modules**: Break down infrastructure configurations into modular components. Modules encapsulate specific functionality or resources and are reusable in various parts of the infrastructure. This helps you avoid duplication, reduce complexity, and promote reusability. Some of the tools that support modules are Terraform Modules and AWS CloudFormation Nested Stacks.

- **Parameterization**: Parameterize configurations to make them more flexible and reusable. Parameters allow you to customize configurations for different environments or use cases. For example, you can use parameters to specify the region, instance type, or network settings for your infrastructure. Some of the tools that support parameterization are Terraform Variables and AWS CloudFormation Parameters.
- **Conditional logic**: Incorporate conditional statements to handle different scenarios or environments within the infrastructure code. For example, you can use conditional logic to create different resources based on the environment (dev, test, and prod) or the platform (Windows, Linux, and so on). Some of the tools that support conditional logic are Terraform Conditional Expressions and AWS CloudFormation Conditions.
- **Resource tagging**: Implement consistent resource tagging to add metadata to resources, aiding in organization, cost tracking, and resource management. For example, you can use tags to group resources by project, owner, or purpose. Some of the tools that support resource tagging are Terraform Resource Tags and AWS CloudFormation Resource Tags.
- **Version control integration**: Store infrastructure code in VCSs to track changes, collaborate with teams, and enable rollbacks. VCSs also allow you to implement workflows such as branching, merging, and pull requests. Some of the popular VCSs are Git, GitHub, and GitLab.
- **Secrets management**: Separate sensitive information (such as passwords or API keys) from the infrastructure code. Use secure methods for managing and accessing secrets. For example, you can use encryption, hashing, or vaults to store secrets and use environment variables or APIs to access them. Some of the tools that support secrets management are HashiCorp Vault, AWS Secrets Manager, and Azure Key Vault.

We discussed some of the most common design patterns in the *An overview of design patterns* section in *Chapter 1*, but we can extend the list with some additional ones since they're important for us to get a clear view of design patterns in IaC:

- The **Factory Method pattern**, something we also slightly touched on in *Chapter 1*, defines an interface for creating an object but lets the subclasses decide which class to instantiate. This allows different types of infrastructure resources to be created based on different parameters or environments. For example, with a factory method and a tool such as Terraform, you can create a cloud instance or a local instance, depending on the availability and cost of the resources. Please be aware that while this example follows the spirit of the Factory Method pattern, Terraform is a declarative language and doesn't support the same level of abstraction and encapsulation as object-oriented programming languages.

We can see this method applied in IaC in the following diagram:

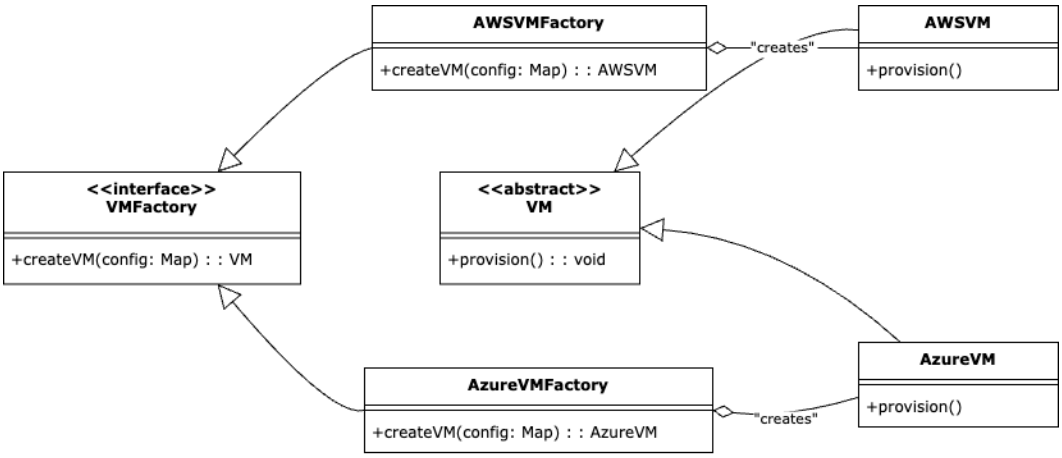

Figure 2.4 – The Factory Method pattern applied in IaC

This diagram describes a scenario where you have two different cloud providers (AWS and Azure), and you want to deploy VMs using Terraform. Depending on the cloud provider that's chosen, Terraform will use different modules or configurations. Here are the components we can see:

- `VMFactory`: This is a conceptual interface that defines the `createVM` method. In Terraform, this would correspond to different modules or configurations.
- `AWSVMFactory` and `AzureVMFactory`: These represent the cloud-specific factories that would use Terraform modules or scripts to deploy VMs on AWS and Azure, respectively.
- `VM`: An abstract class representing a generic **virtual machine** (**VM**) resource.
- `AWSVM` and `AzureVM`: Concrete classes (or Terraform modules) representing specific VM deployments on AWS and Azure.

- The **Builder pattern** separates the process of constructing a complex object from its representation. This allows for the creation of different representations of the same infrastructure resource, such as a VM or a container. A builder can create a VM using Ansible scripting, creating a playbook with different operating systems, memory sizes, or disk types depending on the requirements of the application. It follows the spirit of the Builder pattern, but it's not object-oriented. Let's look at an example that uses this Ansible scripting:

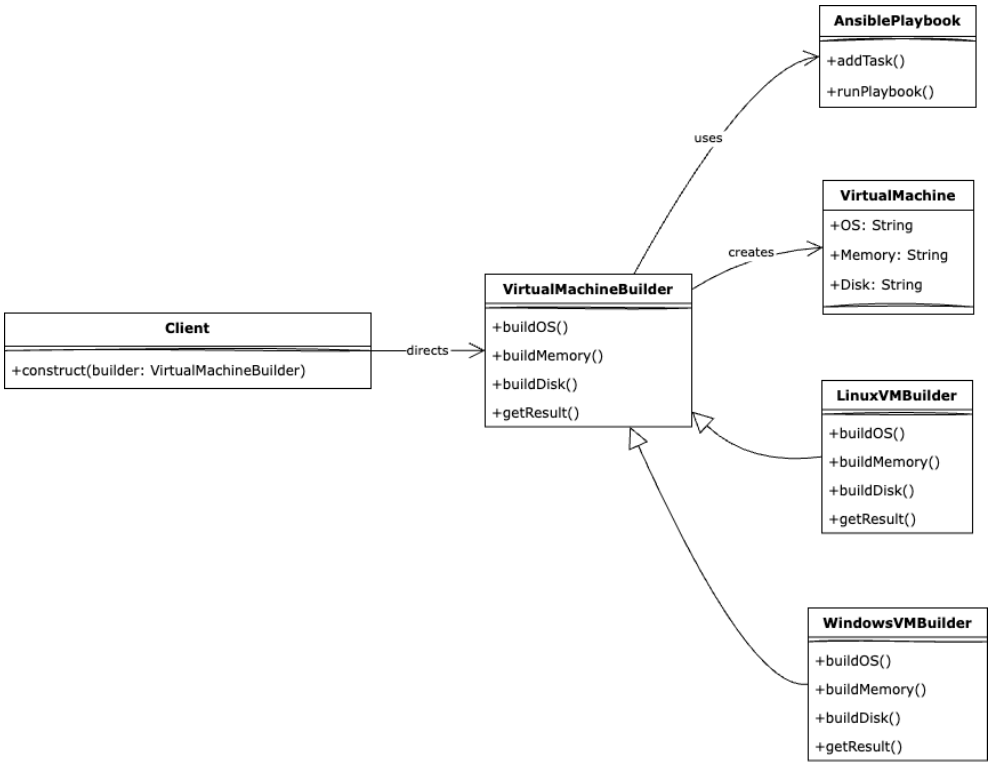

Figure 2.5 – The Builder pattern in IaC

In this example, we have the following components:

- `VirtualMachineBuilder`: This is the builder interface with methods to build different parts of the VM.
- `AnsiblePlaybook`: This represents the Ansible scripting that adds tasks and runs the playbook.
- `VirtualMachine`: This is a complex object being built.
- `LinuxVMBuilder` and `WindowsVMBuilder`: These are concrete builders that implement the `VirtualMachineBuilder` interface to create specific configurations.
- `Client`: This is responsible for constructing the VM using the builder.

This follows the spirit of the Builder pattern, allowing for the creation of different representations of the same infrastructure resource.

- The **Adapter pattern** converts the interface of a class into another interface that clients expect. This allows us to integrate different infrastructure components that have incompatible interfaces, such as cloud services or legacy systems. An adapter can provide a common interface by using

a tool such as Chef for accessing different storage services, such as Amazon S3 or Google Cloud Storage. We can see this pattern in the following diagram:

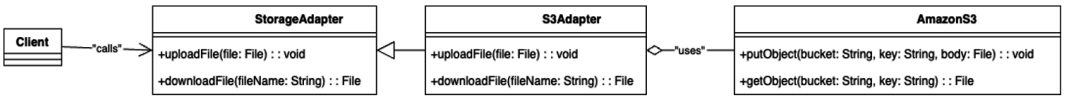

Figure 2.6 – The Adapter pattern in IaC

The idea is to create an adapter that allows a client to interact with Amazon S3 consistently, abstracting the details of the S3 API. The following components have a role to play here:

- `StorageAdapter`: This is an interface that consists of the `uploadFile` and `downloadFile` methods.
- `S3Adapter`: A concrete class that implements `StorageAdapter`. It translates the `uploadFile` and `downloadFile` calls to Amazon S3 API methods.
- `AmazonS3`: This represents the actual Amazon S3 service and its API methods, such as `putObject` and `getObject`.
- `Client`: This represents the application or script that needs to upload or download files while using the `StorageAdapter` interface to interact with Amazon S3.

- The **Facade pattern** provides a simplified interface to a complex system of interfaces. This allows us to simplify the interaction between clients and a complex infrastructure system, such as a network or a database. For example, a facade can provide a single method using Puppet for performing common operations on a database, such as querying, inserting, or updating data. Let's look at a simple example of a CI/CD scenario that uses Puppet to manage database operations such as querying, inserting, or updating data:

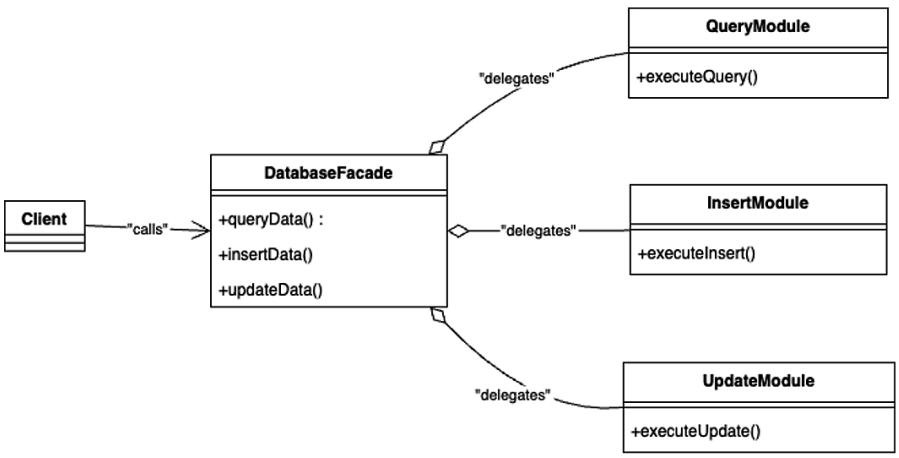

Figure 2.7 – The Facade pattern in a simple scenario

The following components are involved:

- `DatabaseFacade`: It acts as the facade, providing a simplified interface for common database operations (`queryData`, `insertData`, and `updateData`).
- `QueryModule`: This handles the execution of database queries.
- `InsertModule`: This inserts data into the database.
- `UpdateModule`: This handles updating existing data in the database.
- `Client`: This represents the CI/CD pipeline or a script that interacts with `DatabaseFacade` to perform database operations.

- The **Observer pattern** establishes a one-to-many relationship between objects. When one object changes state, all its dependents are automatically updated. This allows us to synchronize and coordinate infrastructure resources that depend on each other, such as load balancers or web servers. For example, an observer can notify all the web servers using Terraform when the load balancer changes its routing policy or when a new server is added or removed.

  Here's how the Observer pattern could be applied in a real-life CI/CD scenario using Terraform to manage infrastructure resources such as load balancers and web servers:

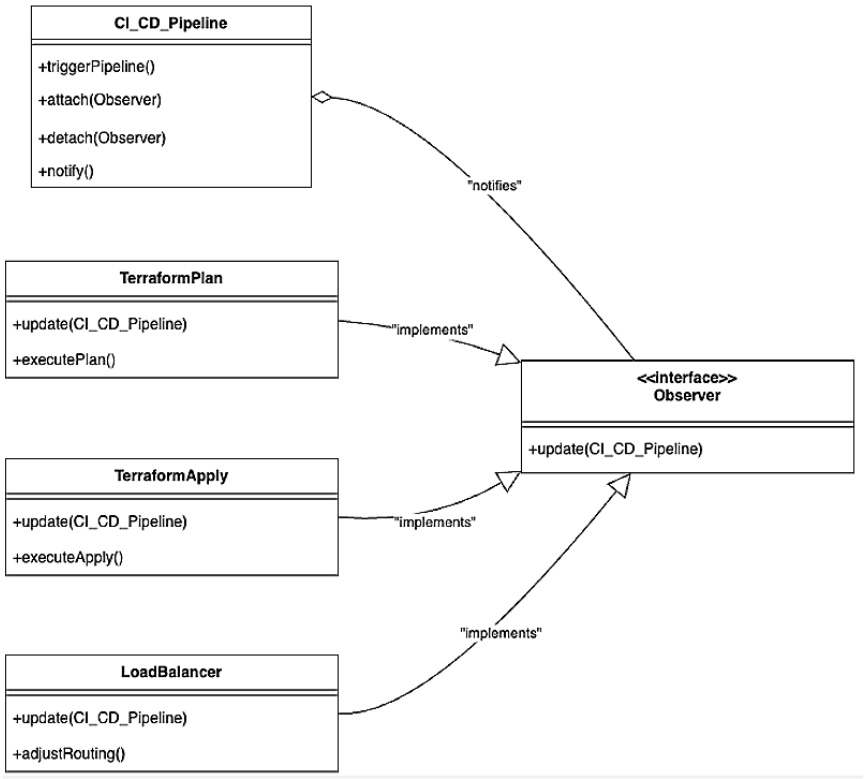

Figure 2.8 – Observer pattern example

Let's look at the components in this diagram:

- `LoadBalancer`: The subject that maintains a list of observers (web servers) and notifies them when its state changes, such as when a routing policy is updated.
- `Observer`: An interface that defines the `update()` method, which will be implemented by the `WebServer` class.
- `WebServer`: This represents the observers that implement the `update()` method to perform actions when notified by `LoadBalancer`.
- `Client`: This represents the CI/CD pipeline or script that configures `LoadBalancer`.
- `LoadBalancer`: Configured in `load_balancer.tf`, this resource represents the load balancer that might change its routing policy. The life cycle and dynamic blocks ensure that changes notify observers (web servers).

It might be interesting to explore how a theoretical construct, such as the Object pattern, is implemented in a real-world scenario using a well-known tool such as Argo CD. We'll demonstrate the practical application of this concept in this context.

Argo CD operates as a declarative continuous delivery tool specifically designed for Kubernetes. It enables you to manage deployments using GitOps, where Git serves as the single source of truth for the desired state of your applications. By adhering to this model, Argo CD ensures that any changes to the application's state in Git are automatically reflected in the Kubernetes cluster.

Moreover, the Observer pattern can be modeled effectively within Argo CD by defining separate application resources. These resources are responsible for observing changes in a specific Git repository and responding accordingly to ensure that the desired state is always maintained.

To provide a clearer understanding of this process, here's a step-by-step breakdown of a typical Argo CD flow:

1. **Git repository structure**: The journey begins with a well-organized Git repository, where your Terraform files and Kubernetes manifests are stored. Different stages of the deployment process – such as Terraform Plan, Terraform Apply, and load balancer adjustments – are represented by distinct folders or branches within this repository.
2. **Argo CD applications**: Each stage in the pipeline, such as Terraform Plan, Terraform Apply, and load balancer configuration, is represented by an Argo CD application resource. These applications are set up to observe changes in the corresponding Git repository and apply those changes to the Kubernetes cluster, ensuring that the cluster's state aligns with the declared state in Git.

Example Argo CD applications (the Observer pattern)

The application which is used as an example is as follows:

- **Terraform Plan application**: This application watches the `terraform-plan` folder or branch and runs Terraform Plan:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: terraform-plan
spec:
  project: default
  source:
    repoURL: 'https://github.com/your-org/your-repo'
    targetRevision: HEAD
    path: terraform/plan
  destination:
    server: 'https://kubernetes.default.svc'
    namespace: terraform
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

- **Terraform Apply application**: This application watches the `terraform-apply` folder or branch and runs Terraform Apply:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: terraform-apply
spec:
  project: default
  source:
    repoURL: 'https://github.com/your-org/your-repo'
    targetRevision: HEAD
    path: terraform/apply
  destination:
    server: 'https://kubernetes.default.svc'
    namespace: terraform
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

- **Load Balancer adjustment application**: This application watches the configuration of the load balancer and applies the necessary adjustments:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: load-balancer-adjustment
spec:
  project: default
  source:
    repoURL: 'https://github.com/your-org/your-repo'
    targetRevision: HEAD
    path: infrastructure/load-balancer
  destination:
    server: 'https://kubernetes.default.svc'
    namespace: loadbalancer
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

  This piece of code of the Argo CD application monitors the specified Git repository's `terraform-plan` folder or branch and automatically runs Terraform Plan to ensure the Kubernetes cluster's state matches the desired configuration.

Now, let's have a look at how the Observer pattern fits in this Argo CD flow:

1. **Argo CD monitoring**: Each Argo CD application monitors a specific path or branch in your Git repository. When changes are detected, it automatically applies those changes to the Kubernetes cluster.
2. **Observer pattern**: The Terraform Plan, Apply, and load balancer adjustment are analogous to observers. When the Git repository changes (the subject), each observer (Argo CD application) is notified and updates its corresponding part of the infrastructure.
3. **GitOps workflow**: By adopting GitOps, you maintain a versioned history of your infrastructure changes. Argo CD ensures that the live state of the infrastructure matches the desired state in Git, effectively implementing the Observer pattern within a CI/CD context.

Looking at this, we have a pretty good idea about how patterns like these can be implemented in your daily CI/CD operations.

As we conclude our exploration of design pattern components (IaC), we've seen how it revolutionizes the management and provisioning of technology stacks. In the following chapter, we'll transition to another integral component of design patterns: **test automation**. This plays a pivotal role in enhancing the efficiency and reliability of software development. We'll follow this with some other very interesting topics.

Summary

CI/CD design patterns are essential for achieving high-quality software delivery in a fast and reliable way. By understanding the different types of CI/CD design patterns and their components, you can choose the best solution for your specific needs and goals. In this chapter, we explored the different types of CI/CD design patterns and components that can be used to create a robust and scalable software delivery pipeline. First, we learned about the benefits and challenges of each pattern. We also discussed the key components of a CI/CD pipeline, such as source code management and build automation. By understanding these concepts, we can choose the best practices and tools for our specific needs and goals. In the next chapter, we'll continue to explore other implementations of CI/CD patterns.

3

Advancing on CI/CD Design Patterns – from Testing to Deployment

Following the previous chapter, we're going to continue our journey of discovering design patterns that apply to CI/CD. This chapter looks at the stage of testing to deployment, where CI/CD workflows can be implemented in various ways, such as via branching, feature toggles, and trunk-based development. The tools and technologies that enable these processes are known as CI/CD key components, which include the following:

- Testing frameworks
- Deployment tools

Understanding the different types of CI/CD design patterns and components available can help developers choose the best options for their projects and optimize their software quality and delivery speed.

In this chapter, we'll cover the following topics:

- Design pattern components – test automation
- Design pattern components – artifacts
- Design pattern components – deployment types
- Industry best practices and challenges

Design pattern components – test automation

Test automation involves using specialized tools to perform various testing tasks on an application. These tasks can include functional testing, regression testing, performance testing, and other forms of testing that are required to ensure that the software functions as intended. Test automation can

reduce the manual effort involved in testing, increase test coverage, and improve the quality and reliability of software products.

An essential feature of test automation is that it allows software development teams to carry out thorough and rigorous testing without requiring extensive manual effort. This helps with identifying bugs, performance issues, and other problems that may not be immediately apparent during manual testing. Additionally, test automation can help increase the overall test coverage and ensure all aspects of the software are tested and any issues are identified and resolved.

Another key feature of test automation is improving the quality and reliability of software. By allowing testing to be carried out more efficiently and accurately, test automation can ensure that software products are delivered to customers with minimal defects and issues. Customer satisfaction will increase and the risk of costly product recalls or reputational damage resulting from faulty software is reduced.

Overall, test automation is a critical component of software development that can help improve software products' efficiency, quality, and reliability.

Let's have a look at a few of the commonly used patterns:

- **Data-driven testing**: A popular approach in software testing is to separate the test data from the test logic or scripts. This is done by designing test scripts that read test parameters and input data from external sources such as databases, XML files, Excel sheets, JSON files, or CSV files. By following this strategy, automation engineers can implement a solitary test script that's capable of executing tests for all data present in a table. This advanced technique of testing automation not only reduces the time and effort required for testing but also ensures that all test data within the table is tested comprehensively and systematically. This saves time and effort as well as makes the testing process more efficient. The following figure shows a simple view of data-driven testing:

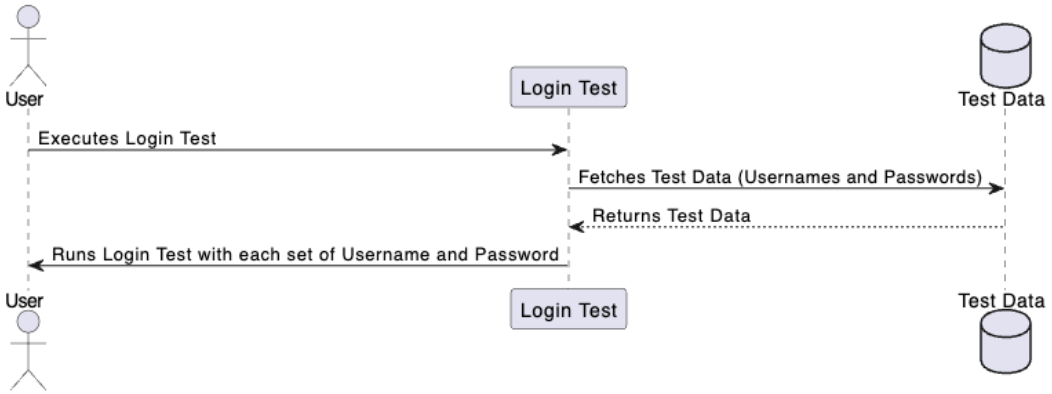

Figure 3.1 – Example of a login functionality with multiple sets of data using the data-driven testing approach

This shows different test cases. The *username* and *password* are the inputs for the test, and the *result* is obtained from the test data.

In this case, you would write a test script that loops over each row in the data, inputs the *username* and *password* into the login function, and checks whether the result matches the *expected result*.

- **Retry testing**: An application can be enabled to handle temporary failures that may occur when connecting to a service or network resource. This is done by automatically retrying a failed operation, which can improve stability. Such situations aren't uncommon in cloud environments. However, it's important to monitor the retry metrics to prevent endless retries and limit the impact of retry storms on failed or recovering services. This can be seen in the following figure:

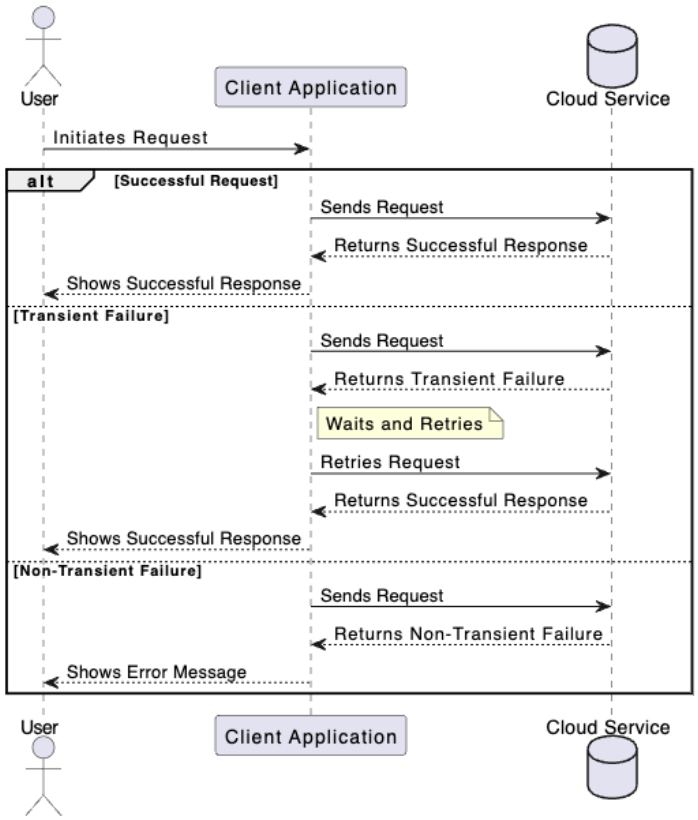

Figure 3.2 – Example of the Retry pattern in a cloud environment

In the preceding figure, the Retry pattern is shown in the form of the client application retries with temporary failures, and finally shows the successful response from the cloud service.

- **The Factory Design pattern**: The Factory Design pattern is a creational pattern that allows you to create objects without having to specify their concrete classes. It provides an interface for creating objects and permits a class to postpone instantiation to its subclasses. In the context of test automation, the Factory Design pattern can be utilized to create various types of test objects based on specific rules. For example, if you have multiple classes that implement an interface and you don't want to specify which concrete class to instantiate in your code, you can use the Factory Design pattern to create the object that suits your needs.
- **Page Object Model**: This is a design pattern that helps with creating reusable and maintainable code for web automation testing. It allows us to separate the test logic from the web elements and group the web elements by pages or components. By using the Page Object Model, testers can write more readable and robust test scripts, as well as reduce code duplication and maintenance efforts. For example, if a web page has a login form, you can create a `Login Page` object that contains the web elements for the username, password, and login button, as well as the methods to interact with them. Then, we can use this `Login Page` object in our test scripts, without having to write the same code for finding and manipulating the web elements every time.
- **The Facade pattern**: The Facade pattern simplifies complex system interfaces and hides details and dependencies. It reduces coupling, improves readability, and makes maintenance easier. It can also adapt incompatible systems to new environments. It involves creating a facade class that offers a single method to the client, taking care of all interactions with underlying components. In test automation, patterns are used to simplify complex workflows and provide user-friendly interfaces. This makes software libraries more accessible and understandable and reduces dependencies. It can also improve poorly written APIs. The following figure shows an example of the Facade pattern:

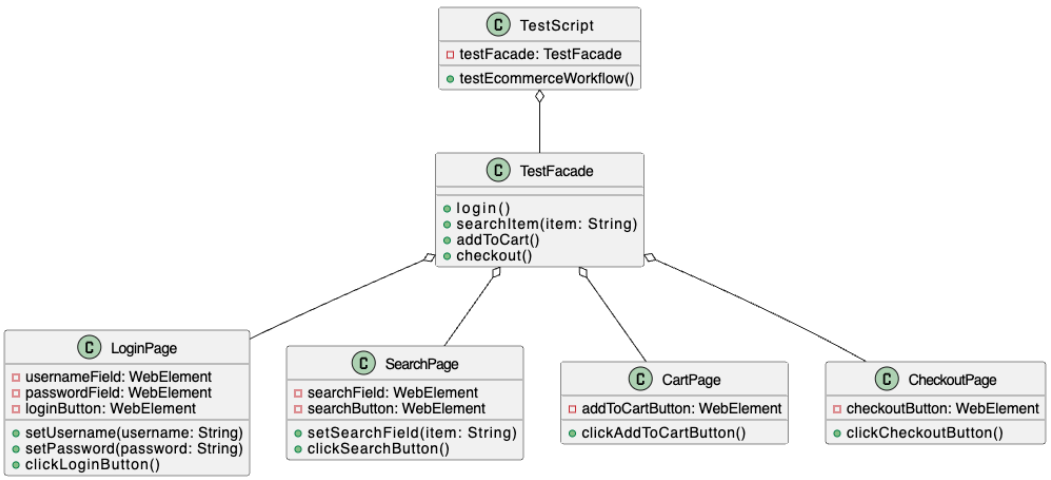

Figure 3.3 – An example of the Facade pattern

This figure shows `TestFacade`, which is the facade that simplifies the interface for an e-commerce workflow. It contains methods for logging in, searching for an item, adding the item to the cart, and checking out. Note that `TestFacade` is a class for `TestScript`:

- **Singleton pattern testing**: To test whether the Singleton pattern is working correctly, you need to create a test class and follow these steps:

  I. Use a reflection API to access the private constructor of the class and try to create another instance.

  II. If the constructor throws an exception or returns the same instance, then you can confirm that the Singleton pattern is working as expected.

  III. Serialize and deserialize the instances and check whether they're equal using an assertion method.

  IV. Use a mocking framework (such as Mockito) to verify the behavior of the class and its methods.

> **Tip**
> You can use testing frameworks such as JUnit or TestNG to run the test class and generate a report of the results.

- **API testing pattern**: This particular pattern delves into examining **application programming interfaces** (**APIs**) in the context of both direct and integrated testing. The primary objective of such testing is to ascertain the extent to which the APIs meet the expected standards for functionality, performance, reliability, and security. API testing can be performed at all levels of the test pyramid or test diamond. The following are some possible test strategies:

  - **Unit tests** (**test pyramid base**): You would start by writing unit tests for individual functions in your code. These tests ensure that each function behaves as expected when given a certain input. For example, you might write unit tests for functions that handle user authentication in your API.

  - **Service tests** (**test diamond base**): Given the rise of microservices, you would write a large number of integration tests to ensure that your services interact correctly. For example, you might write tests to check that your API communicates with your database and other services correctly.

- **The Fluent Page Object pattern**: Fluent interface is an object-oriented API that aims to provide readable code in software engineering. It's usually implemented through method cascading, which is a process of method chaining that relays the instruction context of a subsequent call. The Fluent Page Object pattern applies the concept of a Fluent interface to Page Objects in automated testing. Page Objects are a popular test automation pattern that abstracts the details of the UI structure and behavior into separate classes or objects. The Fluent Page Object pattern

takes this a step further by allowing these Page Objects to have methods that return the Page Object itself. This enables method calls to be chained together, making the test scripts more readable and maintainable.

As we wrap up this section, we've seen how test automation plays a pivotal role in ensuring the reliability and efficiency of software development. Now, let's transition to another integral component of design patterns: **artifacts**. These are the tangible byproducts that are produced during the development process.

Design pattern components – artifacts

Artifacts are important components of design patterns as they represent the tangible products of the design process. Artifacts can be documents, models, diagrams, code, tests, or any other output that helps to communicate, document, or implement the design. Artifacts can also be classified according to the design pattern components they belong to – **participants**, **collaborations**, **consequences**, and **implementations**. To give you an idea of how this classification works, let's take a closer look at these components:

- **Participants** are the elements that make up a design pattern, such as classes, objects, interfaces, or roles. Participants define the structure and behavior of the pattern. Artifacts that belong to this component are usually diagrams or code that show the relationships and interactions between the participants. In this case, as shown in the following figure, a class diagram can show the inheritance and association relationships between the classes that participate in a pattern:

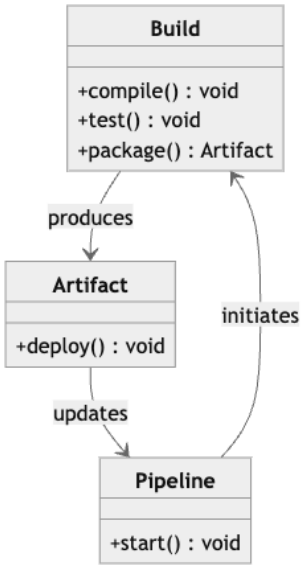

Figure 3.4 – A simplified example and a real-world CI/CD pipeline with various participants, including the components we're discussing

The preceding figure shows a simplified CI/CD diagram. All the classes – that is, `Build`, `Artifact`, and `Pipeline` – are participants. The arrows in the diagram represent the relationships between these participants. For example, the pipeline initiates the build process, the build process produces an artifact, and the artifact updates the pipeline when it's deployed.

Collaborations are the ways that participants cooperate to achieve the desired functionality of the pattern. Collaborations describe the protocols, contracts, and responsibilities of the participants. Artifacts that belong to this component are usually documents or models that specify the rules and scenarios of the collaborations. The following figure shows a sequence diagram that shows the messages and events that occur between the objects that collaborate in a pattern:

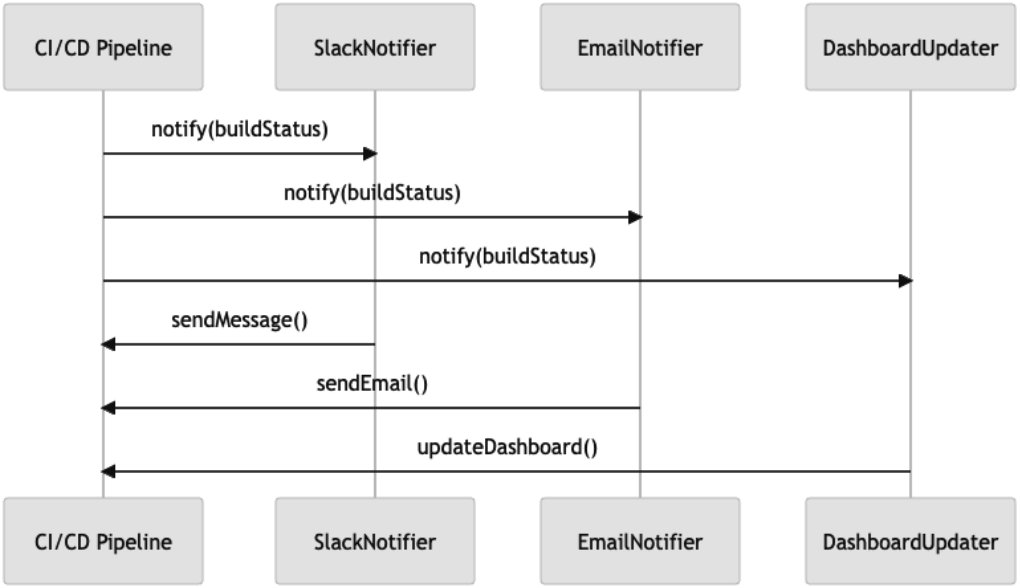

Figure 3.5 – Collaboration in a CI/CD pipeline

This real-world scenario shows how the Observer design pattern can be used to implement a notification system in a CI/CD pipeline where changes in the pipeline's state are propagated efficiently to all relevant services. This ensures that all stakeholders are promptly informed about the status of the builds. The sequence diagram is an artifact that belongs to the collaborations component of the Observer design pattern. It specifies the sequence of interactions between the CI/CD pipeline (subject) and notification services (observers).

- **Consequences** are the trade-offs and implications of using a design pattern in a particular context. Consequences include the benefits, drawbacks, and risks of applying the pattern. Artifacts that belong to this component are usually documents or models that analyze the

impact and suitability of the pattern for a given problem. For example, a decision table can show the pros and cons of choosing one pattern over another:

CI/CD Pipeline State	SlackNotifier	EmailNotifier	DashboardUpdater
Build Success	Send Message	Send Email	Update Dashboard
Build Failure	Send Message	Send Email	Update Dashboard
New Commit	Send Message	Send Email	Update Dashboard

Table 3.1 – Decision table showing the pros and cons of choosing a pattern

This decision table is an artifact that belongs to the consequences component of the Observer design pattern. It illustrates the consequences of the interactions between the CI/CD pipeline (subject) and notification services (observers). It shows that regardless of the state of the CI/CD pipeline, all notification services are notified and perform their specific actions. This ensures that all stakeholders are promptly informed about the status of the builds.

- **Implementations** are the concrete realizations of a design pattern in a specific programming language or environment. Implementations provide guidance and examples of how to code and use the pattern. Artifacts that belong to this component are usually code snippets or tests that demonstrate the syntax and semantics of the pattern. For example, a source code file can show you how to implement a pattern in a `Jenkinsfile` file with Groovy.

  An example of this is shown in the following code:

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo 'Building...'
                // Here you can add the commands to build your
project, e.g., 'mvn package'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing...'
                // Here you can add the commands to test your
project, e.g., 'mvn test'
            }
        }
        stage('Deploy') {
```

```
            steps {
                echo 'Deploying...'
                // Here you can add the commands to deploy your
project
            }
        }
    }
}
```

This `jenkinsfile` file is a part of the implementations component. In the context of software development and design patterns, *implementations* refer to the concrete realizations of a design pattern in a specific programming language or environment.

The Jenkins CI/CD pipeline example is an implementation of the Singleton CI/CD concept, respectively. It demonstrates how these abstract concepts can be translated into working code in a specific programming language (a Groovy for Jenkins pipeline).

Next, we'll look at how artifacts can be used.

Using artifacts

Software artifacts are pivotal in shaping the modern technological landscape. Let's learn how to handle them:

- Identify the problem or goal that you want to solve or achieve with your software.
- Choose a design pattern that suits your problem or goal. A design pattern is a general and reusable solution to a common design problem.
- Create artifacts that belong to the `participants` component of the pattern. These artifacts will define the structure and behavior of the elements that make up the pattern, such as classes, objects, interfaces, or roles.
- Create artifacts that belong to the `collaborations` component of the pattern. These artifacts will describe the protocols, contracts, and responsibilities of the elements that cooperate to achieve the functionality of the pattern.
- Create artifacts that belong to the `consequences` component of the pattern. These artifacts will analyze the trade-offs and implications of using the pattern in your context. They will help you evaluate the benefits, drawbacks, and risks of applying the pattern.
- Create artifacts that belong to the `implementations` component of the pattern. These artifacts will provide guidance and examples on how to code and use the pattern in your programming language or environment.
- Review and refine your artifacts as you develop your software. Make sure they are consistent, accurate, and complete.

Artifacts are essential for understanding, communicating, and applying design patterns in software development. By categorizing artifacts according to the design pattern components they relate to, we can better organize and manage them throughout the design process. Knowing this, next, we'll explore their relationship with CI /CD.

Artifact components related to CI/CD

Artifacts play a vital role in maintaining the quality, reliability, and consistency of software delivery pipelines in the CI/CD pattern. As you may recall, CI/CD is a pattern that automates the integration, testing, and deployment of software changes using tools and practices that enable rapid feedback and frequent releases.

Artifacts can be used in various stages of the CI/CD pipeline to achieve different objectives. For instance, artifacts can be employed to store the build output, deployable packages, and other versions of the application. They can also be used to maintain the configuration and environment settings of the application, as well as to provide visibility into the deployment process, including identifying any issues that may arise. Overall, artifacts serve as a crucial foundation of the CI/CD pipeline and facilitate the smooth delivery of software changes.

The following stages help in this smooth delivery:

1. In the integration stage, artifacts such as source code and configuration files are merged and compiled into a build artifact that can be tested and verified.
2. In the testing stage, artifacts such as test scripts, test data, and test results are used to validate the functionality, performance, and security of the build artifact.
3. In the delivery stage, artifacts such as deployment scripts, deployment packages, and deployment logs are used to deploy the build artifact to a target environment and monitor its status and performance.
4. In the feedback stage, artifacts such as metrics, logs, and reports are used to collect and analyze data about the quality and performance of the deployed artifact and provide insights for improvement.

You can use artifacts in various ways to improve your software development and delivery process. Here are some examples:

- Use source code artifacts to collaborate with other developers, review code changes, and track the history and version of your software.
- Use build artifacts to automate the compilation and packaging of your software and ensure that it's consistent and reproducible across different environments.
- Use test artifacts to automate the testing and verification of your software and ensure that it meets quality and security standards.

- Use deployment artifacts to automate the deployment and configuration of your software and ensure that it's reliable and scalable across different platforms.
- Use feedback artifacts to collect and analyze data about the performance and usage of your software and provide insights for improvement and optimization.

To use artifacts effectively, you need to store, manage, and share them using artifact repositories that provide features such as versioning, security, traceability, and collaboration. Artifact repositories can help you to ensure that the artifacts are consistent, accessible, and reusable across different teams and environments. The following are some examples of artifact repositories:

- **Nexus Repository Manager**: A universal repository manager that supports various formats, such as Maven, Docker, npm, RubyGems, and more
- **Artifactory**: A binary repository manager that supports various formats, such as Maven, Docker, npm, Helm, and more
- **GitHub Packages**: A package hosting service that integrates with GitHub workflows and supports various formats, such as Maven, Docker, npm, and more

So, how does an artifact repository fit into a CI/CD pipeline? The best way to explain this is to show you:

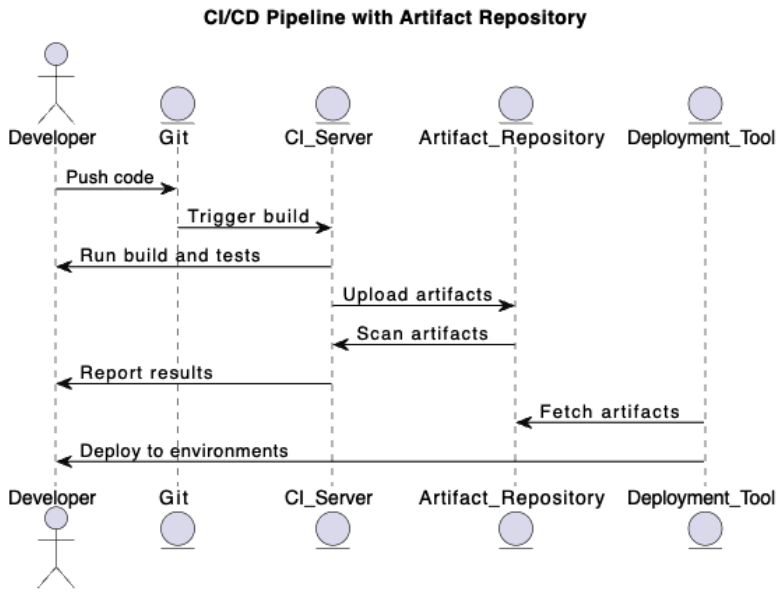

Figure 3.6 – An example of a CI/CD pipeline with an artifact repository

The preceding diagram shows the interactions between the developer, Git (VCS), CI server (such as Jenkins, Travis CI, or CircleCI), artifact repository (such as Artifactory, Nexus, or GitHub Packages), and deployment tool (such as Ansible, Chef, or Puppet) in a CI/CD pipeline.

Of course, this is a simplified representation and actual implementations may vary based on specific tools and configurations used, but it gives you a basic idea of the concepts.

With a solid understanding of the artifacts involved in design patterns, we're now equipped to delve into the various deployment types. The next section will explore the different ways these patterns can be implemented and deployed.

Design pattern components – deployment types

Deployment types are strategies and practices that are used for releasing and distributing software. There are various types of deployment types, each suitable for different scenarios. Here are some common deployment types and their key components:

- **Monolithic deployment**: Here, the entire application is deployed as a single unit. It has the following components:
  - Single code base
  - Single database
  - Scalability challenges

  The use case of monolithic deployment is small to medium-sized applications with simple architectures.
- **Microservices deployment**: Here, the application is divided into small, independent services that communicate through APIs. It has the following components:
  - Independent services
  - Distributed databases
  - Communication through APIs

  The use case of microservices deployment is large and complex applications requiring scalability and flexibility.
- **Containerized Deployment**: Here, applications and their dependencies are packaged into containers for consistent deployment across environments. It has the following components:
  - Containers (Docker, Podman)
  - Orchestration tools (Kubernetes, Docker Compose)
  - Container registry

  The use case of containerized deployment is consistent deployment across development, testing, and production environments.

- **Serverless deployment (Function-as-a-Service – FaaS))**: Here, code is executed in response to events without the need to manage server infrastructure. It has the following components:
  - Stateless functions
  - Event triggers
  - A serverless platform (AWS Lambda, Azure Functions)

  The use case of serverless deployment is event-driven and stateless workloads with variable demand.
- **Blue-green deployment** (see the *The deployment of CI/CD* section in *Chapter 1*): Here, two environments (blue and green) are maintained, allowing seamless transitions between versions. It has the following components:
  - Blue environment (current version)
  - Green environment (new version)
  - Load balancer

  The use case of blue-green deployment is minimizing downtime during updates and testing new releases.
- **Canary deployment** (see the *The deployment of CI/CD* section in *Chapter 1*): This involves gradually releasing a new version to a subset of users before deploying to the entire user base. It has the following components:
  - Control group (existing version):
  - Experimental group (new version)
  - Monitoring and rollback mechanisms

  The use case of a canary deployment is testing new features with a smaller audience to detect issues early.
- **Rolling deployment** (see the *The deployment of CI/CD* section in *Chapter 1*): This involves incrementally deploying new versions across the infrastructure, often node by node. It has the following components:
  - Load balancer
  - Incremental updates
  - Monitoring and rollback mechanisms

  The use case of rolling deployment is continuous delivery with minimal disruption.

- **Feature toggles** (**feature flags**): Features are toggled on or off at runtime, allowing controlled release of new functionality. It has the following components:
  - Feature toggle configuration
  - Conditional code execution
  - Monitoring and control mechanisms

  The use case of feature toggles is gradual feature rollout and A/B testing.

These deployment patterns provide flexibility and efficiency in handling the release and distribution of software applications, where teams can choose the most suitable approach based on their specific requirements and constraints.

Next, we'll look at test automation patterns.

Test automation patterns

Test automation patterns are best practices that help programmers design and maintain test automation **testware** efficiently. Testware is the set of software and data that are used to test a software application or system. Test automation patterns can be classified into four categories: *process*, *design*, *execution*, and *management*. Each category contains several patterns that address common issues or challenges in test automation.

One of the most popular design patterns, the **Page Object pattern**, is a way of modeling the behavior and structure of a web page or a part of a web page using an object-oriented class. The Page Object class encapsulates the elements and actions specific to that page, such as locators, methods, and properties. The Page Object class provides a clear and concise interface for the test scripts to interact with the page without exposing the implementation details of the page. The Page Object pattern has several benefits:

- It makes the test scripts more readable and maintainable by hiding the complexity of the web page behind simple methods and properties.
- It reduces code duplication and improves reusability by centralizing the logic for each page in one place.
- It enhances test reliability by abstracting away the synchronization and error-handling logic from the test scripts.
- It facilitates code changes by isolating the impact of web page modifications to only the Page Object class.

To implement the Page Object pattern, follow these steps:

1. Identify the web pages or parts you want to model using Page Objects. For example, you can create a Page Object for a login page, a home page, a product page, and so on.

2. Create a Page Object class for each web page or part of the web page that you identified. The class name should reflect the purpose or functionality of the web page – for example, `LoginPage`, `HomePage`, `ProductPage`, and so on.
3. Define the elements present on the web page or part of a web page using locators. Locators are expressions that identify an element on a web page using attributes such as ID, name, class, XPath, CSS selector, and so on. You can use tools such as Firebug or Chrome DevTools to inspect the elements and generate locators. It would help if you stored the locators as constants or variables in the Page Object class.
4. Define the methods that perform actions on the web page or part of a web page using Selenium WebDriver commands. This tool automates web browser interactions using various programming languages, such as Java, Python, Ruby, and so on. It would help to use descriptive names for the methods that reflect what they do – for example, `enterUsername()`, `clickLoginButton()`, and `verifyWelcomeMessage()`.
5. Define the properties that return information about the web page or part of a web page using getters and setters. Getters and setters are methods that access or modify the values of attributes or fields in an object. It would help to use meaningful names for the properties that reflect what they represent – for example, `getUsername()`, `setUsername()`, `getErrorMessage()`, and `setErrorMessage()`.

Using this Page Object class, you can write test scripts that interact with the login page simply and intuitively. The following figure shows how:

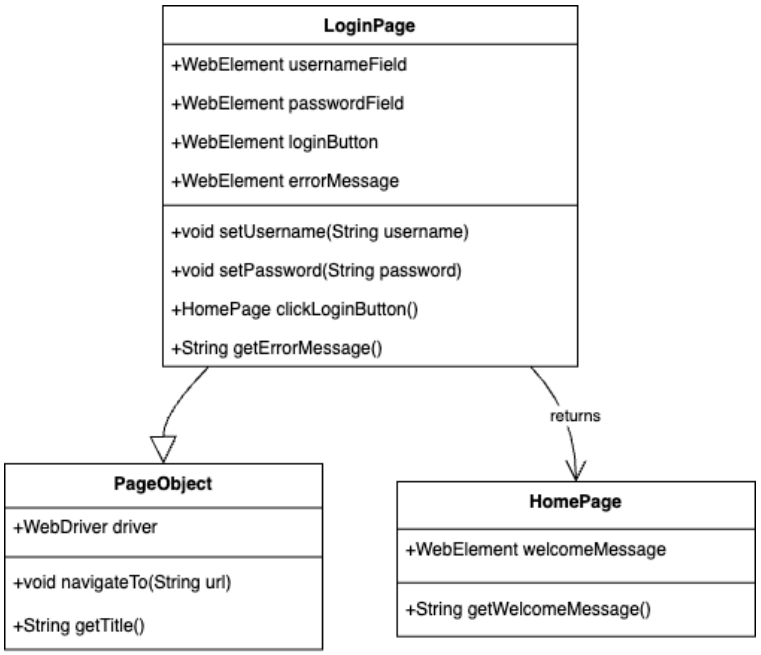

Figure 3.7 – Example of the Page Object pattern

In the preceding figure, `LoginPage` is a subclass of `PageObject`. The `LoginPage` class has fields for the username, password login button, and an error message, which would typically be the locators for these elements on the web page. It also has methods to interact with these elements: `setUsername()`, `setPassword()`, `clickLoginButton()`, and `getErrorMessage()`.

The `PageObject` class has a `WebDriver` field, which represents the browser window or tab that the page is loaded in. It also has methods to navigate to a specific URL and get the title of the current page.

The `HomePage` class represents the page that the user is redirected to after a successful login. It has a field for the welcome message and a method to get this message.

The `clickLoginButton()` method in the `LoginPage` class now returns an instance of `HomePage`, which represents the page transition that occurs after a successful login.

Next, we'll look at **branching strategies**.

Branching strategies

A branching strategy is a set of guidelines for creating and managing branches in a **version control system** (**VCS**). It defines how and when to create branches, what kind of branches to use, how to name them, and how to merge them back to the main branch. A branching strategy can help developers collaborate more effectively, isolate changes, and maintain code quality. Let's have a look at the various branch components:

- **Main branch**: The main branch is the source of truth for the current state of the software. It should always be stable and deployable. All other branches should be derived from and merged back into the main branch.
- **Feature branch**: A feature branch is a short-lived branch that's created to implement a specific feature or functionality. It should be isolated from other feature branches and only merged into the main branch when the feature is complete and tested.
- **Release branch**: A release branch is a branch that's created to prepare a new version of the software for release. It should only contain bug fixes and minor changes, and no new features. It should be merged into the main branch and tagged with a version number when the release is ready.
- **Hotfix branch**: A hotfix branch is a branch that's created to fix a critical bug or issue in the production environment. It should be branched from the main branch or the latest release branch and merged back into both once the fix has been verified.
- **Development branch**: A development branch is an optional branch that's used to integrate multiple feature branches before they're merged into the main branch. It can help reduce merge conflicts and ensure compatibility between features.

Now, let's look at a few other CI/CD patterns.

Other CI/CD patterns

Here are some other CI/CD patterns related to the test automation pattern:

- **Pipeline orchestration**: This involves managing and coordinating the execution of various stages in a CI/CD pipeline.
- **Artifact promotion**: This refers to the practice of promoting an artifact (such as a software build) through various stages of testing and validation until it's ready for production.
- **Immutable infrastructure**: This is a paradigm where servers are never modified after they're deployed. If updates are needed, new servers are provisioned from a common image with the appropriate changes.
- **Parallel workflows**: These are workflows that are executed simultaneously to increase the speed and efficiency of the CI/CD process.
- **Run builds periodically**: This involves scheduling builds to run at specific intervals, which can help with catching issues early.
- **Automate and test the build**: This is the practice of automating the build process and performing tests to ensure that the build meets functional requirements.
- **Continuous integration version control patterns**: These are best practices for managing and integrating changes from different developers in a VCS during the CI process.
- **Deployment strategies**: Various methods or patterns are used to deploy software to production in the CD process. This was discussed in *Chapter 1*, in the *The deployment of CI/CD* section.

With that, we've explored various design patterns that are common in CI/CD. Now, it's time to discover some industry best practices!

Industry best practices and challenges

In this chapter, you learned about the components that belong to CI/CD design patterns as were provided with some clear examples.

But how do you implement CI/CD effectively? What are the industry best practices that you should follow? In this section, we'll answer these questions by exploring some of the most popular and useful CI/CD design patterns that were described earlier in this chapter. We'll also explain how they can help you improve your software delivery performance and quality. Finally, we'll discuss the main components of a CI/CD pipeline, and how they interact with each other.

There are many ways to classify CI/CD design patterns, but one of the most common and useful distinctions is based on the level of automation and feedback they provide. Based on this criterion, we can identify three main types of CI/CD design patterns:

- **Basic CI/CD**: This involves automating the build and test stages of the software development life cycle and providing fast feedback to developers on the quality and status of their code

changes. Basic CI/CD helps developers detect and fix errors early before they become more costly and difficult to resolve.

- **Continuous delivery**: This enables developers to deliver software to a staging or production environment with minimal manual intervention when the code passes all the automated tests and checks. Continuous delivery helps developers reduce the time-to-market and increase the frequency of software releases.
- **Continuous deployment**: This involves automating not only the deployment stage but also the release stage of the software development life cycle. It enables developers to deliver software to end users without any human approval or intervention, so long as the code meets all the quality and performance criteria. Continuous deployment helps developers achieve maximum agility and responsiveness to customer feedback and market changes.

Now, let's have a look at the different components of a CI/CD pipeline.

What are the components of a CI/CD pipeline?

A CI/CD pipeline is a sequence of steps or stages that a code change goes through from development to delivery. Each stage performs a specific task or function that contributes to the overall quality and reliability of the software product. The components of a CI/CD pipeline may vary depending on the type of CI/CD design pattern. We touched on all of them in this chapter, but let's summarize them in a compact overview:

- **Source control**: This is where developers store and manage their code changes using a VCS such as Git or SVN. Source control enables developers to collaborate effectively, track changes, and revert to previous versions if needed.
- **Build**: This is where developers compile, package, or transform their code into an executable or deployable format using a build tool such as Maven or Gradle. The build stage ensures that the code is syntactically correct and consistent across different environments.
- **Test**: This is where developers verify the functionality, quality, and performance of their code using various types of automated tests, such as unit tests, integration tests, or end-to-end tests. The test stage ensures that the code meets the functional and non-functional requirements and expectations.
- **Deploy**: This is where developers/admins/operators deploy their code to a target environment such as staging or production using a deployment tool such as Ansible or Kubernetes. The deploy stage ensures that the code is properly configured and installed on the target environment.
- **Release**: This is where developers make their code available to end users or customers using a release tool such as Jenkins or Spinnaker. The release stage ensures that the code is delivered in a controlled and safe manner, with features that can be enabled or disabled as needed.

Now that we know about the components, we should have a look at the implementation steps.

What steps should be taken to implement CI/CD?

The first step to implementing CI/CD is to use a VCS, such as Git, to manage your code. Version control helps you track changes, collaborate with other developers, and revert to previous versions if needed.

The second step is to set up a CI server, such as Jenkins, Travis CI, or GitHub Actions, which can automatically build, test, and verify your code whenever you push a new commit. This way, you can catch bugs early, ensure code quality, and avoid integration issues.

The third step is to use a CD tool, such as Ansible, Chef, or AWS CodeDeploy, which can automatically deploy your code to different environments, such as staging, testing, or production. This way, you can deliver your software faster, reduce manual errors, and ensure consistency across environments.

Some of the industry best practices that you should follow when implementing CI/CD are as follows:

- Use a branching strategy that suits your project and team. For example, you can use a feature branch workflow, where each developer works on a separate branch for each feature and merges it to the main branch after testing and review.
- Write unit tests and integration tests for your code and run them on every commit. This will help you ensure that your code meets the requirements and works as expected.
- Use code quality tools, such as SonarQube, CodeClimate, or Codacy, which can analyze your code for issues such as complexity, duplication, security vulnerabilities, or coding standards. This will help you improve your code quality and maintainability.
- Use code review tools, such as GitHub Pull Requests, Code Review, or Gerrit, which can facilitate peer review of your code and provide feedback and suggestions. This will help you improve your coding skills and learn from others.
- Use configuration management tools, such as Puppet, SaltStack, or Terraform, which can automate the provisioning and configuration of your infrastructure. This will help you ensure that your infrastructure is consistent and scalable. IaC is an important part of CI/CD nowadays.
- Use monitoring tools, such as Prometheus, Grafana, or Datadog, which can collect and visualize metrics and logs from your application and infrastructure. This will help you track the performance and availability of your system and troubleshoot issues.
- Use feedback tools, such as Jira, Trello, or Slack, which can enable communication and collaboration between developers and stakeholders. This will help you gather feedback from users and clients and incorporate it into your development process.

So, why are these industry best practices? We'll explain this in the next section.

Why are these industry best practices?

The CI/CD design patterns and components we covered in this chapter have been established as industry best practices due to their ability to deliver high-quality software in a fast and reliable manner. These practices have been proven effective and efficient over time. By adhering to them, software development teams can ensure seamless integration, testing, and deployment of their code changes, resulting in faster release cycles. With them, you can do the following:

- Improve the quality of your software by detecting and fixing errors early in the development process. This will prevent bugs from reaching production and ensure that your code meets all the standards and expectations.
- Increase the speed of your software delivery by reducing manual tasks, eliminating bottlenecks, and accelerating feedback loops.
- Enhance the reliability of your software delivery by minimizing risks, avoiding failures, and ensuring consistency across different environments.
- Optimize the performance of your software delivery by maximizing resource utilization, reducing costs, and increasing customer satisfaction.

In conclusion, these CI/CD design patterns and components are considered best industry practices due to their proven effectiveness in delivering high-quality software swiftly and reliably.

Challenges around implementing CI/CD

Implementing CI/CD design patterns comes with a set of challenges that organizations must navigate to ensure successful deployment. One of the primary hurdles is integrating these practices into existing projects, particularly those that are large-scale or legacy, which can be complex and require significant expertise. Security and compliance are also major concerns as CI/CD pipelines can introduce vulnerabilities that need to be addressed through automated security testing at every stage. Additionally, managing version control, scaling the pipeline, operational inefficiencies, and a lack of expertise can impede the smooth adoption of CI/CD methodologies. To overcome these challenges, it's crucial to conduct thorough feasibility studies, start with smaller projects to test approaches, and ensure continuous learning and adaptation of the team to new tools and processes.

Summary

In this chapter, we continued to explore CI/CD design patterns, focusing on the transition from testing to deployment. It's a journey that uncovers the versatility of CI/CD workflows, highlighting the adaptability of branching strategies, the practicality of feature toggles, and the efficiency of trunk-based development. These methodologies aren't standalone; they're supported by a backbone of key CI/CD components such as robust testing frameworks, which ensure code integrity, and deployment tools, which streamline the delivery process.

This chapter served as a guide for developers to navigate various CI/CD design patterns, empowering them to select the most fitting options tailored to their project needs. This selection is important as it directly influences the quality of software and the velocity of its delivery. By understanding the nuances of each design pattern and component, developers can craft a CI/CD pipeline that's not only effective but also resilient to the ever-evolving demands of software development.

Our journey through CI/CD design patterns began with the cornerstone of any reliable pipeline-test automation. This crucial element ensures that every iteration of the code is rigorously tested for functionality, performance, and security. Then, we looked at artifacts, the byproducts of continuous integration processes that need to be managed and stored efficiently. Lastly, we addressed the various deployment types, each with its own set of practices and challenges, providing a comprehensive view of the deployment landscape within CI/CD paradigms.

This chapter provided a conclusive synthesis of the critical elements that constitute CI/CD design patterns and components. It elaborated on the importance of a well-orchestrated CI/CD strategy, which is evident in achieving a seamless flow from code conception to code deployment, ultimately leading to accelerated delivery of high-quality software. The insights provided here are invaluable for developers looking to enhance their CI/CD practices.

In the next chapter, we'll look at implementing CI/CD patterns.

4

Business Outcome Alignment with CI/CD Design Patterns

In this chapter, we will connect the business outcome to CI/CD design patterns. Business outcomes can be considered as top-level goals. Organizations steer to launch new product features and services to enhance profitability, brand recognition, and customer satisfaction.

It is not always simple to clearly articulate and prioritize business goals without a systematic plan, and then it becomes even more complex to connect initiatives such as CI/CD design pattern implementation to **objectives and key results** (**OKRs**). OKRs are a goal-driven framework that provides an effective way of defining business objectives and managing them through goals. The OKR-based way of driving business outcomes with goals mapped to different functional areas helps siloed teams come together for a purpose. Moreover, OKR-based goal setting can help connect technical teams and initiatives (such as CI/CD design pattern implementation) to a particular business outcome.

Often, business outcomes and connected goals depend not only on the end state of CI/CD implementation but also on the current state of maturity of how CI/CD practices are implemented. In this chapter, we will start to describe the state of CI/CD design patterns and a systematic plan to connect CI/CD design pattern implementation to actionable goals and objectives at an organizational level.

By the end of this chapter, you will have a better understanding of how to connect CI/CD design patterns to business objectives. We will also discuss how to overcome some of the common barriers to the adoption of CI/CD design patterns and connected challenges. Moreover, you will gain insights into some of the best practices and tips for aligning the implementation and adoption toward a bigger business outcome.

This chapter has the following main headings:

- CI/CD design patterns – strategic intent
- State of play – CI/CD design patterns
- A decade of action using CI/CD design patterns

- North Star for CI/CD design patterns
- Scaling adoption of CI/CD design patterns
- Resistance to change – cultural aspects

Let's get started!

CI/CD design patterns – strategic intent

It is undebatable that software plays a vital role in every business today. Many of the core capabilities of a business are likely software-dependent. Making the most of investment in software capabilities is not a choice anymore, it is a must. Software initiatives must align with the overarching business goals, help identify and close the strategic gaps in the overall ecosystem, and improve the development and deployment of software capabilities with ease and speed. Moreover, software applications are becoming inherently more complex and data-intensive. They should be more readable, modular, evolvable, and efficient to serve the business purpose.

CI/CD design patterns can provide you with a systematic, well-designed approach to improving productivity and profitability and removing short- and long-term inefficiencies while developing, deploying, and maintaining software applications. The use of predefined templates, infrastructure, and pipelines (which form the components of a catalog of ready-to-use CI/CD design patterns to apply to software applications) can be very handy. These patterns can help speed up the must-have software features, simplifying the roll-out of the features and facilitating feedback loops, reliability, and maintenance.

As we all agree, the main reason why an organization exists is to launch profitable products and services that can be sustained and scaled. That is an obvious statement. However, we often observe that the technical part of an organization is separated from the business side. In this case, how will the business side know that IT is doing a good job? And from the opposite perspective, how will IT know that a business might struggle because of bad strategic decisions made by the IT side?

Let's start with a very simple approach to connect features of a design pattern to business outcomes. Strategic intent, in a nutshell, refers to applying CI/CD design patterns to improve developer productivity and product profitability by removing inefficiencies in software life cycle management.

The following table depicts how to connect the strategic intent of a design pattern with business outcome simply:

Features of design pattern	**Strategic intent**	**Business outcome**
Reusability	A solution to common problems	Develop, design, and deliver cost-efficient software solutions to enhance profitability
Abstraction	Provide a channel to extend and add more features	Competitive advantage to enhance the value of the product with limited investment

Features of design pattern	Strategic intent	Business outcome
Maintainability	Easy to modify and support	Reliability improving customer satisfaction
Flexible and adaptable	Fast to evolve without substantial effort	Fast innovation enabling brand recognition

Table 4.1 – Connecting the strategic intent of design patterns with business outcomes in a simple way

As a next step, we will attempt to detail the implementation of the business outcomes described in *Table 4.1* and align them to technical aspects of CI/CD design patterns. Furthermore, we have ways to connect the technical aspect with the business purpose and the actionable objectives of CI/CD. Taking reference from *Strategizing Continuous Delivery in the Cloud*, (Garima Bajpai and Thomas Schuetz, 2023) where the authors have described very well a systematic way of connecting the business purpose with the technical side of CI/CD.

In the following table, we will look at different aspects of creating a systematic plan for connecting the technical side and aligning it with purpose, long-term goals, actionable objectives, and measurements (*Strategizing Continuous Delivery in the Cloud*, 2023, by Garima Bajpai and Thomas Schuetz).

Characteristics	Description	Example
Purpose	Articulate customer values aligned with your organizational values	Accelerate the organization's ability to digitally transform
Long-term and forward-leaning goals	Related business outcomes that are aligned with the purpose and provide your organization's strategic advantage	Enhance the performance of software delivery in the organization
Actionable objective	How do business, software engineering, and other related teams orchestrate value? How can you rapidly iterate and evolve to create that incremental value?	Quickly release software features and provide a rapid response to any failures
Measurable	How progress and success will be quantified and measured	Release frequency of new features and bug fixes

Table 4.2 – Simple illustration of a strategic plan (strategic goals and objectives of modernizing CD in the cloud)

In *Table 4.2*, we mentioned some examples of implementing business outcomes through CI/CD, particularly considering features that improve release frequency, enhance troubleshooting, and so on. In the next section, we will try to understand the state of play, map the existing design patterns, and analyze the success of adoption, keeping in view the specific goals affected.

State of play – CI/CD design patterns

In this section, we will investigate the CI/CD design patterns' state of play, which means how CI/CD implementation is done today, highlighting patterns of implementation, areas of improvement, core challenges, and other related adoption barriers. Let's start with an exploration of the potential of the sequential, parallel execution of a software application (see the *Design pattern principles – sequential and parallel execution* section in *Chapter 2*) and applying CI/CD design patterns.

Different generations of software technologists have given many inferences, connection points, and differently interpreted types of software application development and execution. Today's modern software applications can be classified as a hybrid of parallel and sequential computing. In the following table, we provide you with an overview of the state of design methodology, and CI/CD design pattern integration with sequential, parallel, and hybrid execution techniques.

Type of execution	Definition	Description
Sequential	Sequential execution is a technique that follows a linear and ordered logic of execution, where a software program executes one part of its code after another on a single processor or core.	Outlining discrete instructions and finding the right order to execute them with no overlap. Inherently, CI/CD is a sequential set of automated steps designed to facilitate continuous integration and delivery. It supports sequential execution efficiently. Specific design patterns can be implemented and can evolve CI/CD into more event-driven models such as behavioral design patterns for CI/CD. Often, sequential execution looks at the application from the top down, takes the whole software system as one entity, and then decomposes it to achieve more than one sub-software system.
Parallel	Decompose the software application into independent tasks, design and assign pieces of code to process, orchestrate the processes, and map the output.	Outlining independent tasks for execution and orchestrating them for execution; task execution can overlap. All major CI/CD systems support some form of parallelization. Some more work needs to be done for parallel execution – for example, features such as the adoption of auto-splitting jobs and running parallel jobs on different infrastructure environments, including cloud providers. Often, we experience that a bottom-up design approach is used to redesign sequential applications to parallel, starting from basic/specific tasks and moving up.

Type of execution	Definition	Description
Hybrid	A mix of sequential and parallel execution.	A mix of top-down and bottom-up approaches working together to develop an application. Choosing the right granularity is important. Optimize the cost and performance by tuning the code and the environment to make use of the best of both worlds. Implementing hybrid execution often means breaking down the application from a top-down and bottom-up perspective. An example is the implementation of progressive delivery techniques with continuous deployment and integration into the CI/CD design pattern. Another example is where CI/CD facilitates interoperability of applications by deploying them on more than one environment.

Table 4.3 – Sequential, parallel, and hybrid execution of a software application and CI/CD design patterns

The preceding table depicts how the evolution of designing software applications has taken place from sequential implementation to hybrid execution. With that, CI/CD design patterns are also evolving, and new practices and techniques such as progressive delivery have come into existence. Adding to this, the core components of the design patterns, specifically the pipeline, have also undergone improvements.

Now, we will discuss the core components of the design pattern, specifically the pipeline and infrastructure and its present-day implementation to assess the state, and how they are helping organizations achieve their core objectives. In the next section, we will dig deeper into the evolution of design pattern components.

A decade of action using CI/CD design patterns

Since the rise of DevOps in around 2009, continuous integration, delivery, and deployment have taken center stage. Some of these practices (such as continuous integration) existed before the inception of DevOps; however, with the DevOps movement, these practices and their adoption have accelerated. Some people might think that, if CI/CD were around before being connected to the massive movement of DevOps, that must be enough time to mature and scale the adoption in a standard way.

The following are some sources where experts talk about components of CI/CD patterns; one conclusion can be made – CI/CD is well established on the market:

- `https://martinfowler.com/articles/continuousIntegration.html`
- `https://aws.amazon.com/builders-library/cicd-pipeline/`

- https://bestpractices.cd.foundation/
- https://cd.foundation/state-of-cicd-2024/
- https://techstrong.tv/videos/cd-pipeline

However, it is not entirely true (what we explain in the next few pages of this chapter), especially in terms of approaching scale. We can easily find many articles describing how to scale CI/CD; however, these articles miss important points – standardization and governance. This book aims to fill these gaps.

The part of the process that is well defined is related to the CI part, in both theory and implementation. Let's explore the way CI should be built and the defined, high-level flow for this process.

Figure 4.1 – Default process visualization of CI

This diagram shows the high-level, common structure of the CI pipeline. The stages can differ, depending on the needs. For example, when working with code that doesn't need to be compiled, such as Python, we don't need a compilation phase. Sometimes even the packaging phase is not needed.

Also, a very well-established pattern is related to configuring feedback loops. Although these can differ between organizations, the design is the same.

The following figure shows the possible feedback loops that can be defined in the CI process. We can set up a lot of triggers for feedback and define very fine-grained loops to keep the process operative.

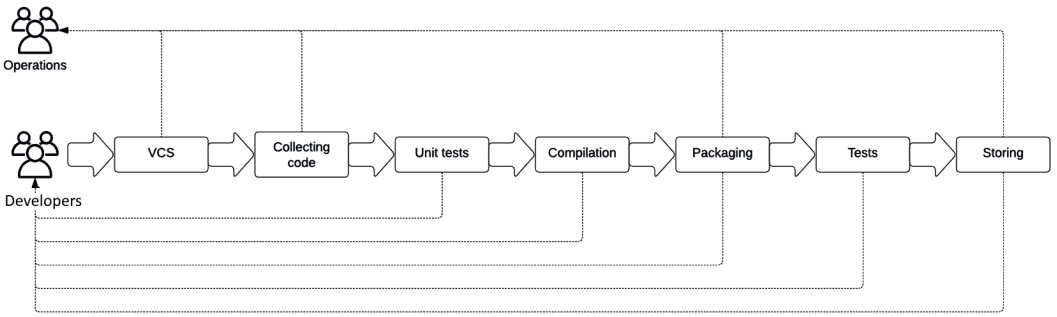

Figure 4.2 – Feedback loops can be defined in many stages of the process to give accurate and quick responses

Some may say that this way, we create a lot of noise in the process. Well, that might be true from the configuration standpoint, but the successful process will not be noisy at all. Please remember that these feedback loops give us an understanding of how well our processes work! CD patterns are also well established; however, in practice, we have learned that there is still a gap in understanding what

continuous delivery and deployment are as well as their differences. In the established approach, which is shown in the following figure, we go through the continuous delivery process.

Figure 4.3 – Continuous delivery visualization

The best practices for high-level design enforce tests after each deployment and before moving to the next environment. By this, the deployment of broken packages is prevented. How will the pipeline behave when a problem occurs? This is also a well-known approach, established with many feedback loops implemented through integrated steps during the development and deployment of the CI/CD pipeline, as shown in the following figure:

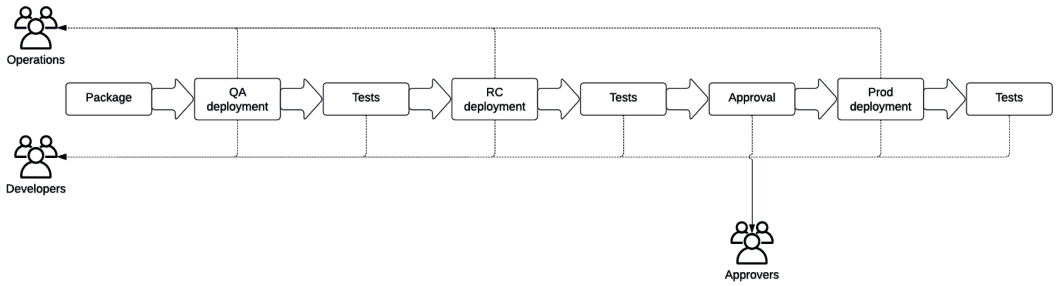

Figure 4.4 – Continuous delivery feedback loop diagram

In the case of continuous delivery, we need to ensure that an *approvers* group is in the loop. Continuous deployment doesn't need this element, as all processes are fully automated, and decision gates are evaluated automatically. Continuous delivery, however, expects that the process will have some manual steps, such as approvals for the continuation of the remaining steps. In this case, we need to define and configure the approvers group, which will receive proper notifications, such as requests for approval, and so on.

Why did we say that CI/CD patterns were not well-established at the beginning of the *A decade of action using CI/CD design patterns* section? It is for the following reasons:

- **Infrastructure delivery**: As an industry, we are still searching for the best patterns to deliver infrastructure. At this point in time, we have learned that generic CI/CD tools might not be enough to properly manage infrastructure delivery and many new platforms related solely to this arise.
- **GitOps**: GitOps is a relatively new approach and changed the model from a push to a pull model (more information about these models can be found in the first chapter of this book). Though it doesn't change the design of the pipelines much, it changes the definition of the triggers. Many teams are still struggling with it.

- **Scale**: Practically all discussions we had with engineers and experts reassured us that scaling is one of the biggest challenges in CI/CD for organizations. There are many reasons for that, but most of them have their roots in bad planning of the CI/CD design or, even worse, no planning at all.
- **Security**: We always put security in the last place, and we do it on purpose. During speeches at events, if the audience remembers only one thing from the talk, it will be security. Too often, we forget about security and we do not treat it as an integral part of CI/CD. Application security – yes. Secure workstations – of course. Yet security for CI/CD was never an important element. Of course, we use security tools in pipelines to check the quality of the code, but how do we secure pipelines themselves?

The aforementioned components must be integrated and still need to be properly established and implemented in the organization's procedures. Currently, we see that organizations have started to focus on these aspects. In this book, we help to establish best practices in these areas and avoid simple mistakes.

Introduction to CI/CD measurements

Let's dig deeper into the strategic alignment of CI/CD design patterns with business outcomes and how to make it measurable. Measurement doesn't mean monitoring. The goal of monitoring is to ensure the control and stable function of the system. Measurements here are about measuring the quality of the process. In our case, this is the delivery process. What to measure? There is no simple answer to this. In fact, decisions should be taken by joint forces of the IT and business units. The measurements must be relevant for business and achievable and deliverable by IT.

The most well-known framework to measure DevOps delivery processes is **DORA**. However, DORA is not a simple framework for measuring some metrics. In fact, it is a project and team called **DevOps Research and Assessment**, born at Google. This group publishes yearly reports called *State of DevOps* reports and these are available (with much more material) at `https://dora.dev`. Every DevOps practitioner should be familiar with DORA.

The DORA model consists of four main metrics, and these metrics are split into two categories – **throughput** and **stability**.

Throughput shows how capable our process is, while stability responds to the question of how stable the processes and systems are, as shown in the following figure.

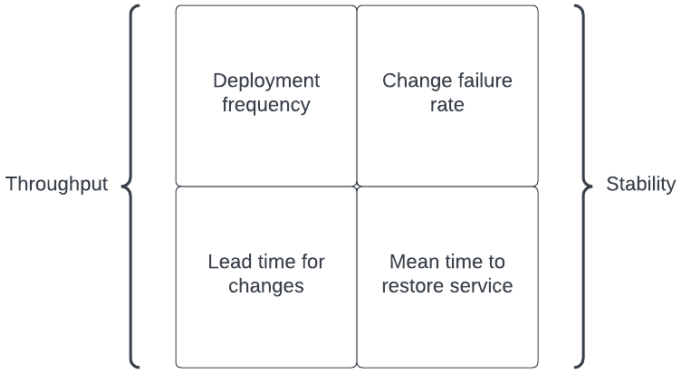

Figure 4.5 – DORA metrics measurement matrix

DORA measurements have wide usage. We use them to measure the organization's level of DevOps adoption and to compare the hundreds of companies on the market. However, it is possible to use DORA measurements to control simple processes. As this book is about CI/CD, let's take a look at CI/CD through the DORA lens.

First, let's depict the throughput category from the DORA matrix presented previously:

- **Deployment frequency** measures the number of deployments in time. More deployments generally mean more effective processes, and smaller changes deployed (which also has an impact on improved stability and security).
- **Lead time for changes** answers another important question – how long does it take to deliver the change from commit to production?

These two metrics are shown on the left side of the preceding figure. We measure the throughput with them. These metrics allow us to understand the current state and trends, how well our process works, how fast teams can deliver changes, and how often they deliver them.

The right side of the figure represents the stability metrics and covers the following:

- **Change failure rate** looks at the ratio between executed pipelines and the number of failures. It shows the stability and efficiency of the processes. A high percentage shows problems that should be attended to. This metric doesn't show where the problem is, it simply shows that the problem exists.
- **Mean time to restore (or recover) service** shows the time needed to restore failed pipelines, in other words, to deliver the change to the target environment after a failed run of the pipeline.

With these metrics, we know how robust our process is and whether there are any signals of incoming issues.

In the early 2020s, DORA introduced a new metric – reliability. In fact, we should use it as a third dimension (throughput, stability, and reliability). This metric is not fully quantifiable, as it relies on many individual factors related to the specific system. The main elements that comprise the reliability dimension are availability, scalability, performance, and so on.

DORA is a key model for measuring the effectiveness of CI/CD in the organization. But that is not all. If we use only the four metrics mentioned previously, the data received can lead to wrong conclusions. Therefore, we need other measurements to understand the proper alignment with the business.

Examples of complementary measurements are as follows:

- **Build success rate**: This is a metric related to building the artifact. When we troubleshoot the pipeline's failed execution, we need to understand the cause. The causes can be grouped into three sets:
  - **CI/CD tool failure**, which means that the tool we use encounters an internal issue. There might be no space on the device, no memory available, execution nodes not available, and so on. These are issues related strictly to the CI/CD tool and its runtime.
  - **Development failure** means that the code cannot be compiled and the reason lies on the development side.
  - **Environment failure** might be related to wrong or missing credentials or misconfigurations of the target environment.

If the executable or package is built successfully, it should direct our attention to non-development issues. This is important, as the goal of CI/CD processes is to deliver the work of developers to production, after all.

Another set of measurements that should be used contains the following:

- **Stage duration** (including build duration, test duration, etc.): As a part of the lead time for changes, this metric plays a crucial role in the effectiveness of the pipeline execution. By understanding each stage, we are able to implement improvements to make the process quicker and more robust.
- **Test coverage**: Although there are discussions about the relevance of test coverage, it might be a very important metric in our process, especially if the failure rate is high.
- **Mean time to detect** (**MTTD**): Similar to **mean time to restore** (**MTTR**) tells us about the effectiveness of recovery after the failure. While MTTR describes the full process, MTTD tells us how long it took to acknowledge the issue and, therefore, whether there is room to improve feedback loops, notifications, and so on.

At this point, we know the best practices and patterns we should use. In *Chapter 4*, we will see why a proper approach to designing a single pipeline is crucial when we start to scale our setup.

In this section, we discussed a few opportunities and challenges in analyzing the current state of the key components for CI/CD to help in creating a ready-to-use catalog of CI/CD design patterns. Let's take a look at the pattern used widely as the **North Star**.

North Star for CI/CD design patterns

Outlining the opportunities of CI is one practice that is well-defined. Another area that has made substantial progress is metrics with DORA.

However, continuous delivery and deployment still need to address some improvements, as discussed in the previous section. We also discussed that the infrastructure layer is still evolving with new projects such as GitOps, OpenTofu, and others. A very important point is practitioners are still working on planning CI/CD design patterns and implementing them in the best possible way with limited standardization.

Also, we tried to align with the tactical outcome by implementing DORA measurements. In this section, we will discuss our North Star for establishing CI/CD design patterns so we achieve our core objective of building profitable software products and services at scale with resiliency.

If we want to establish well-defined design patterns that are ready for deployment, we need to establish guidelines to assess the maturity of the pattern, its use in specific scenarios, and other factors. We can start by outlining some characteristics that will differentiate and assess the patterns as follows:

- **Productivity of CI/CD design patterns**: Assessing the reusability, abstraction, maintainability, flexibility, and adaptability. How well it fulfills the DORA measurements, complemented with other measurements listed in the previous section.
- **Transparency, traceability, and accountability**: A transparent supply chain reduces the administrative costs and effort for maintaining the deployed design patterns.
- **Interoperability**: Navigate CI/CD way into the broader product portfolio with limited to no human intervention.
- **Discovery, including new methods**: Achieving scale by the adoption of new methods, such as injecting security, compliance, AI-enabled tools, and techniques.

In *Chapter 11*, we will discuss the auditing and assessment of design patterns and establish the maturity model for the patterns in detail. For now, let's move on with the adoption of CI/CD design patterns in the next section.

Scaling adoption of CI/CD design patterns

As the quick release of software features and rapid response to any failures remain the key actional objectives of the business community, they can act as our catalyst for increasing the adoption of CI/CD design patterns that are tried and tested to rapidly achieve business objectives. A robust foundation of design patterns will enable purposefully creating, developing, and deploying software at scale. This will also enable the business to steer responsible innovation. To introduce the catalog of design patterns for CI/CD in a simple way and connect it to the core business outcome, we are taking the four specific design patterns as our baseline for our catalog. These four design patterns comprise sub-design patterns and components, which are discussed in detail in *Chapter 2*:

- **Structural CI/CD design pattern**: This is the most fundamental pattern, which uses a well-defined pipeline and infrastructure components that integrate in a well-documented way. It focuses on the productivity of the CI/CD.
- **Creational CI/CD design pattern**: This design pattern primarily refers to the step-by-step implementation of existing capabilities integrated in one way. You can imagine the CI/CD capabilities provided by cloud providers, with full-stack CI/CD deployments as examples. It reduces complexity and increases transparency and accountability by clearly defining the roles and responsibilities for managing and maintaining the components of the CI/CD design patterns, such as **IaC** and pipelines.
- **Behavioral CI/CD design pattern**: This design pattern focuses on interaction and communication within the pipeline in an event-driven way. It supports interoperability with well-defined roles and responsibilities, ensuring a high degree of transparency and traceability.
- **Domain-driven CI/CD design pattern**: This design pattern is tailor-made for specific domains, introducing new ways to inject security and a high degree of compliance that is required for specific domains such as aviation, defense, and so on.

The adoption of these design patterns requires removing the fundamental barriers and roadblocks. Different sets of actors have specific roles to play to enhance the adoption, starting from focusing on standards, infrastructure investment, and R&D initiatives focused on simplifying the adoption of design patterns. These design patterns can be integrated into developer portals as a catalog and integrate the data, API, and key metrics to further evolve the patterns.

Large-scale adoption needs collaboration, coordinated advocacy, and enabling standards and guidelines. It is also important to be data-driven and understand the practitioner's pain, providing commercialization support and technology guidelines. In the next section, we will explore some ways to remove the roadblocks to the adoption of such a catalog of design patterns for CI/CD.

Removing the barrier for CI/CD design pattern adoption

We now can understand some of the fundamental challenges for the large-scale adoption of design patterns for CI/CD. It is essential that we try to also address how we can remove this barrier through a systematic outlook on the evolution of the design patterns aligned to the core business need.

It is evident that to evolve the CI/CD posture and facilitate the onboarding of design patterns, its components, and integration within the ecosystem, we need to investigate the various blockers for which we would require quality data. Not enough is done to survey the state of the nation of CI/CD adoption and exchange of data within.

the practitioner community. There is a limited understanding of the common challenges and limited effort to remove them.

Another important aspect is technology infrastructure; silos are predominant and distributed ownership of applications and infrastructure is posing a challenge to advance the posture.

The adoption and evolution of the patterns alongside the tools would need a significant number of human resources; although it seems that everyone is catching up with the next technological advancement, there is still a substantial lack of expertise. The pool of experts and practitioners is still a small bunch of people. Rapid upskilling and reskilling would be needed to remove the barrier of CI/CD pattern adoption.

Lastly, collaboration to advance the guidelines, reference architecture, and regulations hand in hand with business needs is a must. Without business professionals' involvement and collaboration, there will be a lack of incentives for the evolution of CI/CD design patterns and for building a catalog of patterns for adoption.

In the next section, we will look into the cultural aspects briefly and provide recommendations to overcome them.

Resistance to change – cultural aspects

The CI/CD world is not just about **automation**. In fact, automation is only one part of the bigger picture. Connecting CI/CD to the larger movement of DevOps and the core principles – **CALMS**, which is short for **Culture, Automation, Lean, Measurements, and Sharing** – we also have a cultural aspect. To conclude the chapter, in this section, we will discuss an important aspect that often leads to blocked initiatives – resistance to change.

Resistance to change is commonly observed in organizations. It is an unwillingness to adapt to new ways of doing well-known things. It is vocalized as *We always did things this way and it always worked*. We bet you've all heard it at least once in your career. It is no different when we take the CI/CD approach into consideration.

Quite often, we have observed pipelines designed in the same way as Bash scripts (or any other scripting language). These pipelines were, simply speaking, moved from one place to another, without changing anything except the tool.

The problem with doing this is very similar to doing *lift and shift* from monolithic virtual machines to a microservices world. It works, but it is far from being optimized for the new technology. Although we use scripting in the pipelines we create, these scripts perform specific tasks, such as creating the bundle, executing a set of commands to prepare the environment, and so on. The overarching CI/CD tool provides much more than any other carrier for script execution. Therefore, we have to start thinking differently about using it.

Automation itself is also a subject of resistance. Fully automated pipelines, when built properly, might replace engineers – that is a common fear. For those who are responsible for the processes, or even organizations, the fear is different, especially in well-established, older organizations, or those that are under strict regulations. This fear is about automation itself. Consciously abandoning the possibility of controlling the process manually and leaving everything in the tools' decision is very risky and not reliable in their minds. This means that the process that can be easily automated is interrupted many times by manual checks and controls.

To overcome resistance, the organization must find the true reasons for it. It is our experience that what we see as a reason at first sight is only a projection of a deeper problem.

Summary

In this chapter, we simplified how CI/CD design patterns can impact the overall business outcome and the strategic goals of the organization at the top level. We also addressed how CI/CD design pattern implementation is critical for software-driven organizations. Furthermore, we investigated the progress made in the last decade for CI/CD, outlining the key opportunities and challenges. Lastly, we described how robust CI/CD design pattern implementation can provide a foundation for developing and deploying software at scale. These helped you gain practical insights into overcoming common barriers to CI/CD adoption, which supports scalable, resilient software deployment in alignment with organizational goals.

In the next chapter, we will explore how to implement and scale the Structural design pattern for CI/CD.

Further reading

Here, you can find the sources to expand your knowledge about the specific concepts in this chapter:

- *Design Patterns for Cloud Native Applications: Patterns in Practice Using APIs, Data, Events, and Streams* (May 2021) by Kasun Indrasiri and Sriskandarajah Suhothayan.
- *Strategizing Continuous Delivery in the Cloud* (2023) by Garima Bajpai and Thomas Schuetz.

Part 2: Structural Design Patterns for CI/CD

In this part, you will learn how structural design patterns deal with components and their relations in the CI/CD pipeline and how they get applied in terms of software delivery. You will also dive deeper into different deployment strategies and their advantages and disadvantages while deploying software.

This part has the following chapters:

- *Chapter 5*, *Exploring Structural CI/CD Design Patterns*
- *Chapter 6*, *Deployment Strategies for Structural Design Patterns for CI/CD*

5
Exploring Structural CI/CD Design Patterns

In this chapter, we'll dig deeper into **structural design patterns** and how software teams can leverage them to design efficient CI/CD pipelines to develop, test, build, and deploy modern software. This chapter describes two different design approaches – the **monolithic approach** and the **polylithic approach**. The monolithic approach to repository design is where we put all the components of the entire application in one repository or pipeline. The polylithic approach is where we define a clear and strong separation between components or types of executions.

In this chapter, we will also explore dependencies between CI/CD patterns and their components, tools, and code, as well as the modularity of code. Lastly, we will conclude by highlighting the importance of the level of control and considerations around it.

By the end of the chapter, you will have a solid understanding of applying structural design pattern concepts to make the CI/CD implementation more scalable and efficient.

We will cover the following topics in this chapter:

- Introducing structural design patterns
- CI/CD design patterns principles – monorepo and polyrepo scalability
- CI/CD design patterns – dependencies between pipelines
- Key components – tools, code, and modularity
- Key components – level of control

Introducing structural design patterns

Structural design patterns are focused on simplifying the arrangement of and relationship between the components or elements of CI/CD and removing the complexity of CI/CD. They also facilitate the scalability of CI/CD so an organization can adopt it with ease. Structural design patterns enable

developers to explore ways to simplify, integrate, aggregate, extend, and structure the key components of CI/CD for scalability. Let's have a look at these key components:

- **Common components layer**: The design of the delivery process contains a few components that, when organized, interfaced, and connected in different ways, form complex relationships. It builds the foundation for reusability, inheritance, and so on. Examples are the number of designed repositories, databases, meta-data, and tools.
- **Pipeline layer**: This encompasses the basic structure of the pipeline for the project and extension capability if new functionality is introduced or projects are added.
- **Deployment layer**: This represents the deployment stages and techniques.

In this chapter, we will detail each of these parts. For now, let's see a very simple depiction of the structural CI/CD design pattern in the following diagram:

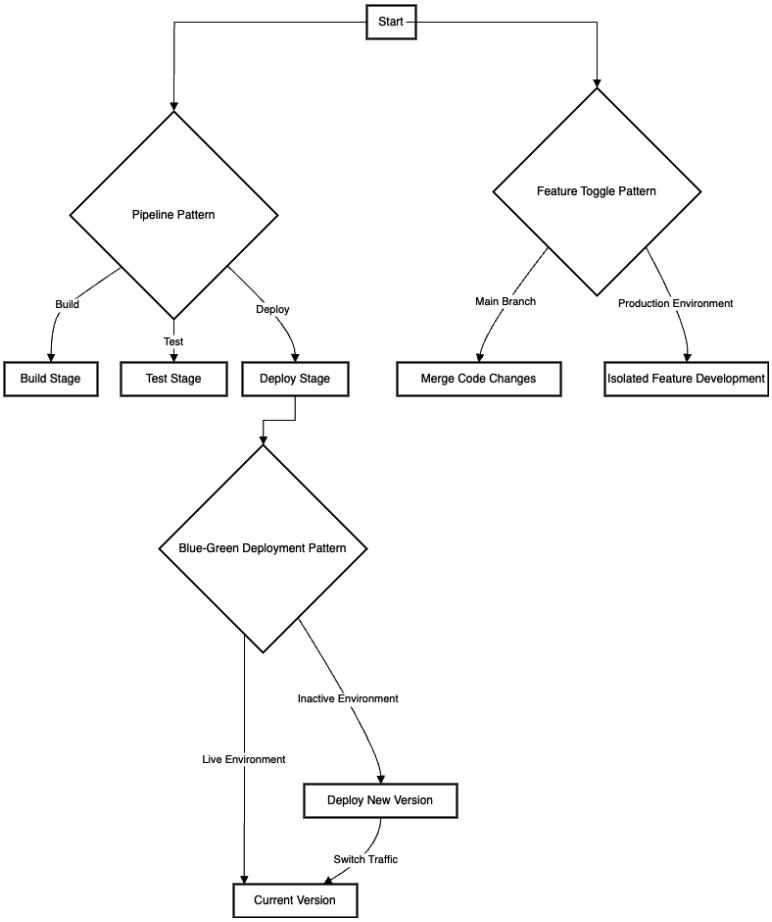

Figure 5.1 – The basics of structural CI/CD design patterns

The pipeline layer node structures the CI/CD process into build, test, and deploy stages. The **blue-green deployment** layer node represents a pattern where two identical production environments are maintained. One environment is live (**Current Version**), while the other is inactive (**Deploy New Version**). Once the new version is fully tested and ready, traffic is switched over to the new environment. The **feature toggle pattern** node represents a pattern that enables teams to merge code changes into the main branch and develop and test features in isolation in the production environment.

CI/CD design pattern principles – monorepo and polyrepo scalability

In this section of the chapter, we will look at two different approaches to storing and managing code – **monolithic repository** (**monorepo**) and **polylithic repository** (**polyrepo**). We will learn the differences in scaling between the two.

Before we dig deeper into this, let's quickly recap the two concepts:

- **Monorepo**: In this pattern, all code is stored in one code repository in a **version control system** (**VCS**). It is important to understand that this doesn't mean the application code only, but also configurations and infrastructure.
- **Polyrepo**: This pattern enforces separation. Projects contain many repositories and every repository stores code for one application.

The following diagrams represent examples of monorepos and polyrepos. Understanding how the directories are designed is less important than getting a high-level picture of the principles. The following figure presents the potential setup of a monorepo. We can clearly see that all separation between components is done by having separate directories:

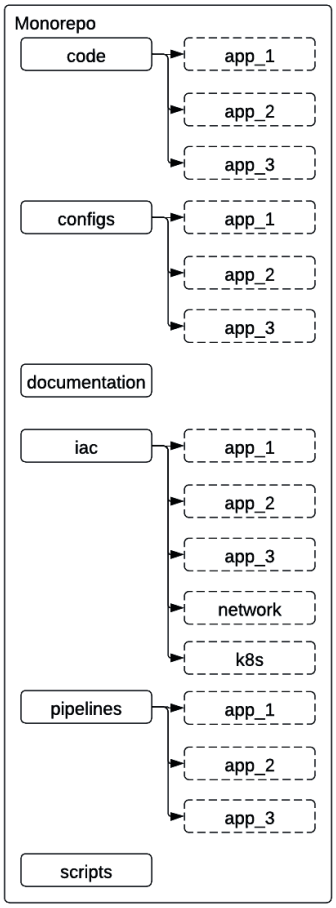

Figure 5.2 – High-level monorepo setup

Now, the following figure presents the potential setup of a polyrepo:

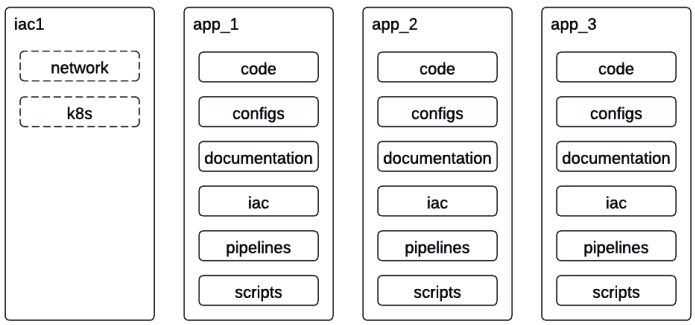

Figure 5.3 – High-level polyrepo setup

Every repository is smaller and dedicated to one application. The code is easier to manage, although there are many repositories to control.

This sounds simple, doesn't it? Well, let's make it more complicated then. As the world is not simply black and white, we can find multiple approaches that fall between monorepos and polyrepos. If we wish to be very strict, most of them, if not all, will fall under the polyrepo definition, but here we can find multiple approaches to achieve the best results.

First, we need to understand how the separation between code repositories is influenced by architecture. There is no need to explain monoliths here, as this seems to be fairly simple – a monolithic application is stored in a monolithic repository. This often leads to a very complicated and very large code base. The solution to this is the use of submodules. Although this option solves the problem with the size of the repository, it overcomplicates the daily work and the need to control all submodules.

However, we can also encounter a setup where code is stored in one repository but the configuration of the application is separated to another. What's more, the infrastructure is stored in yet another repository. There might be multiple reasons for this situation. We will go through some of them soon.

Another architecture that we need to consider in repositories and the code design process is microservices.

Microservices, as distributed systems, are key to many architectural approaches, starting with the well-known containerized application, to serverless, and even to semi-no-code applications (here, I'm thinking about the use of AWS Step Functions, for example). Every approach has a different separation between components.

There is one more aspect related to microservices architecture that needs to be properly designed – databases. In most mature microservices architectures, we tend to separate databases per microservice (or per group of microservices acting as a single service).

This is important for CI/CD design because the chosen architecture approach and code storage method will dictate the specific CI/CD design requirements. What are the most common approaches here? Let's have a look:

- **Full separation**: In this case, every microservice is stored in a separate repository. The benefit of this solution is that we have a truly microservices approach, from code to running workloads in production. Every microservice has a separate code base and pipelines that deliver the code to environments.
- **Separation by service type**: Let's consider an application for booking flight tickets. In this type of application, we can encapsulate a couple of services, such as the user (login, user management, user data, etc.), booking service, payment service, and airline management service. Depending on the technology selected, we can design one container for each service, or multiple of them. In the case of the serverless approach, we will end up with many functions per service.

  Separation by service requires less effort for database design and management and makes the configuration simpler; configuration is related to the service, not each container.

- **No separation (monorepo)**: In this case, we have one repository where all code for all microservices exists. From the VCS management side, it looks like the simplest solution. It is also very convenient from a development perspective – all configuration files, scripts, and so on are stored in a single place.

Design examples for the monorepo and polyrepo approaches

As we can see, what approach is selected has a very deep impact on the design of the application and development work. Also, of course, it has a huge impact on the design of delivery pipelines.

The design of the delivery process contains a few elements. We have not split the functionalities between CI and continuous delivery (or deployment) here; we want to focus on the procedural aspects of the delivery process. What happens when the developer commits the code? The code lands in the repository, obviously. After that, some magic starts to happen:

1. We test the code. In an automated process, we have pipelines or parts of pipelines responsible for test execution.
2. We deliver to lower environments. By lower environments, we mean development or QA environments.
3. We execute the pull request process. In the continuous deployment process, this also requires some automation – therefore, some actions are executed through pipelines.
4. We merge the code after the pull request is accepted. In this step, some tests can be executed again.
5. Finally, we deploy on higher environments (such as the release candidate, pre-production, and production). This also requires some actions to be done in the pipelines.

As we can see, each delivery process requires multiple steps. We identified only these steps, which we differentiate usually. Even if one pipeline executes all the preceding steps, we have a strong tendency to see them as separate components of the delivery process.

This leads me to another approach to the construction of the pipeline. In many projects, we can see multiple pipelines that run on different triggers. One pipeline is executed after the developer commits code, another during the pull request, and another after the merge to the stable branch. There can be more pipelines than these three mentioned.

Continuing the naming convention, we can think about these approaches in a similar way to monorepos and polyrepos – a monolithic pipeline and a multi-pipeline project.

Let's look at the high-level construction of a mono pipeline.

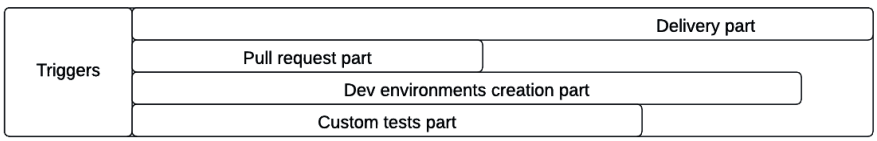

Figure 5.4 – High-level monorepo pipeline

What is clearly visible is how the way the pipeline is executed depends on the type of trigger. This means that the pipeline is constructed with many conditionals. In effect, we have one pipeline that seems easy to control, but the pipeline itself is complicated and it is very hard to properly measure the specific type of execution (such as pipelines triggered by pull requests).

In the following figure, we can see the multi-pipeline pattern:

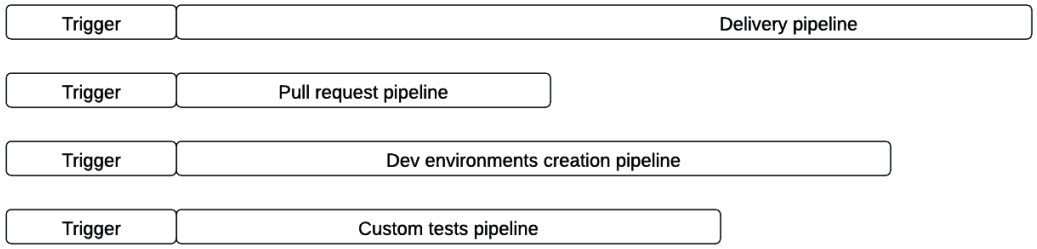

Figure 5.5 – High-level multi-repo pipeline

In the preceding diagram, we see the multi-pipeline pattern. In this case, we are able to quickly measure and understand specific executions. However, the downside of this solution is that we must manage many pipelines per project.

To have a better understanding of this, we should explain the purpose of these pipelines:

- **Delivery pipeline**: This is the core pipeline of the setup. It is our main *transmission belt* for the code, from repository to production. It covers all deployments and test and quality gate executions.
- **Pull request pipeline**: Although the most common scenario is that after the developer creates a pull request, they ask someone to do the review, it is not something we want to have in our automated process. The pull request pipeline is triggered by the creation of the pipeline and provides all the required tests, such as code validation, code coverage, and unit test execution. At the end, this pipeline comments a pull request with information about the status and blocks or allows the merge.
- **Dev environment creation pipeline**: This is an interesting approach that is becoming more and more popular. In this scenario, when developers create a branch, the CI/CD process automatically spins up the dedicated environment, where developers can test their work. When the branch is removed (deleted or merged), the environment is removed as well.
- **Custom test pipeline**: In this scenario, we create pipelines that can perform specific actions, such as E2E tests. Very often, these tests are very long and shouldn't be part of delivery pipelines.

We stated earlier that the selection of the approach to the collaboration model in a VCS (monorepo or polyrepo) is very important. This determines the approach to the delivery of the applications. It is less crucial to define the approach to delivery itself (monolithic pipeline or multi-pipeline), as it can be relatively easy to change.

But what we must remember is scale. While it might be easy to switch between approaches when we have one or two repositories, it becomes harder, if not next to impossible, to do so when projects and organizations grow. This impossibility isn't related to technical aspects because, in this field, there is no difference between 1 and 1,000 repos; it is related to cost and effort, for example, where we have hundreds of repositories and we need to redesign all of them. In the scenario where we code our pipelines manually, one by one, we will be limited by the number of people who can do the job – and it will certainly not be their sole task. Additionally, we will need to think about how we will perform the stability tests of each pipeline. Finally, some focus should be put on the maintenance windows when this work can be done. Especially in intensively developed projects, this work can interfere with the development and delivery process.

This is why we stated that the change in approach to managing pipelines is possible but will not always be economically sound.

There are consequences to choosing any approach. However it sounds, it is very important to keep the scaling option available. We can bet that every reader of this book has been in a situation where the approach between two identical tasks was changed, without the reason being documented. On one day, task A is done in one way. On another day, task B is done another way. Well, yes, these are two different tasks that belong to two different projects, but in fact, they are the same, considering the purpose of these tasks. Despite knowing this, people may still want to try a different method. This leads to inconsistency, creates inaccurate processes, and, most importantly, prevents the organization from collecting comparable measurements to understand the process and find gaps and improvements. This leads to a chaotic process at best.

Effective scaling requires consistency, that is, consistency in the development of pipelines, consistency in the management of these pipelines, and, finally, consistency in the measurement of these pipelines.

What aspects should be considered during the design process?

- The starting number of designed repositories.
- The assumption for growth in the next months and years, if possible to assess. These two categories will inform how many pipelines are needed.
- The development and delivery approach. This comes down to how many deployments will be executed per day, or per hour. In fact, this refers to not just the deployments but any actions that will trigger the pipeline.
- The predicted duration of pipeline executions. It is not a mystery that pipelines can run for a long time. A few examples are as follows:
    - **E2E tests**, where all functionalities are tested one by one.
    - **Performance and load tests**, where we try to find the limits of our application. This means this execution can be very long.
    - **Stability tests**, which can be the longest as we want to check the behavior of the system in the long run.

    - **Infrastructure implementation through Infrastructure as Code** (**IaC**), while it doesn't sound like it fits into this list, is used by many organizations whose Terraform or OpenTofu projects are not optimized, where state files are so huge that the execution of the `plan` command can take even an hour.

- Team capabilities, asking whether the team is able to understand and maintain multiple pipelines per project and whether the team is able to manage the tools needed to deliver code to production.
- The tool and its capabilities, which means understanding the capabilities of the tool that will be used. We can simply start with self-hosted and standalone Jenkins. This gives us two concurrent runs (in the default configuration). This number implicates the *throughput* of the pipeline's execution in time. In other words, how many pipelines can be executed in a given time? The next question here is the potential to grow the tool. In SaaS tools, your throughput can be limited, it can be subject to license limitations, license upgrades can be very expensive, and so on.

Now, let's understand the dependencies between pipelines.

CI/CD design patterns – dependencies between pipelines

Simplistic pipelines, that is, basic ones, do not need complicated design or organization. This comes with the growth of CI/CD usage. Introducing dependencies in the pipelines in the early stages of CI/CD development will lead to bad architecture, error-prone workflows, and delays in executions.

The vertical scaling of pipelines might introduce the need for dependency management. However, in this case, it is hardly expected.

What is the vertical scaling of pipelines? For example, when new steps are added, such as new test structures, or new environments must be configured with their quality and acceptance gates, the pipeline grows vertically. Simply put, it becomes longer and longer. There are more steps and decision points to execute.

Another type of scaling, the one that is of most interest to us, is horizontal scaling. In this pattern, we do not grow the pipeline; we add more of them. There might be multiple reasons for this:

- **New projects or applications**: When we add new applications to the platform our team is developing, we need to deliver these new applications to the system. This requires adding new pipelines to the CI/CD tools.
- **New process implemented**: For example, we could have projects (or applications) that are each managed by two pipelines: the pull request pipeline and the deployment pipeline. We might add another pipeline, let's say the pipeline for performing specific types of tests.

It is important to remember that in all these cases, we scale horizontally. The design pattern selected for CI/CD system evolution in this case is to add new pipelines, not extend existing ones.

In most cases, these pipelines are triggered by an external action. Mostly, it is the trigger from VCS, which happens after the commit, merge, or pull request creation. Some of the pipelines are scheduled to be run at a specific time.

In some cases, however, we need to trigger one pipeline right after another. This creates a chain of execution, which complicates the sequence and can blur the understanding of the process. It is important to understand the best practices here to avoid badly designed pipeline execution chains.

A deeper understanding of this dependency comes with the modularity of the pipelines; however, here we touch on the dependencies between pipelines only.

What are the best practices when we design the dependencies between pipelines?

- **Keep the pipeline autonomous**: This case is very similar to microservices architecture. In microservices, each service is autonomous and can perform its action alone. The same principle applies here. If a pipeline must deploy a microservice, it should do it from start to end.
- **Keep the process autonomous**: What was said in the previous point may lead to the wrong conclusion that we should have multiple pipelines per function, such as collecting code, collecting artifacts, testing, and deploying. This is not the case, of course. The goal of the pipeline is to execute the whole logical chain of actions. This pattern will be described more closely in the *Modularity* section later in this chapter.

Let's look at the execution of a logically completed structure.

CI/CD by nature is a process that contains two elements: continuous integration and continuous delivery (or deployment). The boundary between those two is also very well defined. To ensure that we are on the same page, let's check again where this boundary is. CI ends when the artifact is created and stored. CD starts with collecting this artifact and processing it further.

This clear boundary between CI and CD doesn't mean that two pipelines must be defined, one for CI and one for CD. In fact, most pipelines out there cover both processes in one execution (and one definition file). There are systems, however, where this distinction is obvious.

One example of these systems is Azure DevOps. Although you can build the pipeline that covers a whole delivery, from collecting the code after commit to deployment (you will use build pipeline for it), it is possible to create a distinction between those two elements of the process by using the build pipeline for CI and the release pipeline for CD. In this case, you have two distinct pipelines that are connected through the trigger. This trigger is internal to the Azure DevOps system.

Another option is when two different tools are used. The distinction between CI and CD pipelines is deeper and based on two separate tools. A good example here is a less popular but still used approach, where the CI pipeline is executed in Jenkins, an artifact is created and stored in third-party artifact storage, and the CD pipeline is triggered in Octopus Deploy, for example. This trigger is usually configured as an API call or webhook integration.

While this solution might be selected after considering all the pros and cons, you still need to know the consequences, especially if scaling processes are on the table. Let's look at the potential consequences that you have to keep in mind when you select your approach:

- Every time a new workflow is added, multiple pipelines are in fact created. This leads to huge chaos and lost visibility in the long run.
- Workflow management is complicated. There are hundreds of pipelines (add to this many different purposes of these pipelines) as well as a number of tools. The low visibility of connections between pipelines means a great amount of effort is needed to manage and maintain the system. It doesn't sound complicated when you have 10 pipelines, but when you have to manage hundreds or thousands of pipelines, this becomes a huge and overwhelming effort to control and understand. Of course, you may argue that if there are so many pipelines, the proper pipeline-as-code approach should be implemented. Of course, this is a valid point. But even with full automation, the effort needed to control the environment is similar.
- Monitoring and collecting metrics is extremely hard. This problem lies in the way pipelines communicate. While you can check the configuration and execution of the pipeline, monitoring systems receive a simple value for specific metrics. If you have two different pipelines in two different tools, their metrics will be collected in separate buckets. Of course, you can try to avoid total confusion by implementing naming conventions for the pipeline's names. When the pipelines that belong to one workflow share the main part of the name, it helps to find the correlation. Still, it is not something that you have straight out of the box, and it adds more turmoil to the process. Don't misunderstand, though. The established naming convention is a *must* for growing systems. What is considered here is related to understanding workflows built with multiple pipelines.

These are significant factors and must be considered during the design of the CI/CD landscape.

This section of the chapter showed the principles of the design of repositories and pipelines. In the next part, we will take a look into the components that, when designed properly, give the proper starting point for advanced scalability and security.

Key components – tools, code, and modularity

In this section, we will explore why and how important it is to have proper processes defined and enforce integrity inside projects, teams, and the whole organization. We will discuss the aspects that need to be considered when CI/CD tools are selected, what the key elements to evaluate are, and what the pros and cons of the selected approach are. Similarly, we will evaluate the code structure and modularity of our technology's process.

Tools

In this part, we delve into tool selection and what we have to consider when we design the CI/CD system. We will not discuss specific tools, their pros and cons, and configurations. This is purely a design and high-level architecture discussion. As an outcome of this, we should have a list of **quality attributes** (also known as **non-functional requirements**), which are the input to proper tool selection.

Selection of the proper tool does not seem to be crucial. *We can always introduce some new software* is a common way of thinking. Yes, we can; however, we need to think about how this will affect established processes and the work, delivery, and stability.

As the CI/CD framework for scale (`https://cicd.run`) states, the selection of tools as a concept lies in the common area of the business (organization) and process (execution). In other words, the tool you select might impact both sides – the technical teams and also the business. Therefore, both must be aligned.

Of course, we can start simply by selecting a tool that teams know. The question is, however, whether this is a good choice for scale.

What questions should we ask?

- **Team capability**: First, we need to understand our teams and their plans for growth:
  - Is the existing knowledge in the team(s) enough for them to work effectively?
  - Do we plan to hire more engineers?
  - What is the predicted growth of the project/teams?

  Answers to these questions will lead to another set of decisions we need to make.
- **Platform type**: Here, we need to decide what compute model we should select. However, it is not that simple. We need to consider more aspects, such as the following:
  - Security standpoint
  - License costs and management
  - Regulatory expectations

  These are the most important. When we have this list, we are able to select our approach:
  - **Software as a Service** (**SaaS**), refers to dedicated platforms that require us to work with external services. This brings a lot of flexibility, allowing us to focus on the company's product and reduce costs and administrative efforts.

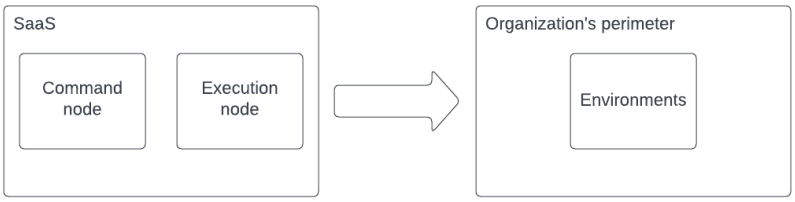

Figure 5.6 – High-level structure of the SaaS model

This diagram presents the structure of the SaaS model. The organization is responsible only for providing proper connectivity and security of the connections and the workload itself.

- A **hybrid approach** is a common scenario selected by organizations where security is more restricted. It means that the majority of the management of the CI/CD platform is done by a third party (as in the SaaS model), but part of the platform (it will mostly be the execution node) is hosted in the organization's perimeter. An example is shown in the following figure:

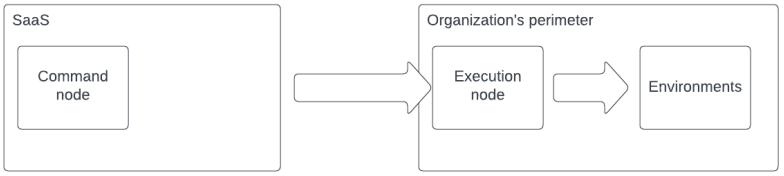

Figure 5.7 – High-level structure of the Hybrid model

In this scenario, the organization is additionally responsible for providing execution nodes and cares about their configuration and security. The SaaS platform acts as a command center, so the organization doesn't need to care about the engine of the CI/CD platform.

- **On-premises** means that we host everything on our infrastructure. To do so, we need the infrastructure, physical security, and skills to be in-house.

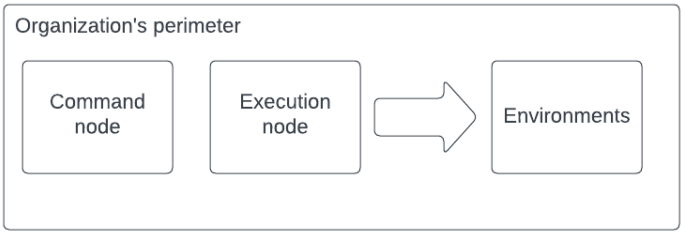

Figure 5.8 – High-level structure of the on-prem (or self-hosted) model

An on-premises setup is done fully in the organization's data center. This is the most flexible solution from the perspective of potential configuration, but it is also the most demanding in terms of skillset and initial cost allocation.

Scaling the tools seems to be easy but requires knowledge, both business and technical. What options are there related to scaling CI/CD tools? Let's have a look:

- **Scaling SaaS tooling**: This mostly requires a proper understanding of the pricing models of SaaS vendors and our expectations. We need to understand what the base pricing is. How many runs are there? What is the execution time? How many pipelines are there? Or, maybe, how many resources are managed, as we see in HashiCorp's Terraform Cloud offering? We need to have full knowledge of this in order to be able to estimate the limitations and the business reason for scale in this case. It is a reasonable action to have a meeting with the vendor and receive information about this, as well as about the potential performance of the scaled solution. Remember, you are not in charge of the SaaS platform. And believe us, you don't want to be surprised that your workload is throttled or limited and the delivery performance is poor.
- **Hybrid tooling**: This is very often selected with the view of it potentially serving the best of both worlds, SaaS and on-premises. But the fact is that this approach also imposes the worst of both. Having a hybrid solution doesn't give us full charge of it and requires us to make business and technical decisions. Also, a technical team must be formed to control the part of the solution that is under our control.
- **On-premises tooling**: How do we build our solution? How are the execution nodes configured? How many of them do we need? How do we network all components? How do we architect the solution? Who will keep the platform up and running? What skills do we have and need for it? How do we scale? What will the measurements be? These and many more questions must be answered to achieve an effective solution. It requires the biggest investment upfront and proper architectural decisions.

Code

For many, this will be an obvious component. Even if DevOps and development teams understand this part very well, it still needs some explanation and organized knowledge.

Of course, every CI activity starts when a developer commits the code to the VCS. Here is where the first decision arises. How do you do it properly?

We mentioned before how monorepos and polyrepos can be built. The more complex part of this picture is the branching strategy.

All developers have an approach and strategy that they feel most comfortable with. But taking a helicopter view of this, from the perspective of a manager who is responsible for delivery at scale, personal preferences should be recognized as only one in a list of reasons why a specific approach is selected.

Branching strategies

Before we explore how to scale our code, we need to ensure that we are all on the same page about the most popular branching strategies:

- **Trunk-based development**: This is the fastest way of delivering code to production. Trunk-based development is based on the principle that changes should be made to the main branch (called the trunk). This implies very short-lived branches that are merged constantly with the main line.
- **Gitflow strategy**: Gitflow can be easily placed on the second side of the *easy and quick delivery* coin. It was one of the first strategies that allowed agile teams to work very quickly. It is well suited for larger teams, and when multiple versions of the product must be maintained. However, this adds huge complexity to the work due to all the intermediate branches, such as hotfixes, releases, and features. An important point to remember is that Gitflow has two main branches. There is the development branch, which acts as the main branch for daily work, and the main branch, which is updated from the development branch and acts as a single source of truth and workable product.
- **GitLab Flow**: GitLab introduces its own strategy, which sits in between Gitflow and trunk-based development. In this approach, our main working branch is **Main**; however, we have referenced branches such as **Production** and **Preproduction**, which represent the current state of these important environments.
- **Branching strategies – revisited**: Of course, there are more strategies. However, if we consider the management of the organization at scale, other factors become more important than *just* which branch is the one where we should merge the code. These branching strategies are related to the delivery and deployment model, selected for the organization (and not for a single repository), and should reflect the approach to introducing changes to production.

  The second aspect here is the architecture of the application we deliver. Monoliths are delivered differently than microservices. If the architecture is deeply decentralized, the method of delivery will be different than if there is one central point of operation.
- **Scale your branching strategy**: To properly scale the code, integration, and delivery, we need to switch our way of thinking to how the change is introduced to the customer. This perspective was presented in one of the blog posts at Atlassian.com (`https://www.atlassian.com/agile/software-development/branching`), which took the perspective of the agile team.
- **Release (version) branching strategy**: In this strategy, we focus on releasing a specific version of the product. Very often, it includes more changes that are significant for the customers, alongside fixes and small updates. It requires more coordination and a specific approach to the branching and promotion of the code to the next steps during the process. For this approach, trunk-based development doesn't sound like a very good fit; Gitflow, however, looks much better.

- **Feature branching strategy**: This approach is very popular when it comes to microservices. It allows developers to *hide* some features that are under development or testing and enable them only when the feature is finished.

  Do not confuse the situation when some features are enabled for specific users (for example, different tiers of pricing) with this approach. The feature branching strategy allows us to strictly control and release the code.

- **Task branching strategy**: The previous strategy was most suited for Gitflow. This one might be best for teams working with trunk. Each task has its own short-lived branch that is immediately merged when work is finished.

  The approach can be modified and used as an Epic branching strategy as well. This gives organizations a good way to align their approach to code with project management.

Modularity

In almost every situation, scaling requires modularity to be manageable and be able to grow further. CI/CD is the same. Scaling is always triggered by specific causes. It might be a growing application, number of services, number of tasks to perform with CI/CD, number of environments, or number of team members and teams. With the last couple of reasons, there might be changes in responsibilities as well. When the organization grows, it might introduce a platform or SRE team. This will shift the responsibilities in terms of designing, creating, and managing pipelines. Let's list the patterns that should be implemented in order to establish a proper process of modularity:

- **Self-service adoption**: Self-service is sometimes perceived as the holy grail of well-functioning organizations. *Stay in your own toolset, but use the power of all tools around* is how we can describe this approach. Similarly, like in self-service restaurants, customers (developers) collect the food they want, and they don't need to be familiar with any cooking practices. They operate in a specific environment, where boundaries are set by the restaurant (the organization) and all food (tools) is provided by the cooks (platform team members). There is limited choice, but we can be sure that the ingredients and measurements are correct and we won't have any problem when we consume the food (I wish that were the truth everywhere!). The purpose of self-service, thus, is to enable developers and provide solutions that should help them to improve their work without increasing cognitive load with new technologies and tools.

  Without going into the details, we can simply say that the self-service approach modularizes the functions of teams. In other words, multiple development teams rely on one platform team that delivers specific solutions and modules to them.

- **Architectural modularity**: Here, we do not discuss how the code should be constructed for microservices or serverless applications; rather, we look at the big architectural blocks.

  CI/CD does not refer to the tools that are used for application code. It is also used for infrastructure, security, or even CI/CD. It is not an uncommon scenario for the whole CI/CD setup (with underlying infrastructure, tools, and pipelines) to be created and managed automatically by one administrative pipeline.

In architectural modularity, we can easily debunk a few different areas:

- **Infrastructure**: When infrastructure is deployed together with applications, it leads to trouble in the future. Infrastructure (networking, connectivity, server and database layers, etc.) needs its own way of delivery. Any change in infrastructure can introduce a problem in other parts of the system. Additionally, changes in infrastructure are less frequent than in the code.

  However, we need to separate different layers of infrastructure. The network layer is a foundation for applications that use the network, but the deployment of the application doesn't impact the network layer. There are different situations where the serverless and microservice approaches are used, where one microservice can deploy a database, storage, or API with the function. But in this case, we should treat this infrastructure as tightly bound to the code. Also, this application-related infrastructure will only be deployed within the boundaries created by the main infrastructure components.

  IaC codifies the infrastructure and allows us to make the shift needed for modern applications. Now, we can keep infrastructure code closer to application code and this enables us to quickly and efficiently deliver microservices.

  Where IaC also plays a huge role is in modularization itself. IaC allows us to create templates for infrastructure. Like in the serverless example from before, each serverless function should have its monitoring components, storage, API to communicate with other services, and so on. When this is standardized and modularized, the organization can achieve better performance.

  The best way to enable the proper delivery of IaC is if the organization uses dedicated delivery tools for infrastructure. A few examples here are Spacelift, env0, and Terraform Cloud.

- **Microservices**: Containerized applications, especially when run on Kubernetes, can be modularized by leveraging the functionality of the Kubernetes platform itself and tools such as Argo CD or Flux. Full enablement of **GitOps** enhances the modularization on different levels, such as service, infrastructure, and connectivity. Platform teams might control the deployment modules and duplicate the code for different teams/applications, keeping the core of the code the same. What load balancer should be used, how to connect to databases, and what storage should be used are aspects that can be defined once and parametrized.

These two examples show the power of modularization. The modularization selected for teams or applications influences the CI/CD design. Here, we can implement modularization as well using the following points:

- **Pull requests**: While it might sound strange, pull requests can be modularized as well. First, the specific approach to the definition and configuration of a pull request can be implemented. Specific checks can be included in the process (for example, code coverage, unit tests, and quality scans). These checks can be configured globally and used as *pluggable* modules into the pull request.

- **Third-party components**: Many CI/CD tools allow us to use components provided by other organizations, vendors, and individuals. In fact, modern solutions today use this approach

on a daily basis. Take a look at GitHub Actions or Azure DevOps. Everything that is defined in the pipeline's code is constructed using tasks or actions. These are, in fact, modules stored somewhere and maintained by someone. We just take the module, agree on its function, and parametrize it.

- **Reusable components**: These are the same as third-party components but written and maintained inside the organization. This gives more control but also requires skills and time to maintain it.
- **Reusable pipelines**: Many tools (such as GitHub Actions) allow us to use defined pipelines as modules. This allows us to share the specific workflows and manage some elements of the delivery on one central level.

Modularity increases the quality of the process and decreases potential problems. It also makes the process manageable at scale. Let's look at the following diagram:

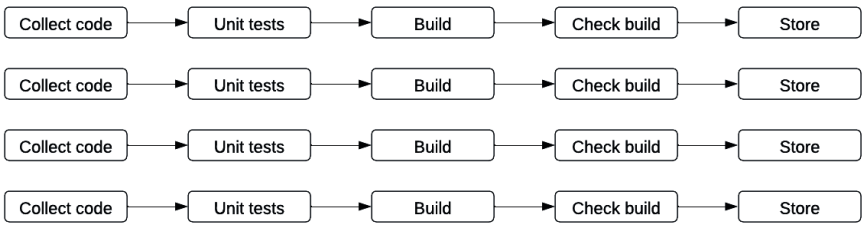

Figure 5.9 – High-level example of a CI pipeline

This diagram shows an example of a CI pipeline. Of course, it is very simplistic, but it is enough for our purposes. In this approach, we build every step per pipeline. We code everything directly, without using any modules. What are the downsides of this approach? Let's have a look:

- **Multiplication of code**: Every time we want to do something, we need to write the code again.
- **Error-prone code**: As we write the code again and again, the risk of bugs, mistakes, or typos is high.
- **Tracking changes is nearly impossible**: Imagine we need to change something in the code. We'd had to make this change in all pipelines. More pipelines mean more work and a greater risk of forgetting about one or more pipelines, as well as a greater risk of introducing bugs or typos. It also means more overhead with the management of the changes.

In *Figure 5.9*, all elements seem to be exactly the same. Well, yes, the steps are indeed the same, but how do we ensure unification if we have to write every pipeline from scratch? Let's take a look at the following code. This example shows how we can log in to AWS from GitHub Actions when we code everything directly in the pipeline:

```
-   name: Install AWS CLI
    run:  |
```

```
      curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip"
-o /tmp/awscliv2.zip
      unzip -q /tmp/awscliv2.zip -d /tmp
      sudo /tmp/aws/install --update
      rm -rf /tmp/aws/ /tmp/awscliv2.zip
- name: Store artifact
  run: |
    aws s3 sync ./artifacts s3://artifactsbucket
  env:
    AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
    AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
    AWS_DEFAULT_REGION: 'eu-west-1'
```

We see here two steps:

1. We download and install the AWS CLI.
2. We copy an imaginary artifact to an S3 bucket. We use secrets for the access key and secret key.

Imagine now that you have 100 pipelines and you have to change the version of the installed CLI or the bucket location. You would need to go through all the pipelines, change the values, and ensure you did it correctly everywhere. This adds more effort to the task and, in the end, it will be clear that the overhead of controlling the task will be higher than the effort needed to complete the task.

Now, let's look at this diagram:

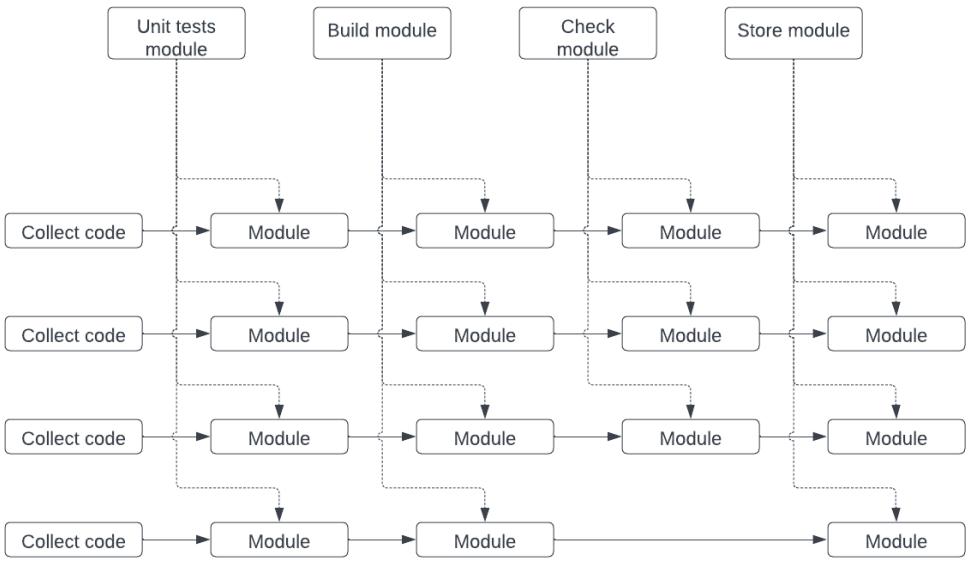

Figure 5.10 – Modularization of the pipeline

While this diagram looks more complicated than the one shown previously, the work that can be done using this approach is not. It is always this way in the IT world – if we simplify something here, something there must get more complicated. The goal is to make repeatable processes simpler. And this is what we achieve by having a more complicated process of architecting the solution.

Let's make our lives easier now. We'll take the preceding code and implement it as a reusable workflow. A reusable workflow is a feature of GitHub Actions that allows us to reuse workflows in other pipelines.

First, let's encapsulate the code as a reusable workflow:

```
name: Reusable workflow example
on:
  workflow_call:
    secrets:
    AWS_ACCESS_KEY_ID:
      required: true
    AWS_SECRET_ACCESS_KEY:
      required: true
jobs:
  example:
    runs-on: ubuntu-latest
    steps:
      -   name: Install AWS CLI
        run:  |
          curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.
zip" -o /tmp/awscliv2.zip
          unzip -q /tmp/awscliv2.zip -d /tmp
          sudo /tmp/aws/install --update
          rm -rf /tmp/aws/ /tmp/awscliv2.zip
      - name: Store artifact
        run: |
          aws s3 sync ./artifacts s3://artifactsbucket
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          AWS_DEFAULT_REGION: 'eu-west-1'
```

With this stored in a separate repository, we need to implement just one line in all application workflows.

```
call-workflow-passing-data:
    uses: organization/repo/.github/workflows/reusable_workflow.yml@
main
```

Whoa! This is indeed what we need in every pipeline! With this simplification of the code in the pipeline, will you now choose the more extensive process of modularization? Well, we bet you will.

What did we achieve with this approach?

- **Simplified the workflow's code**: Less code in each workflow means better readability. The preceding example is very simple; you will see even more benefits with more complicated scenarios.
- **Easy changes**: In this case, the change of the bucket is simple and quick and will be available for all workflows immediately.
- **Easy tests**: When changes are applied, tests are needed only in the shareable pipeline.
- **Simple control**: The change and its control must be done in one place only.
- **Decrease of cognitive load**: In the scenario when developers are responsible for all the code in the repository, including workflows, the management of simple changes such as the bucket name for all repos is a nightmare. Also, a proper knowledge of workflow syntax is needed.
- **Version the workflow**: This requires more work; however, developers can easily decide which version to use (in this case with tags, branches, or even commits).

As we can see, modularization provides the potential of not only automating work but, most importantly, unifying and decreasing cognitive load.

We have implemented a very good approach so far, but there is also another way to modularize our pipelines: using the work done by others, outside our team or organization.

What we've done so far in this section is write and modularize the code, but the code must still be written by us. Let's take a look at this example:

```
- name: Login to AWS
  uses: aws-actions/configure-aws-credentials@v1
  with:
    role-to-assume: ${{ secrets.AWS_CA_ASSUME_ROLE }}
    role-session-name: GitHubActionsSession
    aws-region: eu-west-1
- uses: jakejarvis/s3-sync-action@master
  env:
    AWS_S3_BUCKET: ${{ secrets.AWS_S3_BUCKET }}
    AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
    AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
    AWS_REGION: 'eu-west-1'
    SOURCE_DIR: ' artifacts'
```

This example shows how to use Actions from GitHub Marketplace. GitHub Marketplace allows creators and maintainers (very often a vendor of the service we use, like the first action here – `configure-aws-credentials` – to configure access to AWS).

What are the advantages and disadvantages of this approach? Let's have a look:

Advantages	**Disadvantages**
No need to maintain the modules	More code to write in the workflow, compared to reusable workflows
Deep knowledge of the development of specific solutions is not mandatory (this might be important for the SRE or platform teams)	No control over the module code
Easy and clear versioning	Security should be included in the selection process for the modules (especially in regulated environments)

Table 5.1 – Advantages and disadvantages of modularization

We can clearly see that modularization has extreme benefits over coding every pipeline from scratch. It saves time and cognitive load and decreases the potential for mistakes and errors. It allows the organization to better control the CI/CD executions at a high level and ensure the unification of the processes.

Modularity is a big enabler of scale. In fact, one cannot effectively scale one's workloads without modularity. This is valid for the whole CI/CD landscape – pipelines, tools, and processes. Now, let's have a look at the levels of control.

Key components – level of control

Control and access control are very important aspects. This book is not about security, but it is impossible not to touch on the topic, especially when we talk about scale.

Who can administer the CI/CD platform? Who is the user? How do we separate projects/teams? Who can access what?

Many questions can be asked from an access and control perspective. Without delving into the monitoring side, let's look at one security pattern that should be implemented in the CI/CD landscape – **role-based access control** (**RBAC**).

In short, RBAC enforces assigning permissions to users based on their role in the organization. With this approach, we do not manage users individually, and we do not need to think about what kind of access the imaginary John Doe should have. Everything comes from the central level of access control, and based on it, we add expected actions to groups. Let's explore the following diagram:

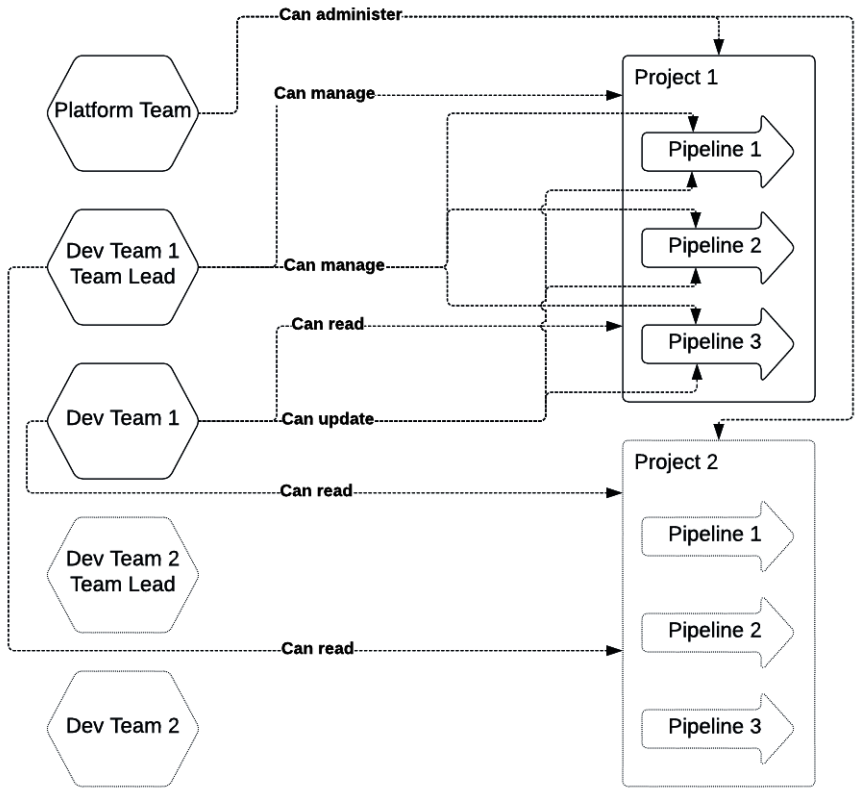

Figure 5.11 – RBAC in CI/CD system in action

We can see a simplified example of RBAC here. The platform team can administer the project in a CI/CD tool. The development team can effectively work with their pipelines and is able to see other projects where they do not contribute to the code base. The team lead can also see the configuration of the project. We can use this same schema with other teams and projects.

Of course, it leads to a complicated structure of access, but what we said about modularization is correct here too. Modularity in access control requires complicated architecture. Although achieved benefits are worth the effort.

Summary

In this chapter, we explored the elements needed to properly and effectively scale CI/CD. Scaling is easy to some point. However, adding new pipelines will not always work. There will be a point where we meet tool limitations, such as performance and storage capacity. With these issues, we will face a skillset problem. The team must know how to react and how to properly build scalable infrastructure, as well as how to measure costs.

Code modularization (CI/CD code in the form of pipeline as code) is a big enabler of the efficient scaling of delivery pipelines. Finally, RBAC gives us the most granular control over executions and management to limit access to only those elements that need to be accessed by people.

In the next chapter, we will learn how to use the patterns we just described in this chapter to conduct a successful discovery phase and make architectural decisions to prepare for implementation.

6

Deployment Strategies for Structural Design Patterns for CI/CD

In this chapter on deployment strategies for structural design patterns within CI/CD pipelines, you will understand the intricate balance between robust design and agile deployment. There can be multiple deployment strategies based on the requirements and components of the CI/CD pipeline. However, in this chapter, we will focus on the following deployment strategies:

- Blue-green deployments
- Canary deployments
- A/B testing
- Feature toggles
- Dark launches

We will learn when to use each deployment pattern and when we can apply a combination of them for a scalable CI/CD pipeline.

We will learn how to apply controls in a CI/CD pipeline and also what data to integrate in the context of a CI/CD pipeline to get better visibility and make decisions around quality, usability, experimentation, and scale.

In addition to the aforementioned, you will also learn about setting up policy controls and the principle of least privilege to apply a security lens to the pipeline design.

Overall, we will cover the following topics in this chapter:

- Structural CI/CD design patterns – deployment strategies
- Types of deployment strategies

- Structural CI/CD design patterns and integration of data
- Setting boundaries through access management and policy enforcement

By the end of this chapter, you will understand the different deployment strategies and important telemetry signals that can be integrated with the different components in a CI/CD pipeline.

Let us begin to explore the deployment strategies in the coming section.

Structural CI/CD Patterns – deployment strategies

Structural design patterns in general provide the structure for how the components in a CI/CD pipeline interact. Applying them in an organization requires thoughtful design and architecture consideration in order for them to scale. It often requires choosing the right deployment strategies so that we can accommodate new components in the pipeline with minimal risks to improve reliability and quality, along with other aspects. Selecting the deployment strategies in a CI/CD pipeline also has implicit goals such as experimentation, scalability, reliability, and segregation of duties, which are associated with non-functional requirements in a CI/CD pipeline component.

Be it a **monolithic approach** or a **polylithic approach** for structural components, both involve deploying changes so that risk is minimal and consider the ability to scale processes and technology along with ease of adding or removing components.

Modern CI/CD pipelines are often evolving, and the choice of technology keeps changing over time, thus it is crucial to adopt deployment strategies that provide flexibility, ease of adoption, and correct telemetry or signals to make the right decision based on user behavior.

The design and importance of determining deployment strategies are crucial so that deployments are efficient and can scale. More importantly, deciding a deployment strategy or combination of deployment strategies is also crucial for overall pipeline design on how the application or code will be deployed to be presented in front of the users.

We will start with the following diagram depicting different deployment strategies in the CI/CD pipeline and how they are applied in the deployment stage:

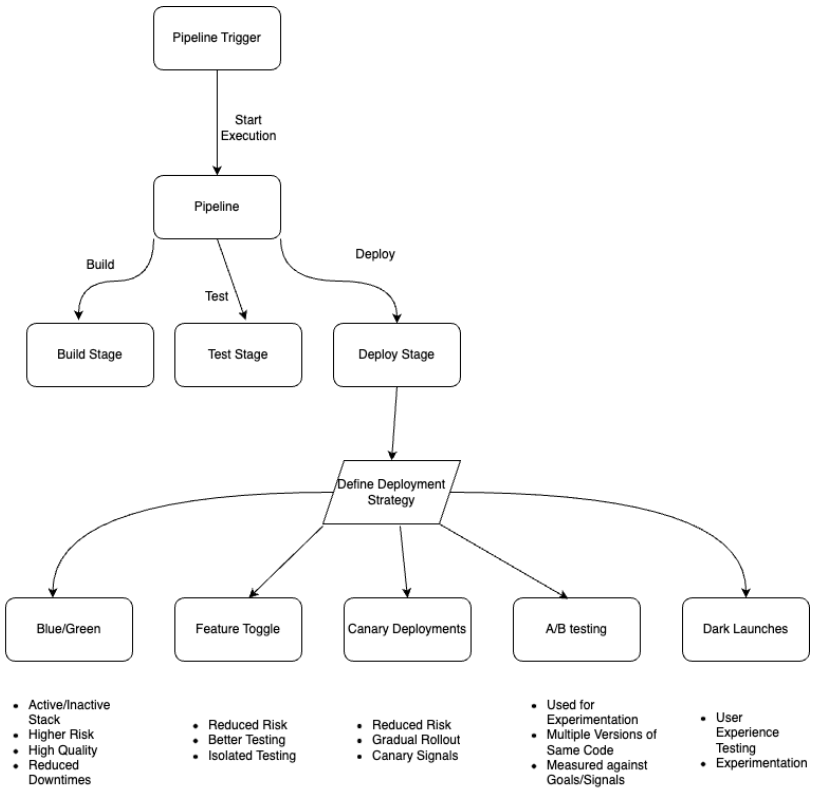

Figure 6.1 – The basics of deployment strategies in structural CI/CD

The preceding diagram shows an overview of different deployment strategies that are applied in the deployment stage along with their major attributes. For example, the **blue-green deployment** strategy requires two parallel production environments to be running, with one serving the active traffic while a new version of the app gets deployed to the inactive stack. Once the new version is tested in isolation and is ready, the traffic is flipped to the new version. However, this also comes with additional costs. Similarly, there are other deployment strategies that we will explore in subsequent sections of the chapter.

Types of deployment strategies

While designing the pipeline, once we decide on the components, stages, and their interactions, we must design the deployment strategy. Depending upon the primary goals and requirements of the system, we can select a particular deployment strategy or a combination of deployment strategies. Moreover, the choice of strategy is also dictated by other factors such as cost, user experiences, experimentation, go-to-market strategy, and the blast radius of the changes. Often, these requirements are subject to applying one or more deployment strategies in a pipeline. For example, sometimes, we want to introduce new features to actual production users, or we want to minimize downtime since

there is a cost and user experience attached to it. It all dictates how we deploy the changes and make them available to the users.

In the next few sections, we will explore a few of the deployment strategies that are applied to structural CI/CD pipelines. You read a highlighted overview of these strategies in *Chapter 1*, but here we will also explore some of their trade-offs and see scenarios to apply them. Let's dive right in.

Blue-green deployments

This deployment strategy is used when we want to minimize downtime and test a new version of an application separately but in an actual production environment. Not only does this provide flexibility in deploying a new version of an application to an inactive but actual product environment but it also reduces risk in case anything goes wrong while testing. Let's understand this through the following diagram:

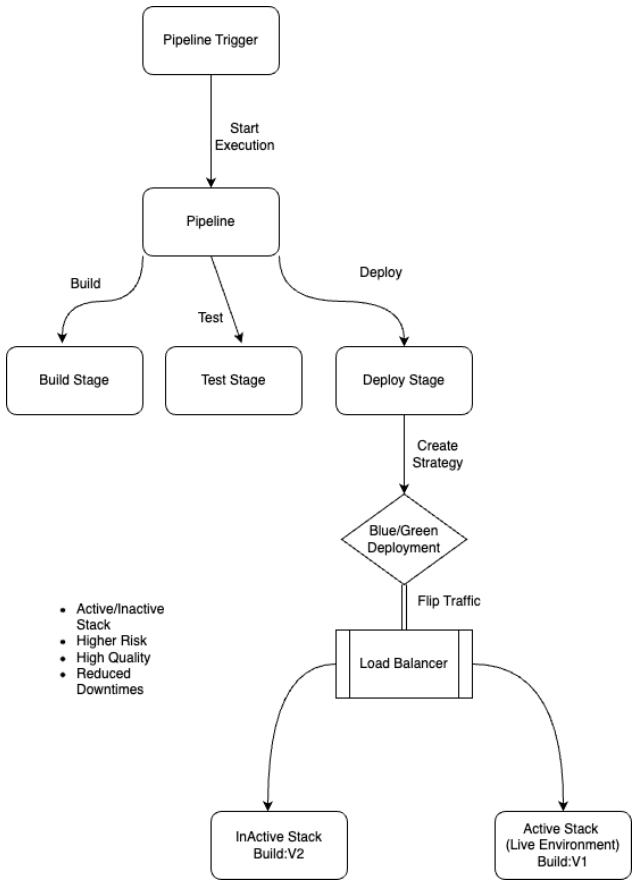

Figure 6.2 – Blue-green deployment strategy in structural CI/CD

In the preceding diagram, we can see that once the pipeline comes to the deploy stage, the new deployment goes to the inactive stack (`Build:V2`), while the current version (`Build:V1`) is still being served with the active stack via the live environment. Essentially, both active and inactive stacks are maintained as per the production environment configuration. However, only one serves the live traffic where users access the application. The other version or inactive stack is used to deploy the new application version, and that is where the isolated testing is performed without affecting the actual production traffic. Once the testing is finished and the new version is good, that is when traffic is flipped to the inactive stack and the new version starts serving live users.

Flipping the traffic between the active and inactive stacks often involves a load balancer and a **Domain Name System** (**DNS**), along with different routing algorithms. One such routing algorithm is the **weight-based routing** algorithm where *one* denotes the active stack and *zero* denotes the inactive stack, and a new deployment always goes to the stack where the weight is zero.

Blue-green deployments reduce downtime and risk but often come with a heavy cost, as maintaining more than one production stack can be cumbersome and tedious. In addition, they can also create a waste of resources at times as the inactive stack is only used for new version deployment and testing or in case anything goes wrong. So, what can we do to reduce the cost? That is where one of our next deployment strategies, known as **canary deployment**, comes into the picture.

Canary deployments

This deployment strategy is based on the **principle of canaries**, where a small set of users (canaries) are exposed to a new version of the software, and their feedback is collected.

The name *canary* comes from an interesting fact where canary birds were used in the coal mines to detect poisonous gases; as long as they kept singing, there was no poison, but if they stopped singing, then it signaled that there was some issue in the mine and it signaled miners to evacuate the mine. In similar terms, if there are any issues, canary deployments are good at detecting them early and we stop rolling down further or impacting users.

As part of this deployment strategy, a new version of the application is gradually rolled out to users in percentages, and at each progressive rollout, some tests or signals get executed to detect potential issues. These signals are sometimes also associated with functional issues or non-functional issues such as performance or bugs, which, when discovered, are fixed, and the rollout is again measured against a set of tests or signals.

Canaries are often deployed as part of the pipeline deployment stage. For this, usually, a load balancer component is used to control the traffic to the new version. In principle, some percentage of the traffic is rolled to the new version or functionality of the application while other users/traffic still stay on the old version. Canary deployments do not have any downtime associated since, if any problems are detected, then users are rolled back to the old version with the usual traffic flow. This provides a significant advantage over the blue-green deployment strategy where traffic needs to be flipped in an *all or none* fashion between stacks. Also, the cost of maintaining and implementing a canary

deployment is less since it is the same stack and there is no parallel production environment that needs to be maintained.

In the following diagram, we can see that the pipeline's *deploy* stage is used to implement the change using a canary deployment methodology. In the production stack, two instances represent the currently running version, V1, of the build and are being served with 40% of the traffic each, while the new build is represented as V2, which is only being served to the remaining 20% of users. A load balancer is used to control the traffic among different instances of the application.

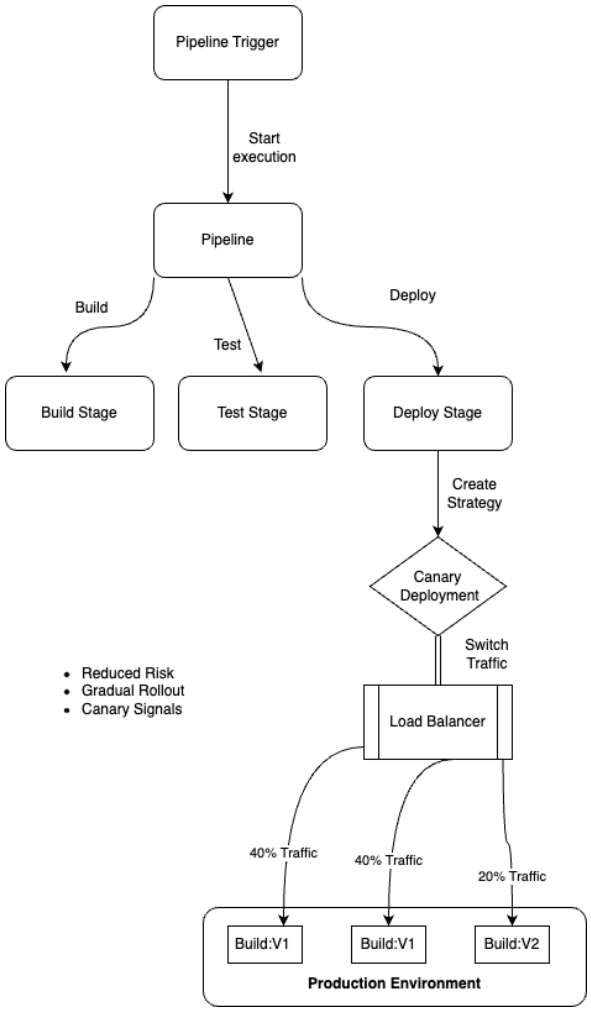

Figure 6.3 – Canary deployment strategy in structural CI/CD

Canary rollouts are implemented using some sort of routing algorithm that can control the traffic in percentages.

For example, A popular **container orchestrator** technology known as Kubernetes uses NGINX-based ingress controllers that use weight-based annotations to determine how much application traffic will go onto one version of the application. The ingress controller is the mechanism by which Kubernetes decides to route the traffic. The following is the sample syntax on how you can configure the weight-based routing to accommodate the canary rollouts:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: canary
  annotations:
    nginx.ingress.kubernetes.io/canary: \"true\"
    nginx.ingress.kubernetes.io/canary-weight: \"30\"
spec:
  ingressClassName: nginx
  rules:
  - host: example.com
    http:
      paths:
      - pathType: Prefix
        path: /
        backend:
          service:
            name: canaryService
            port:
              number: 80
```

In the preceding example, we can see that the ingress controller is turned on via the `nginx.ingress.kubernetes.io/canary: \"true\"` annotation. Then, the next line tells us to set the canary weight as 30% (i.e., 30% of the application traffic will be routed to a service named `canaryService`).

The advantages of canary deployment involve reduced risk (since only limited users are impacted), gradual rollout (as it gives stability and confidence in the new build), and early feedback (since the new version can be easily tested by actual users).

While a canary deployment is preferred for its advantages, there are cases where we want to experiment with the application version as early as possible. That is where feature toggles come in handy as a deployment strategy where users can experience a new version of the application by turning a feature on and off and reporting any issues. Let's explore how the **feature toggle deployment** strategy helps in implementing the CI/CD components and how to design that in a CI/CD pipeline.

Feature toggle deployments

The feature toggle deployment strategy helps by controlling the features of an application using control switches or feature flags.

In terms of structural components of CI/CD pipeline design, the feature toggle pattern is helpful in multiple scenarios. Examples are if you want to test a feature branch while code gets deployed from the main branch, or you want to test selective features within an application that are behind one or more control switches (feature flags) that users get to experiment with by turning these features on and off.

In the following diagram, we can see that the build version V2 is behind a feature flag and has additional features. These features can be accessed by users by turning that feature on or off in the application or by the deployment process.

These features can come from either a feature branch or the main branch in the source code. If the main branch is used for the build, the pipeline's build versions V1 and V2 will be the same. However, feature toggles are also used to test features where builds are deployed from different branches.

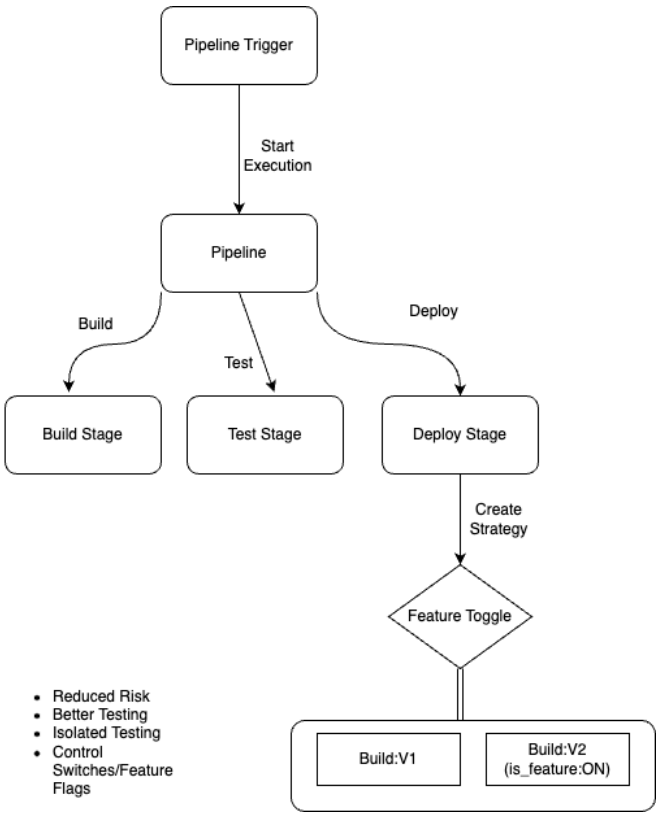

Figure 6.4 – The feature toggle deployment strategy in structural CI/CD

The preceding diagram represents the feature toggle deployment strategy where a control flag's value (`is_feature`) can determine whether a particular feature can be accessed in an application or not.

The feature toggle deployment strategy also reduces risk as features are available to only selective users who turn on the flag and it is also done with actual users.

While this strategy is useful for testing specific features, what happens if you get a requirement to test the same version of an application, and you want to test with actual users and get feedback to decide the next steps? Well, this is where the **A/B testing** deployment strategy can help.

Let's explore the A/B testing deployment strategy in structural CI/CD as part of the deployment phase along with advantages and scenarios when it is used.

A/B testing deployments

This deployment strategy is used when we deploy the same version of an application as part of the CI/CD pipeline and serve it to different sets of users to get their feedback. The A/B testing deployment strategy is often associated with predefined metrics or feedback norms that are used in terms of some statistical formulas to determine which version of the app users like better. Based on the requirement and implementation, A/B testing deployments are often used to experiment with new products by giving users a chance to validate the application and provide feedback. This feedback is then recorded and used in the decision-making process to move ahead with the final version of the application. At the core of this deployment strategy is experimentation and getting user feedback, along with collecting signals to determine the next steps. Let's understand this in a visual representation.

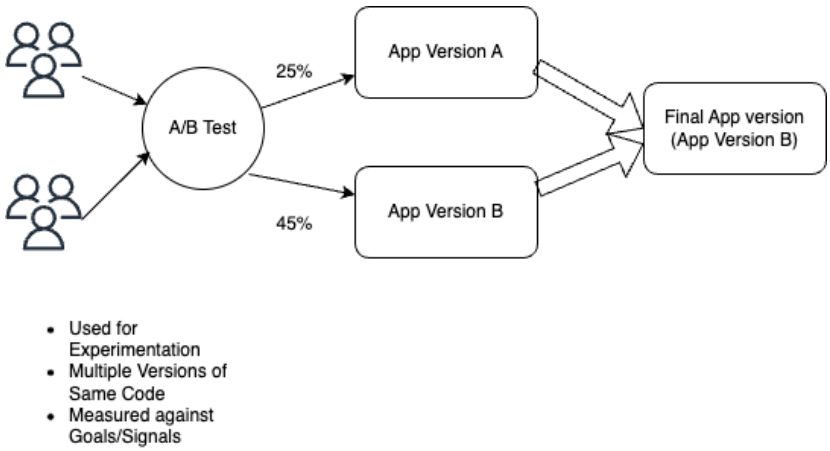

Figure 6.5 – A/B testing deployment strategy in structural CI/CD

In the preceding diagram, we can see that users are presented with two versions of the same app (versions A and B), and their feedback is recorded as signals on different parameters. When more users like version B, it becomes the final version of the application.

A/B tests are pretty common and do not require additional components such as load balancers in the stack. They are also pretty powerful to quickly make the decision on which version to use and also define the best user experiences. In terms of pipeline design, they also help in determining whether a particular implementation is being liked by more users.

So far, we have talked about different deployment strategies and when they are used. Let's talk about our final deployment strategy, which is the **dark launch deployment** strategy, and when it is used.

Dark launch deployments

This strategy is used when you want to test a particular version of your app or pipeline with the users randomly but only for a selected audience. While designing the pipeline of deploying an application, there is a need when we want to test a new feature or a bug fix to a randomized set of users who do not want to use the feature flag. In this case, the dark launch deployment strategy can help to test the feature and get feedback from them or identify bugs. One of the core differences between this strategy and the feature toggle strategy is there is no control flag used in this strategy; rather, users are selected arbitrarily and served with new features, and their feedback is recorded.

While, in simple cases, only one deployment strategy is implemented, there are scenarios where more than one deployment strategy is implemented in tandem – for example, if we want to test a new feature with reduced risk or blast radius or no downtime. In this case, one deployment strategy will not be sufficient, so that is where a combination of deployment strategies is used (in this case, a combination of blue-green and feature toggle can be used). In modern applications, it is more often the case since monorepo or polyrepo use cases both present a need to implement hybrid deployment strategies depending upon the requirement.

Alright, so far, we have seen different deployment strategies in the structural CI/CD context and how they can be implemented in the pipeline. As modern software pipelines are complex and distributed in nature, there is a need to emit the right set of data to understand the behavior of the components and to improve the experience and delivery constantly.

For example, have you come across these questions – *Why is my pipeline execution slow?*, *Why does it take a lot of time to deploy?*, *How much time does it take from code commit to deploy into production?*, and so on?

To answer these sorts of questions, we need to make sure that enough telemetry and signals are emitted through the components of the pipeline to determine the behavior and drive improvements.

In the next section, let's explore how data needs to be integrated with different structural components of a pipeline and the design goals we should prioritize when designing the pipeline.

Structural CI/CD design patterns and integration of data

While structural CI/CD design patterns focus on the structure of a pipeline and how different components related to the code, build, test, deploy, and post-deploy stages interact with each other, there is a need to measure, monitor, analyze, and improve the overall pipeline to continuously improve and provide value to the users. That is where it is important to design pipelines in a way where we have enough telemetry available.

Let's understand the importance of adding the signals and feedback loop in a delivery pipeline to improve further in a visual representation.

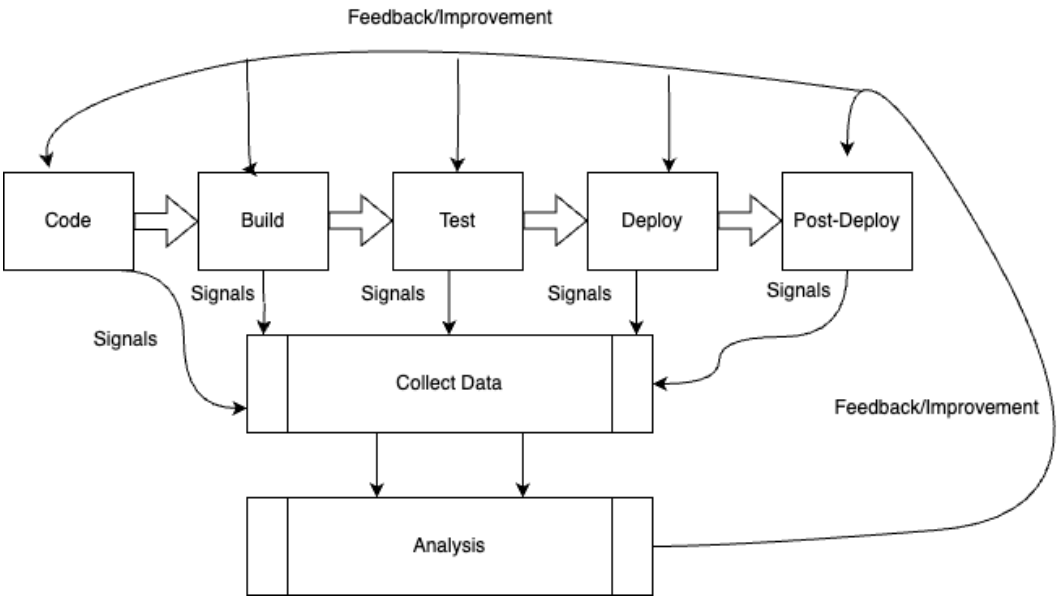

Figure 6.6 – How data is collected and used from each phase of the pipeline for improvement

In the preceding diagram, we can see each phase is emitting some signals or telemetry data that can be analyzed and further used to improve different components of the pipeline.

Let's go a bit deeper and analyze each phase of the diagram and see what data needs to be emitted and used for the analysis of bottlenecks, performance improvements, or other behavioral changes.

Code phase

During the **code phase**, teams contribute code and work on implementing the code. We need to measure practices related to code hygiene, development, and contributions. It also determines how the teams operate while developing a new feature, a new software, or maintaining an existing software.

As part of this phase, we collect data related to code, which needs to be analyzed further for different purposes. For example, we can begin with the following data points:

- Number of branches created/deleted at a repo or a set of repos
- Merge strategy (trunk based on Gitflow-based development or a specific methodology such as a feature branch)
- How many reviews are done per **pull request** (**PR**)
- Static analysis runs
- Code coverage and type of coverages (branch, PR, flow, etc.)
- Unit testing executed
- All changes following the PR review model
- How long it takes for a new developer to submit a PR
- Proper semantic version enabled

While the preceding questions are not comprehensive, these serve as a guiding principle for the data to be measured against the code phase of the CI/CD pipeline since this phase is an important structural component. The idea behind measuring this data is to look at the code practices and improve upon them. For example, a code with low coverage or unit tests can have a greater number of bugs and will gradually need to be worked upon or improved.

Build phase

The **build phase** is associated with the build component of the CI/CD pipeline. When the developer commits the code and it is merged based on the branching strategy, then unit tests and static analysis findings are remediated. At that time, the build gets created by the build phase and is ready to be deployed in different environments depending upon the strategy (for example, tests need to be executed in an ephemeral environment or a dedicated integration environment). This phase is related to the CI part of the overall pipeline.

But how do we determine the build hygiene to detect bottlenecks in the build process? That is where build phase-related signals and data need to be emitted from the pipeline. For example, during the build phase, we can measure the following:

- Time taken to build an artifact
- Time taken to download dependencies
- Number of compiled versus runtime dependencies
- Build provenance (what dependencies are included in the build and their source)
- Final artifact produced

- The type of licenses for the dependencies being used
- Runtime and compile time vulnerability information exist in the build and its dependencies
- Number of queued builds at the infrastructure
- **Software bill of materials** (**SBOM**) produced
- Artifact build version created

Again, the preceding list is not comprehensive and we can measure and determine a greater number of signals to detect the bottleneck, performance, or security issues associated with the build or package produced. It can further be utilized to improve the build phase of the pipeline.

Test phase

Similar to the code and build phase, during the **test phase**, we can have different signals related to the test executed. These are often measured in terms of tests executed and their results. For example, we can measure the following:

- Number of test cases executed
- Number of test suites executed
- Results published per release, per PR, or per change
- Number of tests failed

The idea behind the signals from this phase is to determine how the test strategy is defined against the pipeline and overall application delivery as poor testing can result in a lot of bugs, which impact the quality of the delivery. Again, this is just an example of what to measure as part of the test execution strategy and testing phase of a pipeline.

Deploy phase

The **deploy phase** is associated with the CD aspect of the overall pipeline. It provides feedback on deployment-related tasks. For example, if we want to answer questions such as how much time it takes to deploy a change to a particular environment (test, non-prod, production) and what signals we should have emitted and collected from our pipeline. Signals emitted from the deploy phase of the pipeline help in improving overall pipeline performance as well as deployment infrastructure performance. Also, it can help in determining the deployment strategy aspect of the code being deployed since we can get signals and make decisions on implementing a particular deployment strategy.

Some of the key signals we should be measuring during the deploy phase are as follows:

- Overall time for pipeline execution
- Time for deploy phase execution

- Time for individual tasks in deploy phase of the pipeline
- Time for new environment creation
- New environment created
- Service or application deployed and version

Deploy phase data is essential in pipeline implementation since it can provide actionable insights on what to improve upon when a pipeline is created or implemented especially related to environments or deployment-related tasks.

So far we have talked about the code, build, test, and deploy structural phases/components of a pipeline and some of the signals/data we need to analyze and collect to improve upon. Let's talk about the final phase of the pipeline, which is the **post-deploy phase**.

Post-deploy phase

This component of the pipeline is associated with post-deployment. Once an application is deployed to production, all signals need to be measured. In terms of the CI/CD pipeline, these signals are often associated with the **continuous operations** (**CO**) phase. Data collected during this phase is used to analyze user behavior, monitor any anomalies, detect events before they happen, or ensure an error does not occur again. During this phase, we can collect the following signals from the pipeline:

- Number of incidents
- Number of repetitive tasks
- Repeat incidents
- Actionable alerts created
- Logs published
- Errors handled

In addition to the preceding, other metrics can be collected or analyzed depending on the use cases and requirements.

While there can be multiple signals emitted from different structural components or phases of a pipeline, how do we know what to collect for how to interpret or process the signals? Open source standards and frameworks exist that can help in this regard. In the next two sections, we will talk about some of the frameworks/guiding principles/implementations that are open source and can be adapted easily in the pipeline design or different structural components.

CDEvents

CDEvents (`https://cdevents.dev/docs/`) provides a specification as well as implementation for different CI/CD providers on what data to integrate and collect from each phase of the CI/CD pipeline. The CDEvents specification consists of six main categories where data can be emitted and collected from a CI/CD pipeline. These categories correspond to different structural components of the pipeline but generalize on what structure of events as well as signals to collect from. These categories are as follows:

Category	Description
Core Events	Specific events that are fundamental to pipeline execution and orchestration
Source Code Control Events	Events related to the code phase
Continuous Integration Events	Events related to the build, packages, and artifacts
Testing Events	Events related to the test phase
Continuous Deployment Events	Events associated with CD
Continuous Operations Events	Events related to CO

Table 6.1 – Event types from the CDEvents specification

CDEvents also provides SDKs and integration to common build and deployment systems so a lot of these events/signals are collected by default. You can read more about CDEvents from the link.

Supply-chain Levels for Software Artifacts

Supply-chain Levels for Software Artifacts (**SLSA**) (`https://slsa.dev/`) is another open source standard that deals with the security aspect of the builds in the CI/CD pipeline. It deals with artifact provenance and SBOM generation of artifacts and can answer questions related to artifacts produced during the build phase. It can also be integrated directly into the pipeline and can generate signals.

Overall, both CDEvents and SLSA are geared toward interoperability of CI/CD pipelines so no matter what the underlying implementation of the pipeline is or the components involved, they provide a common way to tap signals or events across the whole pipeline so that it can be more robust, efficient, and performant. These projects can very well serve as a starting point for collecting data from different components of the CI/CD pipeline.

Now we have learned about deployment strategies and some key data signals we need to collect from the different phases of the pipeline, how do we deal with security or access control aspects in the CI/CD pipeline? In modern software delivery pipelines, there are multiple teams and personas involved. There is a need to implement proper access controls between different components of the pipeline

or among multiple pipelines. So how do we solve this challenge and how can we implement a robust and secure pipeline design? Let's explore that in the next section.

Setting boundaries through access management and policy enforcement

Control and access control are very important aspects. Without considering them, we cannot design or implement the pipeline at scale. With modern CI/CD, when there are multiple teams involved in the delivery of software, it is important to consider proper security aspects and provide proper access controls in the CI/CD pipeline structure. It often starts from the beginning – who are the users or personas involved, and how and what are they going to use? It is also needed for regulatory and audit compliance purposes and, by design, the pipeline, platform, and components need to be secured against different threats.

In a typical pipeline, there are multiple teams and roles involved along with various components. Thus, it is necessary to define a clear boundary for each team and component to determine their functions and roles. In the following section, we will explore one of the most important concepts, called **segregation of duties** (**SoD**). Along with that, we will also explore applying various role-based access controls and policies to determine the functions and output of different phases in a pipeline.

Let's dive right in.

SoD

This is a fundamental security control that strives to prevent fraud and errors by dividing tasks and responsibilities among different individuals or systems. By separating the key stages of a process, such as authorization, recording, and custody, the organization can prevent any single individual from having complete control over a transaction. This helps minimize the risk of intentional or unintentional errors, as well as unauthorized access or fraud. In CI/CD, the principle of SoD is crucial for maintaining security and integrity throughout the development pipeline.

For instance, in a financial services company, SoD can be implemented by separating the roles of authorizing transactions, recording transactions, and maintaining custody of assets. This ensures that no single individual has the power to execute all stages of a financial transaction, thereby reducing the risk of fraud or errors. Similarly, in the healthcare sector, duties are segregated between professionals who prescribe medications and those who dispense them, ensuring checks and balances in patient care. In the IT domain, system administrators, application developers, and security personnel all have distinct responsibilities to prevent conflicts of interest and unauthorized data manipulation. These real-life applications of SoD in CI/CD environments demonstrate its effectiveness in safeguarding processes and reinforcing accountability within organizations. Implementing such controls complies with regulatory requirements and builds a fundament of trust and reliability in operational procedures.

Let's take an example of a reference CI/CD platform and understand the different personas and their functions, through the following diagram:

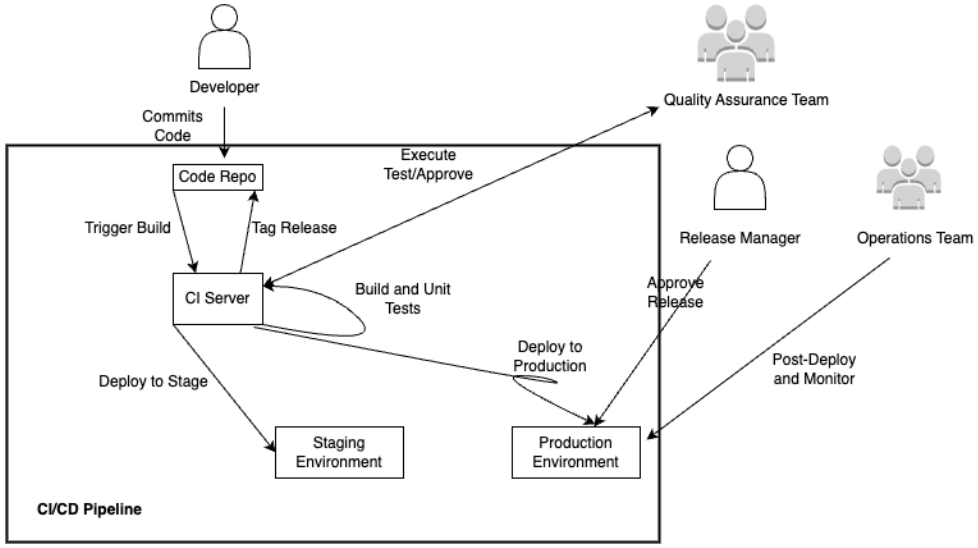

Figure 6.7 – SoD implemented in the CI/CD pipeline

The preceding diagram has the following roles and stages:

Roles	Developer, QA team, release manager, and operations team. These roles are responsible for different tasks to ensure that no single person has control over the entire CI/CD pipeline.
Stages	Code repository, CI server, staging environment, and production environment. These stages represent the different environments and systems involved in the CI/CD pipeline.

Table 6.2 – Roles and stages in a typical CI/CD pipeline

A typical execution flow happens as follows:

1. The developer commits code to the code repository.
2. The code repository triggers the build process on the CI server.
3. The CI server performs the build, and unit tests and deploys the build to the staging environment.
4. The QA team tests the build in the staging environment and approves the changes on the CI server.
5. The release manager tags the release and approves deployment to the production environment.
6. The CI server deploys the build to the production environment.
7. The operations team monitors and manages the production application.

Thus, we can clearly see the personas/roles involved and multiple stages in a CI/CD pipeline implementation. That is why it is important to enforce proper **role-based access control** (**RBAC**) to control the behavior and interaction of different components within the pipeline. Without going into too many details, let's talk about access management and policy enforcement in the context of the CI/CD platform or pipeline.

RBAC

RBAC enforces the permission model on the different users and attaches their access to perform different actions on resources/components. For example, in the previous diagram, we do not want the QA team to accidentally deploy software to production. Another scenario can be where untested code gets deployed directly to the production environment. That is where proper access control measures need to be implemented across the pipeline design. As it can ensure multiple roles and control mechanisms are applied to different flows or pipeline components.

As seen in the previous chapter, we can have multiple teams and members having access to different pipelines or components so it is important that they have proper access to execute or perform their tasks and we can prevent any accidental unwanted task execution.

In general, RBAC has the following guiding principle: *Who can perform what actions on which resource*? Here, the *who* refers to different teams and personas, *actions* refers to execute privileges, and *resource* refers to the components of the single pipeline or multiple pipelines in general.

Of course, implementing RBAC requires thorough design and is not easy to implement, but it is essential to establish these controls in a CI/CD pipeline context.

Approval flows

Approval flows are required in case an extra review or steps are needed that can or cannot be covered in the regular pipeline run. For example, approval flows can be used if a PR needs to be merged into a feature branch before it gets deployed in any environment or if an emergency change needs to be deployed into production. Usually, approval flows are not needed in normal pipeline execution, but they are needed in the tasks that are either not executed in the pipeline run or are executed outside of the pipeline context. However, they need to be thought through and incorporated as part of the pipeline design.

So far, we have learned how to apply access controls in a CI/CD pipeline, why they are needed, and how SoD needs to be a fundamental construct while designing a pipeline or CI/CD platform. Let's look at some open source frameworks that can help in implementing RBAC in a pipeline. We can use that as a guiding principle while implementing RBAC or policy controls in CI/CD.

Open Policy Agent

The goal of **Open Policy Agent** (**OPA**) (`https://www.openpolicyagent.org/`) is to provide a generic way to declare policies that are tied to actions. These policies can be attached to multiple types of resources (but not limited to), such as the following:

- Authorizing API endpoints
- Allowing or denying changes to resources in the pipeline
- Implementing/integrating custom authorization logic

OPA is used across a lot of open source projects, such as Kubernetes, which is a famous container orchestration platform, to allow or deny actions on Kubernetes resources. OPA also provides an engine that is open source and can be used to evaluate policies on different resources. OPA is context-aware and provides a declarative way to create policies on multiple cloud-native resources, such as Kubernetes, Terraform, and other technologies.

OPA policies can be hooked to any cycle of the CI/CD pipeline (code, build, test, or deploy) to have more control over the output. For example, while implementing a CI/CD pipeline's build component, we want to ensure the build is properly following a semantic versioning schema, which can be enforced via an OPA policy and can be evaluated at build steps for the pipeline for every run. Let's look at the general flow of OPA through the following visual representation:

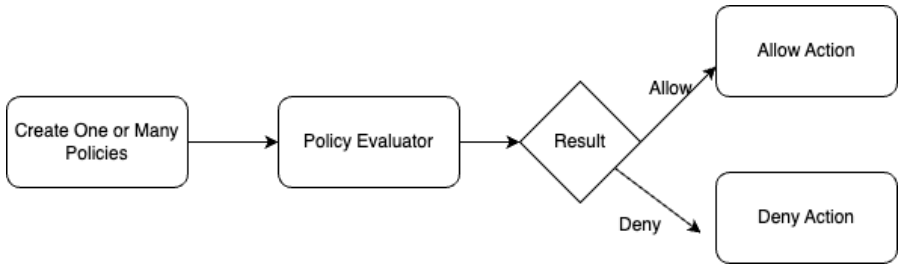

Figure 6.8 – A simple flow for OPA

In the preceding diagram, we can see that policies are defined on the left and are then evaluated by the open source engine. Based on the result, a particular action on a resource is allowed or denied, which, when applied to individual phases of the pipeline, can be responsible for determining the outcome of the actions defined in a phase. OPA can also be applied from an audit or compliance lens. For example, we want to make sure each component of the CI/CD pipeline is attributed with proper metadata schema (such as name, tag, path, etc.). This can be easily implemented and enforced in terms of OPA policy.

The right access controls are crucial in a CI/CD pipeline to prevent security risks and ensure compliance. Implementing these controls enhances developer productivity through automation and promotes efficient resource use. Additionally, segregating duties improves security and stimulates accountability

within the team. Strategically implementing deployment strategies enhances software quality and ensures adaptability to evolving demands.

In conclusion, the thoughtful implementation of multiple deployment strategies can significantly enhance quality aspects, providing a robust framework for the reliable and efficient release of software applications. By leveraging this knowledge, developers and operations teams can ensure that their deployment processes are not only effective but also adaptable to the continuously evolving demands of software development and delivery.

Summary

In this chapter, we explored multiple deployment strategies for structural design patterns along with when and how one or a combination of them can be used to deploy software with high confidence. We learned about the trade-offs of different deployment strategies and how they can be used in the real-world implementation of the actual pipeline. We explored important data signals we need to emit and collect from the different components of the pipeline and how they can be used to further detect issues and improve upon them. We also looked at some open source frameworks such as CDEvents and SLSA, which can give us a head start when integrating events in terms of the pipeline. Last but not least, we also learned about how to apply the principles of SoD and RBAC and some practical aspects of implementing roles and policies in a CI/CD pipeline. While this may give us a good foundation, we should also explore other aspects and data signals beyond the mention in this book to make the pipeline design and components of the pipeline more robust, secure, and performant.

Since modern software systems are complex and have multiple components, it is important to understand the interaction and behavior of each involved component. In the next chapter, we will dive a bit deeper into how behavioral design patterns help us while designing the pipeline to understand the behavior and interaction of different components in a CI/CD pipeline context.

Part 3: Behavioral and Domain-Driven Design Patterns for CI/CD

In this part, you will learn about managing behavior and interactions in a CI/CD pipeline and how design patterns can help in improving observability or reducing toil in software delivery. You will also learn about how **artificial intelligence** (**AI**)-driven monitoring solutions can predict failures in CI/CD in advance and thus have added benefits such as reduced downtime or blast radius reduction. You will also learn the importance of designing CI/CD pipelines in a regulated environment and how it can help with scalability and robustness while delivering software.

This part has the following chapters:

- *Chapter 7*, *Understanding Behavioral Design Patterns for CI/CD*
- *Chapter 8*, *Domain-Driven Design Patterns for Regulated Sectors*

7

Understanding Behavioral Design Patterns for CI/CD

Let's begin this chapter by taking a general look at behavioral CI/CD patterns and understanding what they entail. We were previously introduced to them in *Chapter 1*, so let's dig deeper now.

In this chapter, you will delve into the intricate world of **behavioral CI/CD design patterns**, gaining insights into the systematic approaches that integrate behavioral science with CI/CD practices. The overarching goal is to equip practitioners with the knowledge to design, implement, and evaluate systems that not only meet technical specifications but also align with human behaviors and cognitive processes. Understanding these patterns is crucial for creating more intuitive and user-friendly software development pipelines, which is of paramount importance in an era where technology is deeply intertwined with daily operations. This information is not only essential for software engineers and developers but also for project managers and UX designers, who aim to enhance the efficiency and effectiveness of their development cycles.

In this chapter, the following topics will be covered:

- An overview of behavioral CI/CD design patterns
- Behavioral CI/CD design pattern principles
- Key components – tools and processes for AI-based observability
- Challenges in implementing behavioral design patterns

An overview of behavioral CI/CD design patterns

Behavioral patterns help manage complex interactions and behaviors in software, making your code more modular, flexible, and maintainable. It's crucial to choose the right pattern, based on the specific problem and interactions within your system, to streamline and enhance your software design.

Behavioral CI/CD design patterns are essential for streamlining and optimizing the CI/CD processes in software development. They provide a framework for the effective collaboration of development and operations teams, ensuring that software can be developed, tested, and released more efficiently and reliably. Behavioral patterns deal with algorithms and how responsibilities are assigned between objects. They describe not only patterns of objects or classes but also the patterns of communication between them. These patterns represent complex control flows that can be difficult to follow during runtime.

By implementing behavioral design patterns, teams can automate repetitive tasks, reduce errors, and facilitate a culture of continuous improvement. For instance, the Chain of Responsibility pattern allows you to decouple the sender and receiver, enabling requests to be handled by multiple objects within a pipeline. Similarly, the Command pattern encapsulates requests as objects, allowing for flexible and reversible operations. These patterns not only improve the technical workflow but also enhance communication and collaboration among team members, leading to a more agile and responsive development cycle. Adopting these patterns can significantly contribute to a project's success by reducing time to market and improving product quality. The integration of behavioral design patterns into CI/CD practices is a strategic approach that aligns with modern DevOps principles, promoting a seamless and efficient software delivery pipeline.

There are several patterns to identify, but let's first view a diagram of all the patterns and their specific functions relating to CI/CD:

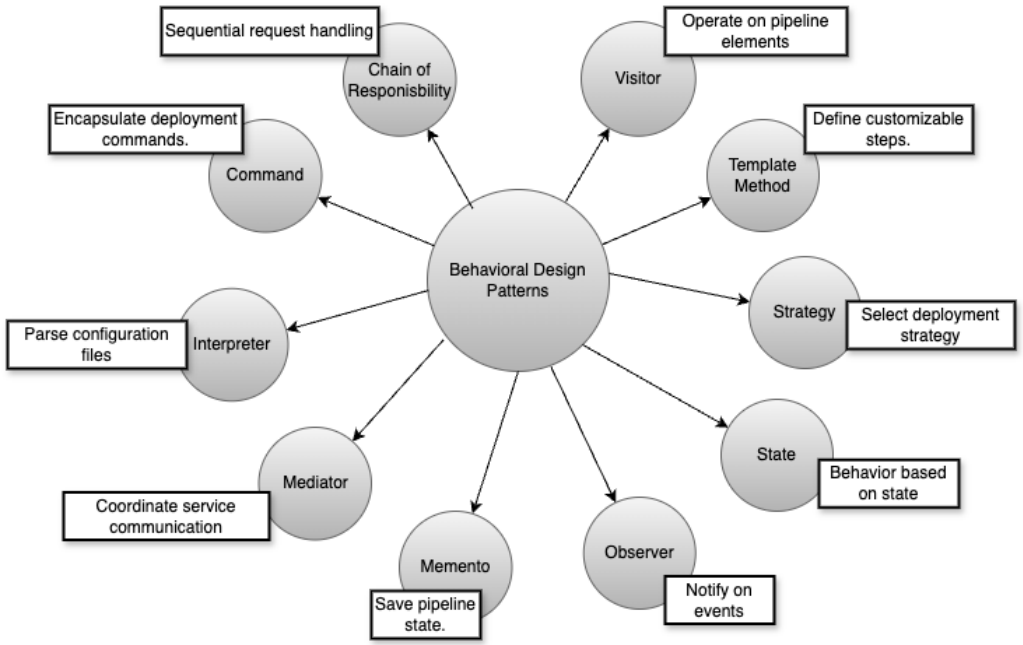

Figure 7.1 – A system diagram of the behavioral patterns in CI/CD

Let's break down the patterns visible in this diagram:

- **Chain of Responsibility design pattern**: This pattern can be used to handle pipeline stages (such as build, test, and deploy). Each stage is a handler in the chain. If a stage succeeds, it passes the request to the next stage. If it fails, it handles the failure. This allows stages to operate independently and handle failures dynamically, and the set of stages can be specified dynamically. This allows multiple objects to handle a request without the sender being coupled to a specific receiver. The request is passed along a chain of receiving objects until one of them handles it.

  Although this pattern (and the others mentioned here) can be applied in general software development, we will focus on CI/CD.

- **Command design pattern**: This encapsulates a request in an object. It parameterizes clients with different requests, queues, or logs requests and supports undoable operations.

  It encapsulates each stage (such as build, test, deploy) as a command. This allows stages to operate independently and handle failures dynamically.

- **Interpreter design pattern**: This pattern describes a grammar representation and an interpreter to interpret language sentences.

  It can be utilized in CI/CD to interpret and execute commands or scripts during the build, test, and deployment process. Each operation in the pipeline is denoted as an *expression* in a specific language, such as a shell script. An interpreter, which may be a component of the CI/CD tool, comprehends this language and carries out these expressions. When the pipeline runs, the interpreter interprets and executes each expression in the specified sequence. This approach enables flexibility and customization in defining the CI/CD process.

- **Mediator design pattern**: The mediator method, also known as the mediator design pattern, involves an object that encapsulates the interaction between a set of objects. This approach promotes loose coupling by preventing objects from directly referring to each other and allowing them to independently change their interactions.

  It can be used to manage the interactions between different stages of a pipeline, such as build, test, and deploy. In this pattern, each stage is represented as a component that communicates with a mediator object, rather than directly with another component. This approach reduces dependencies and promotes loose coupling.

- **Memento design pattern**: The Memento design pattern involves using a token. It allows you to capture and externalize an object's internal state without violating encapsulation, enabling the object to be restored to this state later.

Let's take a look at the Memento design pattern in a diagram:

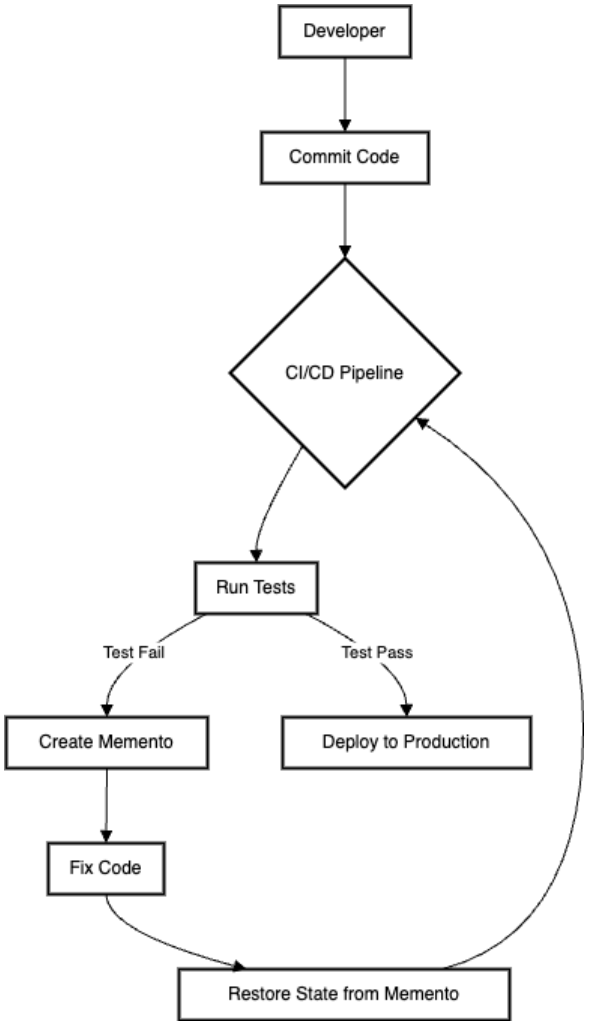

Figure 7.2 – The Memento pattern in a CI/CD pipeline

This diagram illustrates how the Memento design pattern can be used in a CI/CD pipeline to restore a system to a known good state when tests fail, allowing a developer to fix the issues and rerun the tests. It can aid in rollbacks to a prior state if a new deployment causes issues, create system snapshots before testing, and assist in blue-green deployments by capturing the state of the live environment before a switch.

- **Observer design pattern**: The Observer method, also known as the Observer design pattern, establishes a one-to-many object dependency. This means that when one object changes its

state, all objects get notified and updated automatically. This pattern is also referred to as *dependents* and *publish-subscribe*.

In a CI/CD pipeline, the Observer design pattern can be used for real-time monitoring and notifications. When code is pushed, stages such as build and test get notified and start their tasks. If a build fails, subsequent stages are notified to stop. This pattern also allows stages to operate independently, enhancing modularity.

- **State design pattern**: Managing pipeline stages as *states* will automate transitions between stages and maintain pipeline consistency. For instance, if a build fails, a pipeline can transition to a `BuildFailed` state and act accordingly. This pattern helps to handle errors gracefully and ensure that the pipeline remains in a consistent state. Note that its use is conceptual and depends on the specific technologies and tools used in the pipeline.

  The State design pattern can manage stages such as *idle*, *building*, *testing*, *deploying*, *success*, or *failure*. Each state has specific behaviors. For instance, if a build fails, the pipeline enters the Failure state, sends a notification, and stops further execution. Once fixed, it returns to the Idle state, ready for the next code commit. This pattern makes a system flexible and easier to manage.

- **Strategy design pattern**: Allows the algorithm of a system to be independently changed at runtime.

  In a CI/CD pipeline, different testing strategies (such as unit testing, integration testing, and performance testing) can be encapsulated as strategy objects. The pipeline can switch between these strategies, based on factors such as code complexity, system load, and the release cycle.

  Maybe the best way to illustrate the Strategy method pattern is in a code snippet. Here's an example of how this pattern is implemented in a well-known CI/CD platform, GitLab:

```
stages:
  - test

variables:
  UNIT_TEST_PROJECT: my_project_unit_tests
  INTEGRATION_TEST_PROJECT: my_project_integration_tests
  PERFORMANCE_TEST_PROJECT: my_project_performance_tests

unit_tests:
  stage: test
  script:
    - echo "Running unit tests..."
    - gitlab-runner exec docker test --env CI_PROJECT_DIR=$UNIT_
TEST_PROJECT

integration_tests:
  stage: test
  script:
    - echo "Running integration tests..."
```

```
    - gitlab-runner exec docker test --env CI_PROJECT_
DIR=$INTEGRATION_TEST_PROJECT

performance_tests:
  stage: test
  script:
    - echo "Running performance tests..."
    - gitlab-runner exec docker test --env CI_PROJECT_
DIR=$PERFORMANCE_TEST_PROJECT
```

In this example, `unit_tests`, `integration_tests`, and `performance_tests` are different jobs in the pipeline that correspond to different testing strategies. Each job uses a GitLab Runner to execute the tests in a Docker environment. The specific testing strategy used can be switched by changing the `CI_PROJECT_DIR` environment variable to point to the project directory containing the tests you want to run.

- **Template design pattern**: The Template method pattern defines the basic structure of an algorithm within a method, delegating some steps to subclasses.

  A CI/CD pipeline can be seen as a template method, with steps such as build, test, and deploy. The specifics of each step can be defined in subclasses, allowing different projects or environments to have customized pipelines while sharing the same overall process.

- **Visitor design pattern**: This enables you to add new virtual functions to a group of classes without altering the classes themselves.

  In a CI/CD pipeline, different types of code changes (such as feature addition, bug fix, and refactoring) can accept a *build* visitor that applies different build and test strategies, based on the type of change.

Now that you've obtained a concise view of these patterns, let's take a look at the principles of integrating observability in the next section.

Behavioral CI/CD design pattern principles

The integration of observability into CI/CD pipelines is an approach that redefines the traditional boundaries of development and operations. By incorporating metrics, logs, and traces, teams gain a panoramic view of their applications, allowing for proactive issue resolution and performance optimization. This strategic embedding of observability tools not only accelerates deployment cycles but also strengthens system resilience, ensuring that services remain robust and responsive under varying loads.

The following practices help you to adhere to behavioral CI/CD design patterns:

- **Site Reliability Engineering** (**SRE**) practices make things better by creating a culture where everyone is responsible and always looking for ways to improve. SRE principles guide teams to balance the competing demands of releasing new features and maintaining system stability.

- The adoption of standardized protocols and open specifications, such as the **CDEvents** specification, is pivotal in achieving a harmonious orchestration between disparate CI/CD tools. This standardization ensures that communication flows seamlessly, providing a unified and transparent view of the pipeline's health. This specification, an incubated project at the **Continuous Delivery Foundation**, refers to the ability of systems to work together, allowing services to create and use events independently of one another. This concept expands on the Continuous Delivery Foundation's efforts to establish the best practices for CD, defining a common language for CI/CD ecosystem events, which permits the decoupling of pipeline descriptions from physical implementations. This decoupling makes CI/CD pipelines more resilient to failures and easier to scale, especially in complex microservice architectures with numerous independent pipelines. By using CDEvents, organizations can connect workflows from different systems, greatly accelerating the migration to a fully automated CD process. You can learn more about CDEvents at `https://cdevents.dev/`.
- Several other specifications and best practices can also be crucial for optimizing. Apart from the CDEvents specification, other relevant standards include the **Open Container Initiative** (**OCI**), which defines industry standards around container formats and runtimes, ensuring consistency and interoperability in containerized environments.
- The **pipeline as code** standard is another key specification that enables the definition of the CI/CD pipeline configuration in code form, which can be versioned and managed just like any other software system. This approach enhances collaboration, review, and audit processes, making a pipeline more robust and easier to maintain.
- Furthermore, the **Tekton** project provides a set of shared, open source components for building CI/CD systems, allowing teams to build powerful pipelines that are not tied to a single CI/CD vendor.
- The **Spinnaker** project is also noteworthy; it's an open source, multi-cloud CD platform that supports complex deployment strategies, such as canary releases and blue-green deployments.
- Additionally, the **GitOps** methodology, which emphasizes the use of Git as the single source of truth for declarative infrastructure and applications, is becoming increasingly popular in managing CI/CD pipelines and cloud-native applications.

These practices are very helpful, but they can be different for your organization, depending on specific requirements. Nevertheless, it's strongly recommended to embed these practices in your CI/CD pipelines.

Metrics and monitoring play a vital role in CI/CD pipelines, and specifications such as **Prometheus** for monitoring and **Grafana** for visualization are widely adopted. These tools help in collecting and visualizing metrics such as build times, test coverage, and deployment success rates, which are essential for assessing the performance and effectiveness of CI/CD pipelines. Moreover, security is a paramount concern in CI/CD, and specifications such as the **Open Web Application Security Project** (**OWASP**) provide guidelines and tools to secure software through the development process.

In summary, the CI/CD domain is rich in specifications and best practices that aim to streamline the development process, enhance collaboration, ensure security, and maintain high standards of quality. Adopting these standards can greatly enhance the efficiency and dependability of software delivery, allowing organizations to quickly respond to customer needs and maintain a competitive advantage.

Furthermore, the utilization of established observability platforms such as OpenSearch, Prometheus, and Jaeger extends the scope of monitoring beyond production. It brings a view of observability to the release process itself, offering insights into the pipeline's efficiency and revealing potential areas for enhancement. This holistic perspective is crucial for identifying bottlenecks and implementing targeted improvements, ultimately leading to a more streamlined and reliable delivery process.

In the following figure, you can see these tools as they relate to CI/CD, each with its own focus point:

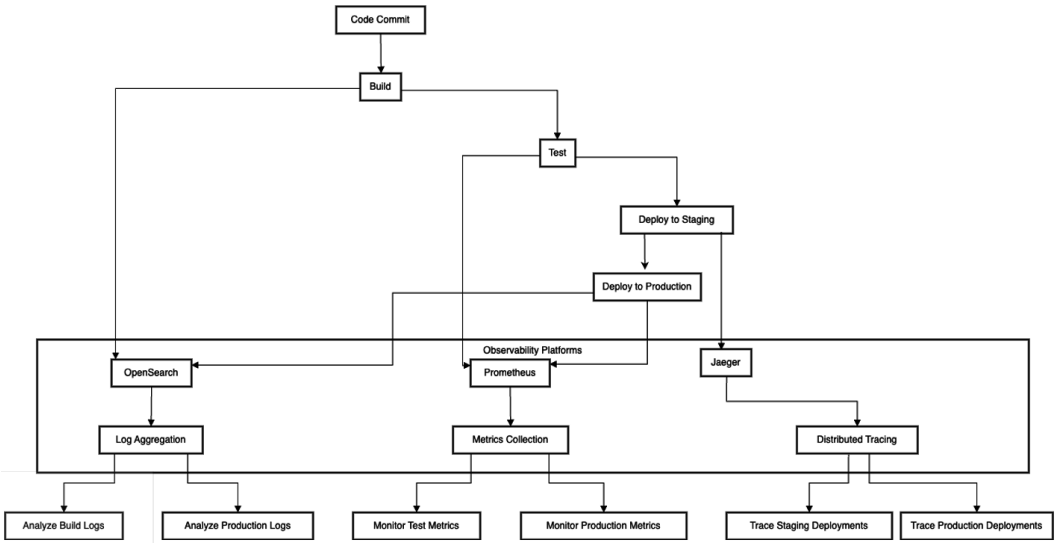

Figure 7.3 – Observability in CI/CD

The preceding figure shows that, by integrating these observability tools into the CI/CD pipeline, organizations can gain deep insights into each stage of the delivery process, enabling them to identify and address issues proactively. This leads to more efficient and reliable software delivery.

In essence, the holistic integration of observability into CI/CD pipelines is a cornerstone of modern software development. It empowers organizations to move faster, with greater confidence, and with a deeper understanding of their systems. As this practice becomes more prevalent, it paves the way for a new era of software delivery, characterized by speed, stability, and scalability.

Next, we are going to have a look at the component part of observability.

Key components – tools and processes for AI-based observability

CI/CD and AI-based observability are components that drive efficiency and innovation. AI-based observability injects intelligence into the monitoring process, providing deep insights into a system's performance and behavior. Machine learning algorithms are utilized to analyze data, predict issues, and automate responses, elevating observability beyond traditional logging and monitoring systems. Let's have a look at the tools and processes that can help. Furthermore, we'll also dive more into what AI-based observability can do in terms of CI/CD.

Tools and processes

CI/CD is powered by key tools such as source code repositories, build tools, and automated testing frameworks, which work together to ensure that code changes are stored, integrated, and validated efficiently. Processes such as version control, code review, and automated deployments are vital for maintaining the integrity and velocity of software releases. We can see these tools represented in the phases of CI/CD, each explained in their specific area:

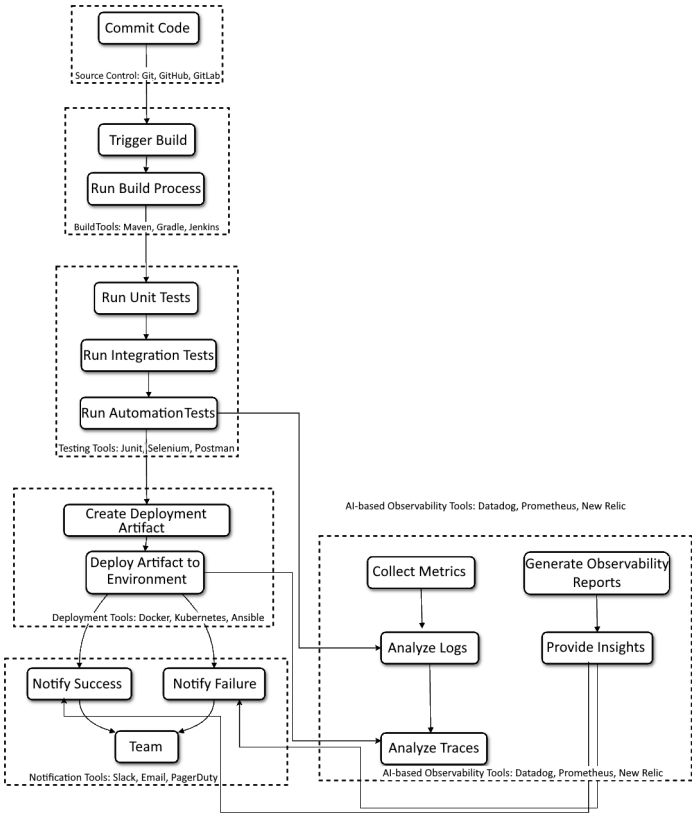

Figure 7.4 – CI/CD tools and phases

The following tools and phases in this diagram are as follows:

- **Source control (Git, GitHub, and GitLab)**:
  - The source code repository stores the code, and code changes are committed
  - The build process is triggered after a commit is made
- **Build tools (Maven, Gradle, and Jenkins)**:
  - Tools such as Jenkins, Gradle, and Maven are used to trigger and run the build process
  - After building, it proceeds to the testing phase
- **Testing tools (JUnit, Selenium, and Postman)**:
  - The pipeline runs unit tests, integration tests, and automated tests
  - Each testing phase generates test results for validation
- **Deployment tools (Docker, Kubernetes, and Ansible)**:
  - After the successful completion of testing, a deployment artifact (e.g., Docker image) is created
  - This artifact is deployed to the target environment (e.g., a Kubernetes cluster)
- **Notification tools (Slack, Email, and PagerDuty)**: Success and failure notifications are sent to the development team based on the result of the pipeline.
- **AI-based observability tools (Datadog, Prometheus, and New Relic)**:
  - **Metrics**: Observability tools gather real-time metrics related to a system's performance (e.g., CPU usage and memory utilization).
  - **Logs**: Logs are analyzed to track application events, errors, and interactions.
  - **Traces**: Traces capture and track requests as they move through distributed systems, providing visibility into how each service or component interacts.
  - **Observability reports**: Observability tools generate detailed reports based on metrics, logs, and traces to provide insights into an application's state and behavior.
  - **Integration**: These insights are sent to notification tools, allowing teams to be alerted based on the dynamic behavior of a system during the CI/CD process.

Conversely, AI-based observability tools focus on the dynamic aspects of software systems, tracking and analyzing metrics, logs, and traces to provide a comprehensive view of an application's state and environment. This integration of AI-based observability into the CI/CD process is a significant step toward enhancing the efficiency and reliability of software development. But first, let's take a look at processes within behavioral CI/CD.

Processes in behavioral CI/CD

Behavioral CI/CD design patterns can help to emphasize the integration of **Behavior-Driven Development** (**BDD**) practices (we will look at BDD practices shortly) within a CI/CD pipeline. The key components of a behavioral CI/CD process include version control, build automation, testing automation, and deployment automation. Systems such as Git manage code changes and track the history of these changes, ensuring that all team members work with the most up-to-date code. Build automation tools such as Jenkins or Travis CI simplify compiling code into executable artifacts whenever new changes are pushed to the repository.

Testing automation is a critical aspect of behavioral CI/CD, as it involves not only unit and integration tests but also behavior-driven tests. These tests, often written and executed using tools such as Cucumber or SpecFlow, ensure that software behaves as expected from a user's perspective. This emphasis on user behavior enhances the reliability of the software. The behavior-driven tests are defined by a clear, domain-specific language that specifies the desired behavior of the software. Deployment automation then ensures that the code is reliably deployed to various environments, from testing to staging, and finally, to production, with minimal human intervention.

The processes in behavioral CI/CD are iterative and agile, allowing frequent code commits, immediate feedback from automated tests, and rapid deployment cycles. This leads to a more efficient and reliable software delivery process, where the software is not only functionally correct but also meets the behavioral expectations set by the stakeholders. By integrating BDD practices into a CI/CD pipeline, teams can create a shared understanding of the desired software behavior, reduce misunderstandings, and ensure that the delivered software provides value to users. Continuous feedback and collaboration are also integral to this process, enabling teams to quickly adapt to changes and continuously improve the software product.

BDD emphasizes collaboration between developers, QA, and non-technical or business participants in a project. It is important to improve communication and cooperation among these stakeholders to ensure that the developed software meets business needs. Here are some critical BDD practices:

- **Collaborative specification workshops**: These include **three amigos sessions**, a practice that involves collaboration between a business representative, a developer, and a tester to discuss and agree on acceptance criteria for new features. **User story mapping** is also used to break down a user's journey into smaller parts for a better understanding of user needs and their context of product usage.
- **Writing user stories with Gherkin syntax**: The Gherkin language is a domain-specific language for defining tests in the Cucumber format.

  BDD uses feature files written in the Gherkin language. These files describe an application's behavior in terms of features and scenarios, using a *given-when-then* format.

- **Automating scenarios**: The scenarios described in Gherkin syntax are automated using BDD tools such as Cucumber, SpecFlow, or Behave. BDD extends **Test-Driven Development** (**TDD**) by focusing on an application's behavior from a user's perspective.
- **Living documentation**: BDD scenarios serve as living documentation that is always up-to-date and reflects the current system state. These scenarios can be read and understood by non-technical stakeholders, uniting technical and non-technical team members.
- Automated BDD tests are integrated into CI/CD pipelines to ensure that new changes do not break existing functionality and meet the specified behavior. Regular execution of BDD tests provides quick feedback to developers.
- **BDD tools**: Tools such as Cucumber, SpecFlow, and Behave are widely used for writing and executing BDD test cases.

Look at the following figure where BDD is incorporated into a test flow:

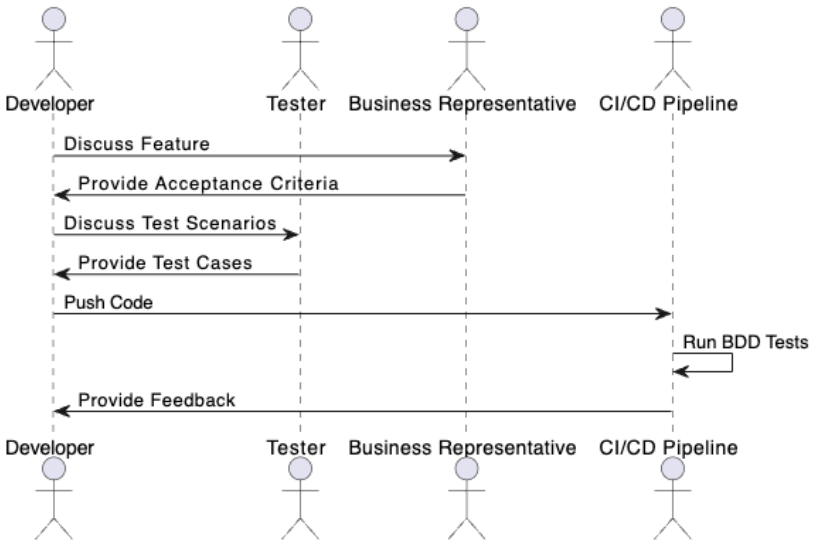

Figure 7.5 – BDD integrated into CI/CD

The preceding diagram shows the following:

- **The developer and the business representative**: They discuss the new feature that needs to be implemented. This step ensures that the developer understands the feature requirements and business needs.
- **The business representative provides acceptance criteria to the developer and tester)**: The business representative provides clear acceptance criteria for the feature. These criteria define what needs to be achieved for the feature to be considered complete and acceptable.

- **The developer and tester discuss test scenarios**: The developer and tester discuss various test scenarios based on the acceptance criteria. This collaboration ensures that both development and testing efforts are aligned and that the tests cover all critical aspects of the feature.
- **The tester provides test cases**: The tester creates and provides detailed test cases derived from the discussed scenarios and acceptance criteria. These test cases guide the testing process and ensure that the feature meets the required standards.
- **The developer pushes the code for the CI/CD pipeline**: After development, the developer pushes the code to the **version control system** (**VCS**), which triggers the CI/CD pipeline. This step integrates the new code into the larger code base and initiates automated processes.
- **The CI/CD pipeline runs the BDD tests**: The CI/CD pipeline automatically runs the BDD tests. These tests validate the new code against the defined acceptance criteria and test cases, ensuring that the feature behaves as expected.
- **The CI/CD pipeline provides feedback to the developer and tester**: The CI/CD pipeline provides feedback on the test results. If the tests pass, the feature is confirmed to meet the acceptance criteria. If any tests fail, the feedback will help identify issues that need to be addressed.

The benefits of BDD are not just theoretical; they also have a significant impact on the development process. BDD improves communication, ensures better quality, increases collaboration, and provides living documentation. These benefits can significantly enhance the efficiency and effectiveness of your agile software development projects, instilling a sense of optimism and motivation for its implementation.

The challenges of BDD

Although BDD offers numerous advantages, it's evident that you need to identify the potential challenges associated with its implementation. Some of these challenges include the initial learning curve as teams adapt to the BDD mindset and practices, difficulties in effectively maintaining and updating test scenarios over time, as well as the additional time and effort required to write and maintain scenarios. By proactively recognizing and understanding these challenges, teams can take the necessary steps to mitigate them and ensure BDD's smooth and successful integration into their projects.

By following these BDD practices, teams can develop software that better aligns with user needs, reduces misunderstandings, and enhances quality.

AI-based observability

Incorporating AI-based observability into CI/CD pipelines presents a transformative opportunity for DevOps teams. It allows proactive problem-solving, where potential bottlenecks can be identified and rectified even before they impact the user experience. This proactive stance is crucial in today's fast-paced, service-oriented tech landscape, where downtime or performance degradation can have significant repercussions. Moreover, AI-driven insights facilitate a more nuanced understanding of complex systems, enabling teams to optimize performance and resource utilization effectively.

The fusion of CI/CD with AI-based observability represents the confluence of automation, intelligence, and continuous improvement. It empowers teams to deliver high-quality software at an accelerated pace while maintaining robustness and stability.

AI-based observability can enhance security in CI/CD pipelines by offering advanced monitoring capabilities that surpass traditional methods. Through the use of machine learning and data analytics, AI-based observability tools can continuously scan and analyze large volumes of data produced during the CI/CD process. This enables the early identification of anomalies and potential security threats that may otherwise be overlooked.

Moreover, AI-driven systems can learn from past incidents to predict and prevent future security breaches. They can identify patterns indicative of security vulnerabilities, such as unusual code changes or irregular access patterns, and alert the relevant teams to investigate further before they escalate into more significant problems.

In addition to threat detection, AI-based observability can enhance security by automating the response to detected threats. For instance, if a potential security flaw is identified, a system can automatically initiate protocols to isolate the affected area, preventing the spread of any malicious activity. This level of automation enables quicker response time but also reduces human error in the management of security incidents.

Furthermore, AI-based observability can assist in compliance monitoring by ensuring that all parts of the CI/CD pipeline adhere to the required security standards and regulations. It can automatically verify that security controls are in place and functioning correctly and generate reports for audit purposes, thus simplifying the compliance process.

Overall, the integration of AI-based observability into CI/CD pipelines represents a forward-thinking approach to software development security. It provides a more dynamic, intelligent, and automated way to safeguard the integrity of the development process, ultimately leading to the creation of more secure software products. The continuous learning and adaptation offered by AI-based tools mean that security measures will evolve alongside emerging threats, maintaining a robust defense in an ever-changing digital landscape.

As the complexity of software systems grows, the role of these key components becomes increasingly critical, underscoring the need for sophisticated tools and processes that can adapt and evolve with the technological landscape. The future of software development hinges on the ability to seamlessly integrate these elements, fostering an ecosystem where innovation, efficiency, and reliability coalesce to drive forward the digital revolution.

Challenges in implementing behavioral design patterns

Implementing CI/CD and behavioral design patterns can be a complicated process, full of challenges that need to be carefully navigated. The first step is often *establishing a solid foundation* by choosing the right CI/CD platform, such as GitHub Actions, GitLab CI/CD, CircleCI, or Jenkins. This choice is critical, as it affects how well the CI/CD pipeline integrates with existing workflows and tools. Once a platform is selected, *setting up environments* to catch errors early in the development cycle is

essential. This includes configuring automated testing to ensure that new code does not negatively impact existing functionalities.

Linting and code formatting are also important to manage a uniform code base, making it easier for teams to collaborate. Initial tests for critical models should be created to verify that they function as expected. Additionally, *leveraging external open source tools* can provide extensive testing capabilities to prevent downstream breakages.

Conversely, behavioral design patterns require a different approach. It involves understanding how behavior patterns are generated and regulated within the context of software development and use. Designers must avoid making incorrect assumptions about user motivation and behavior patterns. Instead, they should focus on creating tailored, mechanism-based interventions that are engaging and sustainable in the context of a user's environment.

The implementation of these patterns should be evidence-based, with detailed analyses of mechanisms and change processes. This ensures that the interventions are not only theoretically sound but also practical and effective in real-world scenarios. Before scaling up, it's crucial to demonstrate success in less expensive efficacy evaluations, which provide the evidence base for larger-scale effectiveness trials.

In summary, the initial steps of implementing CI/CD and behavioral design patterns involve a strategic selection of tools and platforms, setting up robust testing environments, and a deep understanding of user behavior. These steps are foundational to building a system that is reliable, efficient, and user-centric.

Implementing design patterns in CI/CD

Behavioral design patterns, such as the Observer, Strategy, and Command patterns, are crucial for creating flexible and maintainable systems. These patterns facilitate better communication between objects and the delegation of responsibilities, which is particularly beneficial in a CI/CD context where multiple processes need to interact smoothly.

The first steps toward implementing these patterns in CI/CD involve understanding the specific needs of the development and operations teams. *Continuous Integration* focuses on automating the integration of code changes, ensuring that new code does not disrupt the existing system. This requires a robust setup that can handle automated testing and build processes, which are essential for identifying issues early on. Conversely, *Continuous Delivery* emphasizes the automated deployment of code to production environments, necessitating rigorous testing to ensure that each release is reliable and stable.

One of the primary challenges in implementing behavioral design patterns in CI/CD is ensuring that the patterns are adapted to the specific context of a pipeline. This means that the patterns must be designed to handle the dynamic nature of CI/CD workflows, where code is continuously integrated and deployed. For example, the Observer pattern can be used to monitor changes in the code repository and trigger builds and tests accordingly. The Strategy pattern can allow for different deployment strategies to be employed based on the target environment. The Command pattern can encapsulate various deployment commands that can be executed at different stages of the pipeline.

Another challenge is the integration of these patterns with existing tools and technologies. Many CI/CD pipelines are built using a combination of open source and proprietary tools, and behavioral design patterns must be implemented in a way that enables them to interact with these tools seamlessly. This often requires a deep understanding of the tools' APIs and the ability to extend their functionality through plugins or scripts.

Furthermore, the implementation of behavioral design patterns in CI/CD must consider the scalability and performance of a pipeline. As the number of deployments grows, the patterns must be able to handle an increasing load without compromising the speed or reliability of the pipeline. This may involve optimizing the patterns for parallel execution or ensuring that they can handle asynchronous operations effectively.

To address these challenges, it's essential to start with a clear vision and goals for the CI/CD implementation. Defining the scope and objectives of a pipeline can help in selecting the appropriate design patterns and configuring them to meet the specific requirements of a project. Additionally, establishing a robust testing strategy is crucial for ensuring that the implementation of design patterns does not introduce new issues into the pipeline.

Now, after challenges are identified and addressed, an organization can move on to the implementation steps, which will be discussed in the next session.

Implementation steps in CI/CD – a practical example

Applying behavioral design patterns within an organization's CI/CD processes can significantly enhance the efficiency and reliability of software development and deployment. First, it's essential to conduct a thorough analysis of the current CI/CD workflows and identify areas where design patterns can be integrated to solve specific problems or improve the process. Training and workshops can be organized to familiarize the development team with the core concepts of behavioral design patterns and their practical applications in CI/CD pipelines.

Once the team is knowledgeable about these patterns, the next step is to start small by integrating them into existing projects or new components of the CI/CD pipeline. For instance, the Command pattern can be applied to encapsulate deployment scripts or database migration commands, allowing greater flexibility and control over these operations. The Observer pattern can be implemented to create a notification system that alerts the relevant stakeholders about the status of the CI/CD processes, such as successful deployments or test failures.

It's also crucial to establish a culture of continuous improvement where the application of design patterns is regularly reviewed and optimized, based on feedback and evolving project requirements. This iterative approach ensures that the patterns effectively serve their purpose and contribute to the overall goals of the CI/CD initiative.

Moreover, documenting the use cases and outcomes of applying these patterns can serve as valuable references for future projects and help to scale the CI/CD processes across an organization. By sharing success stories and lessons learned, other teams can benefit from the experience and apply similar strategies in their workflows.

In addition to these steps, using tools and platforms to support the implementation of design patterns can streamline the process. Many CI/CD tools offer features that align with the principles of design patterns, such as modularization, automation, and configuration management. Selecting the right tools that complement the chosen design patterns can further enhance the effectiveness of CI/CD pipelines.

Let's illustrate this in the following diagram:

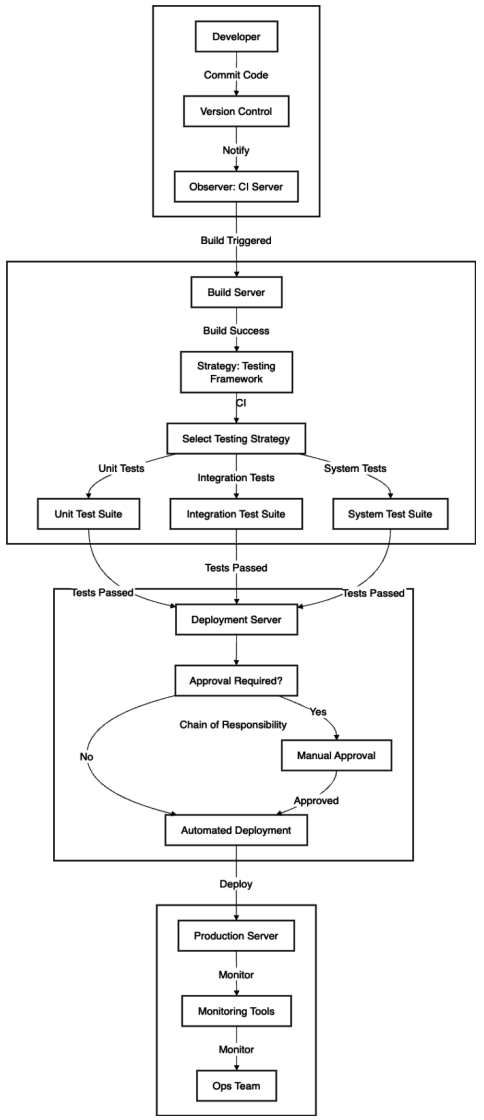

Figure 7.6 – The flow of applying behavioral design patterns

This diagram represents a CI/CD pipeline that uses behavioral design patterns to manage notifications, testing strategies, and deployment approvals effectively.

Finally, it's important to create an environment where collaboration is key for developers, testers, and operations teams, in order to work together in implementing and refining the use of design patterns in CI/CD. Encouraging open communication and knowledge sharing between team members results in innovative solutions and a more cohesive approach to software delivery.

By following these steps and continuously adapting to new challenges, organizations can successfully apply behavioral design patterns in their CI/CD processes, leading to more robust, maintainable, and efficient software delivery systems.

In conclusion, while the implementation of behavioral design patterns in CI/CD pipelines can be challenging, it is a critical step toward creating more efficient, reliable, and maintainable systems. By carefully considering the needs of the development and operations teams, adapting the patterns to the dynamic nature of CI/CD workflows, and ensuring seamless integration with existing tools, organizations can address these challenges and gain the benefits of a well-architected CI/CD pipeline.

Summary

In this chapter, we delved into the interesting world of behavioral design patterns within CI/CD, emphasizing the important role of observability in pipelines. We began with an overview, establishing a foundational understanding of how behavioral patterns can enhance the CI/CD process by providing deeper insights into system performance and user interactions.

Moving on to the design pattern principles, we discussed the holistic approach required to integrate observability effectively. This involves a shift in mindset from reactive to proactive monitoring, where the goal is to anticipate issues and improve system reliability and performance before problems arise.

The *Key components – tools and processes for AI-based observability* section highlighted the essential tools and processes for AI-based observability. We explored various AI-driven monitoring solutions that can predict potential bottlenecks and automate root cause analysis, thereby streamlining a CI/CD pipeline.

Implementation challenges were addressed, and we acknowledged the complexities of adopting new technologies. We outlined the first steps toward overcoming these hurdles, such as selecting the right tools, training teams, and setting clear goals. Some practical examples provided real-world context, illustrating how these concepts are applied in actual CI/CD workflows.

In summary, the integration of behavioral design patterns and AI-based observability into CI/CD is not just a trend but also a necessity for modern software development. It enables teams to build more resilient systems, react faster to changes, and deliver higher-quality software at a more rapid pace. As we continue to evolve in a technology-driven world, the principles and practices discussed in this chapter will become increasingly relevant, guiding us toward more efficient and effective CI/CD processes.

In the next chapter, we will explore **Domain-Driven Design** (**DDD**) patterns that assist product teams in managing the complexities and constraints unique to regulated domains.

8

Domain-Driven Design Patterns for Regulated Sectors

The **Domain-Driven Design** (**DDD**) approach to software design focuses on the creation of accurate models in the business domain. This design approach facilitates a better understanding of the problem domain and enables developers to create more expressive, maintainable, and scalable software solutions that accurately reflect real-world problems and opportunities. However, regulated sectors, such as finance, healthcare, or government, require specific patterns and practices to apply DDD effectively.

This chapter aims to explore DDD patterns that can help product teams deal with the complexity and constraints of regulated domains. Some of the topics covered include identifying and modeling the core domain and subdomains in a regulated context, using **bounded contexts** and **context mapping** to separate different models, and ensuring consistency and compliance. We will also look at designing and implementing domain services, aggregates, entities, and value objects that reflect the business rules and logic of the regulated domain. We do this using ubiquitous language and domain events to communicate and collaborate effectively with domain experts and stakeholders, as well as leveraging strategic and tactical patterns to create a modern, loosely coupled architecture that supports business goals and a vision.

By the end of this chapter, you will have a better understanding of how to apply DDD in regulated sectors and how to overcome some of the common pitfalls and challenges. Moreover, you will gain insights into some of the best practices and tips to implement DDD in how it relates to CI/CD and its pipelines.

We will cover the following topics in this chapter:

- An overview of a domain-driven CI/CD design pattern
- Domain-driven CI/CD design pattern principles – implementing regulation actions
- Key components – managing access control
- Key components – building artifacts according to regulations
- The importance of DDD

An overview of a domain-driven CI/CD design pattern

DDD constitutes a software design approach that endeavors to develop software systems that are capable of accurately reflecting the complexities of a particular domain. This approach is predicated on the invaluable input of domain experts, who possess a profound understanding of the unique intricacies and challenges of the domain in question. The comprehensive nature of this approach is instrumental in the development of software systems that are optimally suited to meet the needs of the domain.

In essence, DDD emphasizes organizing code in a manner that aligns with business problems and uniformly uses the same business terms or ubiquitous language throughout the development process. This approach to software development enables developers to build solutions that accurately represent the business domain and support effective communication between technical and non-technical stakeholders.

When utilizing DDD in building applications, problems are described as **domains**, while independent problem areas are referred to as bounded contexts. Each of them correlates to a microservice and it relates to a common language to discuss these problems. By using the correlation between a microservice and its bounded context, it becomes easier for developers to identify and define the entities, value objects, and aggregates that model the domain. The following figure represents a bounded context in CI/CD:

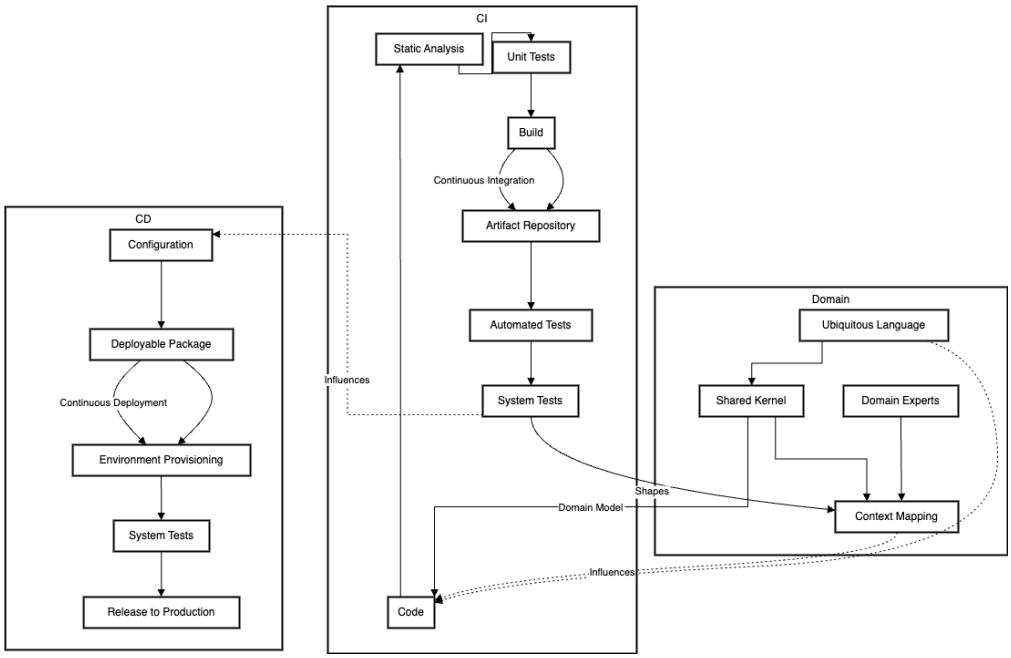

Figure 8.1 – A bounded context in several stages of CI/CD

This diagram captures the essence of a domain-driven CI/CD design pattern:

- **Domain layer**: Represents DDD concepts such as ubiquitous language, a shared kernel, and bounded contexts
- **CI layer**: Encompasses the CI stages, from static analysis to the creation of deployable artifacts
- **CD layer**: Encompasses the continuous deployment stages, including configuration, environment provisioning, system tests, and release to production

Furthermore, DDD patterns help developers understand the complexity of the domain, which is instrumental in developing a domain model for each bounded context. The domain model provides a clear picture of the domain, and this makes it easier to manage the deployment of individual microservices. Each bounded context can be developed, tested, and deployed independently, which aligns well with the principles of CI/CD.

Some of the benefits of using a domain-driven CI/CD design pattern are as follows:

- It improves communication and collaboration between developers, domain experts, and stakeholders by using a common language and shared understanding of the domain.
- It reduces software's complexity and technical debt by focusing on the core domain and its logic, avoiding unnecessary dependencies and coupling between different domains or bounded contexts.
- It increases software's quality and reliability by applying DDD patterns and practices, such as entities, value objects, aggregates, domain services, domain events, and repositories, to code and tests.
- It enables faster and safer software delivery by automating the CI/CD pipeline and using domain-specific tools and techniques, such as feature toggles, canary releases, blue-green deployments, and chaos engineering.
- Scalability can help align CI/CD processes with domain boundaries, so it becomes easier to scale individual components independently. This approach allows teams to focus on specific domains, making it simpler to manage and scale services as needed without affecting an entire system.
- Domain-driven CI/CD design patterns emphasize clear separation of concerns and well-defined boundaries. This approach simplifies the identification and resolution of issues within specific domains, leading to more efficient troubleshooting and enhanced maintenance. Additionally, it fosters better collaboration among teams, as each team can work within its domain without conflicting with other parts of a system.

A domain-driven CI/CD design pattern is not a prescriptive or rigid methodology but, rather, a flexible and adaptable approach that can be applied to different domains and scenarios. It requires close collaboration between the technical and domain teams, as well as a continuous feedback loop to refine the domain model and the delivery process.

However, it's essential to note that DDD should only be applied when implementing complex microservices with significant business rules. Simpler responsibilities such as **CRUD** (**Create**, **Read**, **Update**, and **Delete**) services can be managed with simpler approaches. Although DDD provides benefits such as maintainability, it is recommended only for complex domains where the model provides clear benefits in formulating a common understanding of the domain.

In the next section, we will take a look at the challenges of implementing regulations in CI/CD pipelines, based on the DDD patterns.

Domain-driven CI/CD design pattern principles – implementing regulation actions

One of the challenges of applying DDD to CI/CD is how to implement regulation actions into the pipelines. Regulation actions are the steps or tasks that ensure the compliance of software with the standards, policies, and regulations of the domain. For example, regulation actions can include code reviews, security scans, audits, approvals, or documentation. Regulation actions are important for ensuring the quality, safety, and legality of the software, as well as the trust and satisfaction of the customers and stakeholders.

However, regulation actions can also introduce complexity, overhead, and delays to the CI/CD pipelines. Regulation actions may require manual intervention, human judgment, or external verification, which can slow down the delivery process and create bottlenecks. Regulation actions can also vary, depending on the domain, the environment, or the context, which can make them difficult to standardize and automate. Regulation actions can also conflict with the principles of CI/CD, such as fast feedback, frequent iterations, and minimal waste.

Therefore, it is essential to apply DDD principles and patterns to design and implement regulation actions into the CI/CD pipelines in a way that balances the trade-offs between compliance and speed, quality and agility, and governance and autonomy. Some of the DDD principles and patterns that can help with this challenge are as follows:

- **Ubiquitous language**: This refers to the commonly used language by domain experts and developers to communicate and model a domain. Ubiquitous language helps to define and document regulatory actions, as well as the criteria, rules, and expectations for each action. It also aids in automating and integrating regulatory actions into the pipelines by using the same terms and concepts as the code and tools.
- **Bounded contexts**: These are the boundaries or scopes that define the subdomains or modules of the domains. Bounded contexts are a useful tool in DDD to identify and isolate regulation actions that are specific to each subdomain or module. This helps to avoid dependencies and conflicts that may arise from cross-cutting concerns. By using bounded contexts, you can customize and optimize the regulation actions for each subdomain or module, applying the appropriate level of rigor, frequency, and granularity.

- **Strategic design**: This is the process of analyzing and designing a domain at a high level, by identifying the core domain, the supporting subdomains, and the generic subdomains, and then applying the appropriate patterns and strategies for each. Strategic design can help prioritize and align the regulation actions with the business value and goals of the domain, avoiding over-engineering or under-engineering the software. Strategic design can also help simplify and streamline regulation actions, by eliminating or delegating the actions that are not essential or relevant to the core domain.
- **Context mapping**: This is the technique of visualizing and documenting the bounded contexts and their relationships, as well as the patterns and strategies that govern the communication and integration between them. Context mapping can help coordinate and synchronize the regulation actions across the bounded contexts, ensuring the consistency and compatibility of the software. Context mapping can also help monitor and manage the regulation actions, by identifying and resolving the issues or risks that may arise from the dependencies or conflicts between the bounded contexts.
- **Domain events**: These are the events or occurrences that reflect the changes or activities in a domain, as well as the data or information that is associated with them. Domain events can help trigger and execute regulation actions in pipelines, by using the events as signals or input for the actions. Domain events can also help track and audit the regulation actions, by using the events as records or output for the actions.

The following diagram shows the relationship and dependencies between these principles and patterns:

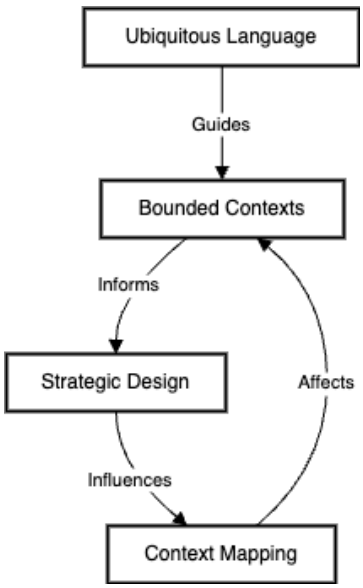

Figure 8.2 – The relationships and dependencies of DDD principles

Let's break down this diagram:

1. **Ubiquitous Language** guides the formation of **Bounded Contexts**. This means that the common language used by all team members helps define the logical boundaries, within which a particular model is defined and applicable.
2. **Bounded Contexts** inform **Strategic Design**. This means that the logical boundaries defined by the bounded contexts help shape the overall design strategy of the system.
3. **Strategic Design** influences **Context Mapping**. This means that the overall design strategy of the system affects how different bounded contexts interact with each other.
4. **Context Mapping** affects **Bounded Contexts**. This means that the interactions between different bounded contexts can lead to changes in the logical boundaries defined by the bounded contexts.

By applying these DDD principles and patterns, developers can design and implement regulation actions into the CI/CD pipelines that are domain-driven, context-aware, value-oriented, and event-driven. These regulation actions can help ensure the compliance of software with the domain standards, policies, and regulations, while also enabling the fast and reliable delivery of the software to customers and stakeholders. Incorporating DDD principles and patterns into CI/CD pipelines can significantly enhance the quality and compliance of software delivery. By focusing on the core domain and domain logic, developers can create more intuitive and robust systems. Regulation actions embedded within these pipelines ensure that each release adheres to predefined standards and regulations, thus maintaining domain integrity. Moreover, by being context-aware and event-driven, these pipelines can react dynamically to changes, ensuring that software remains aligned with business requirements while facilitating rapid and reliable deployment. This approach not only streamlines the development process but also fortifies the software against violations of domain standards, thereby safeguarding stakeholder interests and enhancing customer satisfaction.

Next, we are going to take a look at some key components, such as managing access control.

Key components – managing access control

Access control is a crucial aspect of DDD, particularly in regulated industries. Only authorized entities are allowed to interact with specific parts of the domain model, which reflects real-world permissions and constraints. In DDD, a layered architecture is commonly employed for access control, whereby policies are enforced at the application layer. This strategy enables a clear separation of concerns, with the domain layer focused on business logic and the application layer handling authorization.

Two patterns that can be used to implement access control in DDD are as follows:

- **Role-based access control (RBAC)**: RBAC links permissions with roles within an organization. It simplifies management as users change positions within an organization. You can see a simple example in this diagram.

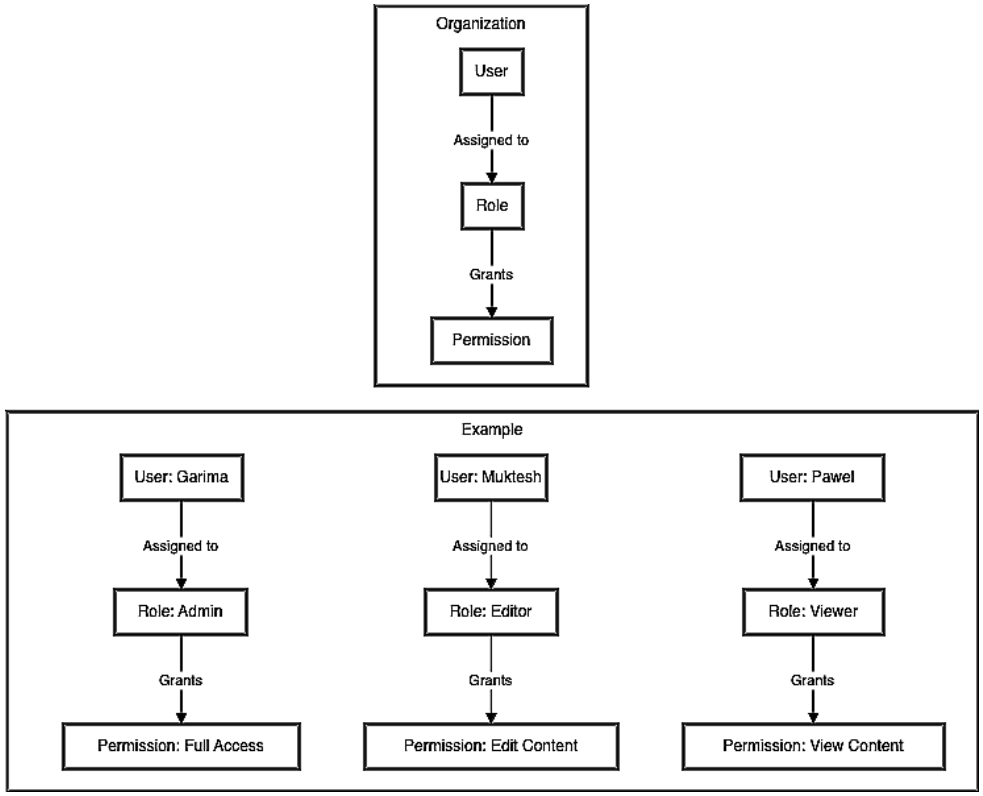

Figure 8.3 – An RBAC example

- **Attribute-Based Access Control (ABAC)**: ABAC considers various attributes of the user, action, and resource. It provides a more granular level of control by considering multiple attributes, such as user characteristics, environmental conditions, and resource information, allowing for dynamic permission adjustments based on a rich context. You can view this concept in the following diagram:

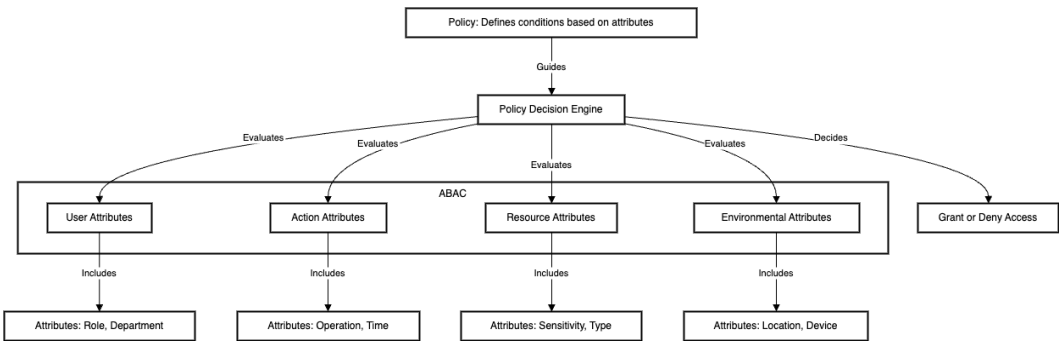

Figure 8.4 – An ABAC example diagram

These patterns are useful in modeling complex access control requirements that are typical in regulated environments. They ensure compliance and protect sensitive operations within a system and are essential in designing robust security models within DDD frameworks.

The management of access control is a critical aspect within DDD, particularly in regulated sectors. Access control in DDD is more than authentication and authorization. It is deeply rooted in a domain's intricacies and the bounded contexts that define its limits. The process involves identifying and enforcing boundaries where different domain models interact, with varying degrees of data sensitivity and transactional requirements.

The key components of access control in DDD for regulated sectors include the following:

- Ensuring that the entities and value objects within a domain are designed with security as a foundational aspect. This means that access rules and permissions should be directly incorporated into the domain models, allowing for a more granular and domain-specific approach to access control. For instance, an entity representing a user in a healthcare application should inherently understand the roles and permissions associated with that user, dictating what information can be accessed or modified.
- Furthermore, the use of aggregate roots in DDD provides a natural boundary for transactional consistency and security. By enforcing access control at the aggregate Root level, you ensure that all operations on the aggregate are authorized and the integrity of the domain is maintained. This is particularly important in regulated sectors where compliance with laws and regulations is paramount and any breach of data can have significant legal implications.
- Another critical component is the strategic implementation of repositories and factories, patterns that help manage the life cycle of domain objects. Repositories can enforce access control by only returning aggregates that a user is authorized to interact with, while factories can ensure that objects are created with the correct state and permissions from the outset.
- Furthermore, the application of a layered architecture, where domain logic is clearly separated from application logic, allows for a more robust access control mechanism. The application layer can act as a gatekeeper, ensuring that all interactions with the domain layer are authenticated and authorized according to the defined policies.
- Lastly, the concept of bounded contexts in DDD helps manage access control by clearly defining the boundaries within which certain models are valid. This helps to segregate a system into distinct areas with their access control policies, reducing the risk of unauthorized access across different parts of the system.

In regulated sectors, where the stakes are high and the cost of failure is significant, these DDD patterns and strategies are not just best practices but also necessities. They provide a structured approach to managing access control, ensuring that a system adheres to the stringent requirements of the sector while maintaining the flexibility to adapt to the ever-evolving landscape of regulations and security threats. Implementing DDD with a focus on robust access control mechanisms is essential for creating resilient, secure, and compliant systems in regulated environments.

Let's take a look at the following diagram, which represents a real-life healthcare application, to help clarify the key concept of access control within DDD.

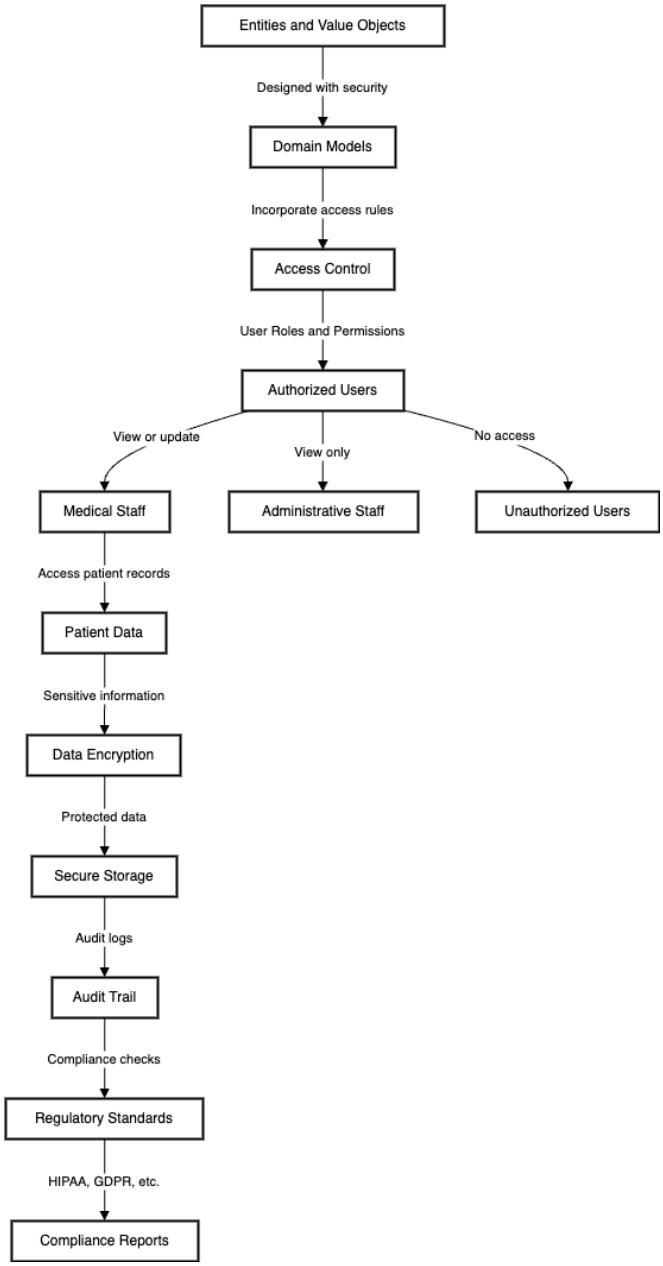

Figure 8.5 – Access control applied in a healthcare application following the DDD principles

The preceding diagram shows a robust and secure access control system that meets regulatory requirements while ensuring data confidentiality and integrity. The following components are represented:

- **Enforcing fine-grained authorization (security contexts)**:
  - Each user's security context includes their roles, permissions, and contextual information
  - Dynamic access decisions are made based on a user's context during runtime
- **Policy evaluation engine**:
  - Responsible for evaluating access control policies
  - Policies define rules for specific actions on specific resources
- **Authorization decisions**:
  - Based on policy evaluation, the system either allows or denies access
  - This ensures that users can only perform authorized actions
- **Data encryption**:
  - Sensitive data within the domain (e.g., patient records) is encrypted
  - Encryption keys are managed securely
- **Secure audit trail**:
  - Records all access events (successful or denied)
  - Helps with traceability, accountability, and incident investigation
- **Regulatory compliance**:
  - The system adheres to industry-specific regulations
  - Compliance ensures data privacy, security, and ethical handling
- **Audit reports**:
  - Regularly generated reports provide an overview of access patterns
  - These reports are essential for compliance audits and risk assessment

In conclusion, when designing access control for regulated sectors such as healthcare using DDD, it's evident that security is a foundational aspect. By incorporating access rules directly into domain models, we achieve a more granular and domain-specific approach. Key components include user roles, fine-grained authorization, data encryption, secure audit trails, and compliance with regulatory standards. These measures collectively safeguard patient data, maintain traceability, and uphold ethical practices.

Let's explore in the next section how artifacts in CI/CD relate to DDD, at the regulatory level.

Key components – building artifacts according to regulations

In the context of CI/CD, building artifacts according to regulations typically involves integrating regulatory requirements directly into the software development process, including the CI/CD pipeline. Here's a high-level approach to building artifacts compliant with regulations in terms of CI/CD:

- **Do a regulatory compliance requirements analysis**:
  - Identify the relevant regulations and compliance standards applicable to your domain or industry. This could include the **General Data Protection Regulation** (**GDPR**), **Health Insurance Portability and Accountability Act** (**HIPAA**), and **Payment Card Industry Data Security Standard** (**PCI DSS**), which are significant regulations that govern data protection and privacy.
    - GDPR is a comprehensive data protection law in the EU that sets guidelines for the collection and processing of personal information.
    - HIPAA establishes standards for protecting sensitive patient health information in the United States.
    - PCI DSS contains a range of security standards developed to be sure that all organizations accept, process, store, or transmit credit card information to maintain a secure environment.

  Together, these regulations form a robust framework for securing personal and sensitive data across various industries and regions. Compliance with these standards is mandatory for relevant entities and is critical for protecting individuals' privacy rights and securing transactional data against unauthorized access and breaches.

  Understand the specific requirements, such as data protection, privacy, security, and auditing, mandated by these regulations.

- **Examine domain modeling with compliance in mind**:
  - Ensure that your domain models reflect regulatory requirements. This involves designing entities, value objects, and aggregates with security and compliance as foundational aspects.
  - Incorporate access control, data encryption, audit logging, and other compliance-related features directly into your domain models.

- **CI/CD pipeline configuration**:
  - Integrate compliance checks into your CI/CD pipeline. This includes automated tests, static code analysis, security scans, and compliance audits.

    - Implement tools and frameworks that support compliance testing and validation, such as SonarQube, OWASP ZAP, or ComplianceAsCode. (ComplianceAsCode is a project that provides content for various compliance standards, including Red Hat STIG content. You can read more about it in the *Further reading* section).

- **Automated testing for compliance**:
    - Develop automated tests that verify compliance with regulatory requirements. These tests should cover aspects such as data protection, access control, encryption, and auditing.
    - Include several tests, such as unit tests, integration tests, and end-to-end tests, which validate compliance features across the entire application stack.
- **Continuous monitoring and reporting**:
    - Implement continuous monitoring tools to track compliance metrics and detect potential violations in real time.
    - Generate compliance reports and audit logs as part of the CI/CD process to demonstrate adherence to regulatory standards.
- **Deployment and release processes**:
    - Define deployment strategies that ensure artifacts are deployed securely and compliantly across different environments.
    - Implement approval workflows and RBAC to manage deployment permissions in the CI/CD pipeline.
- **Documentation and traceability**:
    - Maintain comprehensive documentation that details how regulatory requirements are addressed throughout the development life cycle.
    - Ensure traceability between regulatory compliance requirements, implemented features, automated tests, and deployment artifacts.
- **Continuous improvement and adaptation**:
    - Continuously review and update your CI/CD pipeline to incorporate changes in regulatory requirements and evolving best practices.
    - Encourage collaboration between development, operations, and compliance teams to find areas for improvement and ensure ongoing compliance.

By integrating regulatory compliance considerations into your CI/CD pipeline, you can build artifacts that meet regulatory requirements while maintaining the agility and efficiency of the development process.

Incorporating regulatory compliance within the CI/CD pipeline is essential for several reasons, such as the following:

- **Legal and industry standards**: Firstly, it ensures that software artifacts are developed in accordance with legal and industry standards, which is crucial for maintaining trust and avoiding legal penalties. Regulatory requirements such as GDPR, HIPAA, or PCI DSS often have stringent guidelines on data protection, privacy, and security, and non-compliance can result in significant fines and reputational damage.
- **Proactive compliance approach**: Secondly, integrating compliance into domain models and the CI/CD pipeline facilitates a proactive approach to compliance. By designing entities and processes with compliance in mind, organizations can prevent breaches and ensure that security measures are embedded within the development life cycle.
- **Early detection and resolution**: Moreover, automated testing and continuous monitoring within the CI/CD pipeline allow us to detect non-compliance issues early, enabling teams to address them promptly before they escalate into larger problems. This continuous feedback loop enhances the reliability and security of software.
- **Simplified auditing process**: Finally, by generating compliance reports and audit logs automatically, organizations can provide evidence of their adherence to regulatory standards, simplifying the audit process and demonstrating their commitment to compliance.

Overall, the integration of regulatory compliance into the CI/CD pipeline is a strategic approach that aligns software development practices with business risk management, ensuring that products meet both customer expectations and regulatory demands.

The importance of DDD

We discovered in this chapter that DDD is a powerful methodology that aligns complex software designs with business requirements, particularly beneficial in regulated sectors such as finance, healthcare, and government. These sectors present unique challenges due to stringent regulations, necessitating specific patterns and practices for effective DDD implementation. Practical examples of DDD in such contexts include the modeling of payment systems in financial services, where clear boundaries and responsibilities are crucial. Let's take a closer look at these examples:

- A DDD approach might involve defining a bounded context for payment processing and isolating it from other subsystems, such as account management or fraud detection. Within this bounded context, domain services, aggregates, entities, and value objects are meticulously designed to encapsulate the business rules and logic pertinent to payments.
- Moreover, context mapping ensures that different models interact coherently, maintaining consistency and compliance across a system. This is particularly important in healthcare, where patient data must be handled with utmost care to comply with regulations such as HIPAA in the United States. Here, DDD can help to segregate patient records management from

treatment protocols, each with its own ubiquitous language and domain events, facilitating clear communication among healthcare professionals and stakeholders.

- In government applications, DDD aids in structuring software systems that reflect the complex hierarchies and workflows inherent to public services. For example, a government portal for business licensing could use DDD to separate the domain of license application processing from that of compliance checking. Strategic and tactical DDD patterns enable the creation of a modern architecture that not only supports the business goals and vision but also adapts to the evolving needs of the public sector.
- The use of ubiquitous language and domain events in DDD ensures that all team members, including domain experts and stakeholders, have a common understanding of a system, which is crucial for collaboration and effective decision-making. This shared language helps to bridge the gap between technical implementation and business strategy, making the software solution more expressive, maintainable, and scalable.

In conclusion, DDD provides a structured approach to handle the complexities of regulated domains, allowing product teams to craft software solutions that are not only compliant with regulations but also aligned with business objectives. By focusing on the core domain and subdomains, employing bounded contexts, and leveraging strategic and tactical patterns, DDD facilitates the creation of software architectures that are robust, flexible, and capable of meeting the demands of regulated environments.

Performance considerations of DDD in CI/CD

We know how useful DDD can be, but we also need to address performance considerations. This is crucial for maintaining a smooth and efficient workflow when integrating DDD into CI/CD pipelines. With its focus on clear boundaries and a ubiquitous language, DDD can enhance the CI/CD process by ensuring that the code base is well-organized and that team members have a shared understanding of the system. However, it's important to balance the depth of domain modeling with the need for speed in CI/CD environments. Overly complex domain models can slow down integration and testing processes. To mitigate this, teams should aim for a modular architecture, enabling isolated testing and deployment, which aligns with CI/CD best practices of promoting the same build artifacts at each stage of the pipeline. Additionally, regular and frequent commits are encouraged to minimize integration issues and to benefit from the rapid feedback mechanisms inherent in CI/CD. By carefully considering these performance aspects, teams can leverage the strengths of DDD while optimizing their CI/CD pipelines for speed and reliability.

Some of the significant challenges are as follows:

- The potential for a mismatch between the complex models of DDD and the streamlined processes of CI/CD. This can slow down the delivery pipeline if the domain model is too granular or the team needs a clear strategy for integrating DDD into CI/CD practices. Another pitfall is the underestimation of the learning curve associated with DDD, which can lead to delays and confusion as team members adapt to this approach within the CI/CD context.

- Additionally, there's a risk of neglecting proper communication and collaboration, essential for aligning the DDD approach with CI/CD workflows. To avoid these pitfalls, it's crucial that organizations ensure that the DDD models are aligned with the CI/CD goals, invest in training for team members, and have an open culture of communication and collaborative problem-solving.

By being aware of these common issues, teams can better navigate the integration of DDD into their CI/CD pipelines, leading to more efficient and effective software development processes.

Let's have a sneak peek at how DDD in CI/CD would apply in a real-life scenario.

An example of DDD in CI/CD

Suppose you participate in a project for your organization to model, using the DDD principles of an entirely new payment system, and you are assigned to focus on delivering the system to the end user according to all modern practices.

In such a scenario, the focus would be on creating a model that accurately reflects the intricate business processes involved in financial transactions. The situation might involve a customer attempting to make a payment through an online platform, which requires a robust system to handle various payment methods and currencies while ensuring security and compliance with financial regulations.

The technologies used in such a DDD implementation could include a combination of programming languages such as Java or C#, databases such as SQL Server or Oracle, and messaging systems such as RabbitMQ or Kafka for asynchronous communication. The framework chosen might be Spring for Java or .NET for C#, which offers extensive support for building enterprise applications.

The code would be structured around the core concepts of DDD, such as entities, value objects, aggregates, and domain events, ensuring that the business logic is clearly separated from the infrastructure and application layers.

In terms of CI/CD, the development process would involve continuous integration of code changes into a shared repository, automated testing to validate the functionality, and CD to move the changes into production. Tools such as Jenkins or GitLab CI could be used to automate the CI/CD pipeline, allowing for frequent releases and quick feedback loops. Containerization technologies such as Docker and orchestration platforms such as Kubernetes might also be employed to manage the deployment of microservices, which are often used in DDD to encapsulate bounded contexts.

The CI/CD pipeline would be designed to support DDD's emphasis on evolving models, enabling a team to iteratively refine the domain model and the corresponding code. Automated tests would play a crucial role in ensuring that the domain logic remains consistent and correct as changes are made. The pipeline would also facilitate collaboration between domain experts and developers, allowing for a shared understanding of the domain to be reflected in the code base.

Overall, the combination of DDD with modern technologies and CI/CD practices enables financial services organizations to build scalable, flexible, and maintainable payment systems that can adapt to

changing business requirements and technological advancements. This approach not only improves the quality and reliability of the software but also aligns the development process with the strategic goals of the organization.

Perhaps the following overview of the different components and their relationships will clarify the concepts more:

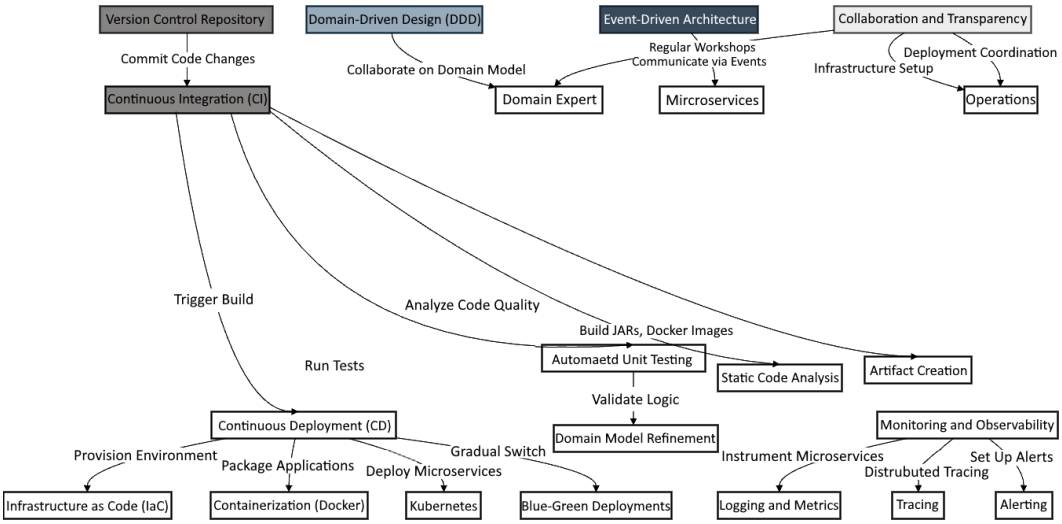

Figure 8.6 – DDD applied in practice

The preceding figure shows a diagram that illustrates the interaction between various components and practices in a software development pipeline, particularly focusing on DDD principles within CI/CD processes:

- **Version control repository**: This is where developers commit their code changes. Further actions that they need to do are as follows:
  - Set up a Git repository (e.g., GitHub, GitLab, or Bitbucket) to manage the code base
  - Encourage all team members to commit their code changes to this repository
  - Use branching strategies (such as GitFlow) to organize feature development, bug fixes, and releases
- **CI**: This phase involves automatically triggering a build upon code changes. It includes running automated unit tests, analyzing code quality, and building artifacts such as JARs or Docker images. Further actions needed for this phase are as follows:
  - Choose a CI tool (e.g., Jenkins, Travis CI, or CircleCI)

    - Configure your CI pipeline to do the following:
        - Automatically trigger builds on code commits
        - Run automated tests (unit tests and integration tests)
        - Analyze code quality (linting and code style checks)
        - Create artifacts (JARs and Docker images)

- **CD**: After successful CI, the code is deployed continuously. This involves provisioning the environment using **Infrastructure as Code** (**IaC**), packaging applications into containers, deploying microservices with Kubernetes, and implementing blue-green deployments for seamless updates. Implement these CD practices to automate deployment:
    - **IaC**:
        - Use tools such as Terraform, OpenTofu, and AWS CloudFormation to define your infrastructure
        - Provision environments (development, staging, and production) consistently
    - **Containerization (Docker)**:
        - Package your applications into Docker containers
        - Use Docker Compose for local development and Kubernetes for production
    - **Kubernetes**:
        - Deploy microservices using Kubernetes
        - Leverage features such as scaling, rolling updates, and service discovery
    - **Blue-green deployments**:
        - Gradually switch traffic from old to new versions
        - Minimize downtime during deployments
- **DDD**: Developers collaborate with domain experts to refine the domain model. Automated unit testing helps validate the logic, and an event-driven architecture is employed to communicate between microservices. Collaborate with domain experts (e.g., business stakeholders and product owners) to do the following:
    - Understand the problem domain deeply
    - Refine the domain model
    - Identify bounded contexts and aggregates
    - Define ubiquitous language

- **Collaboration and transparency**: This phase emphasizes collaboration between developers, operations teams, and domain experts. It includes setting up infrastructure, coordinating deployments, and conducting regular workshops to align on domain understanding.
- **Event-driven architecture**: Implement event-driven communication between microservices by doing the following:
  - Using message brokers (e.g., Kafka and RabbitMQ) or event buses
  - Publishing events when domain entities change
  - Subscribing to relevant events for data synchronization
- **Monitoring and observability**: The operations team monitors the deployed system, instrumenting microservices for logging and metrics, implementing distributed tracing for debugging, and setting up alerts for proactive monitoring.
- **Collaboration and transparency**: Foster collaboration among development, operations, and business teams by doing the following:
  - Regularly communicating infrastructure changes
  - Coordinating deployments and releases
  - Sharing knowledge and best practices

The diagram in *Figure 8.6* illustrates how DDD principles are integrated into CI/CD practices, emphasizing collaboration, automation, and continuous feedback. This ensures that the development process aligns with the strategic goals of an organization and delivers high-quality software.

Remember that successful implementation involves more than just technology – it also requires cultural shifts, team alignment, and continuous improvement. Adapt these principles to your specific context and iterate based on feedback and lessons learned.

A real-life case study

Now that we've looked at an example, it is also worth investigating a real case study where DDD is applied. Consider an e-commerce company that adopts DDD within its CI/CD framework. The company begins by identifying its core domain, which is the online shopping experience. It then models its software around this domain, creating a ubiquitous language that is shared among developers, business analysts, and stakeholders.

In this case study, the company's development team structures its software into distinct bounded contexts, such as inventory management, order processing, and customer relations. Each bounded context corresponds to a microservice that is developed, tested, and deployed independently within the CI/CD pipeline. This modular approach allows for rapid iteration. Another benefit is the deployment of new features without disrupting the entire system.

The CI/CD pipeline is configured to automate testing and deployment processes. Every code commit triggers a series of automated tests that validate the integrity of the domain model and ensure that new changes align with the business objectives. If a test fails, the pipeline stops, and developers are notified to address the issue, ensuring that only quality code is deployed to production.

Moreover, the company implements feature toggles, allowing features to be tested in production without affecting all users. This aligns with DDD's emphasis on experimentation and learning within the domain. By leveraging feature toggles, the company can gather real-time feedback and make informed decisions about the domain model's evolution.

The adoption of DDD in CI/CD also embraces a culture of collaboration and continuous learning. Regular domain events and workshops are held, where team members discuss domain complexities and refine the ubiquitous language. This collaborative environment ensures that the software remains closely aligned with business needs and can adapt to changes swiftly.

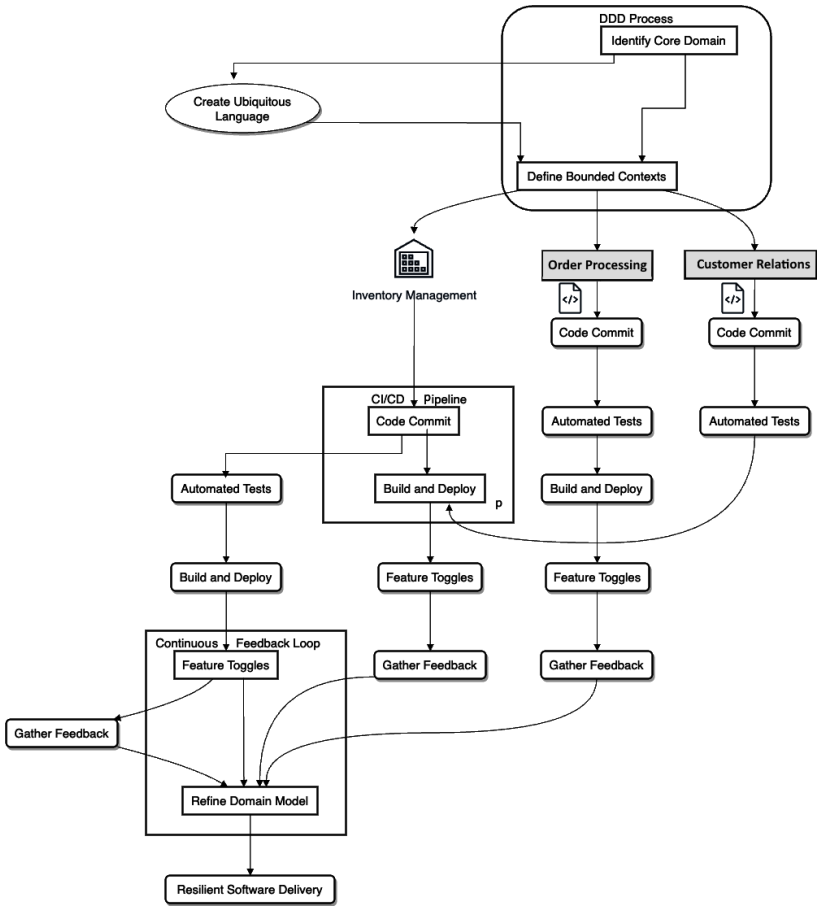

Figure 8.7 – The integration flow of DDD in a CI/CD pipeline of an e-commerce company

This flow shows us the parts of a successful integration of DDD in CI/CD:

1. **Identify Core Domain**: The company starts by identifying the core business domain (e.g., online shopping).
2. **Create Ubiquitous Language**: It develops a shared language that all stakeholders understand.
3. **Define Bounded Contexts**: The company breaks down the domain into key areas (bounded contexts), such as **Inventory Management**, **Order Processing**, and **Customer Relations**.
4. **CI/CD Pipeline**:
    I. Code Commit: Developers commit code for each microservice
    II. Automated Tests: Automated tests run to validate the changes
    III. Build and Deploy: If tests pass, the microservices are built and deployed
5. **Feature Toggles and Feedback**:
    - **Feature Toggles**: Allow new features to be gradually released without affecting all users
    - **Gather Feedback**: Collect feedback on these features to make improvements
6. **Refine Domain Model**: Use the feedback to refine the domain model continuously.
7. **Final Outcome**: The process leads to resilient and adaptable software delivery.

Through this integration of DDD with CI/CD, the e-commerce company achieves a more resilient, adaptable, and business-aligned software delivery process. It exemplifies how DDD principles can be applied to real-world scenarios to improve software development practices and outcomes. While this case study is hypothetical, it demonstrates the tangible benefits that can be realized when DDD is thoughtfully applied within a CI/CD context.

Practical implementation of DDD in CI/CD

Implementing DDD in your CI/CD pipeline involves a strategic alignment of software development practices with business requirements. Following these steps will lead to a successful implementation:

1. The process begins with a deep understanding of the business domain for which the software is being developed. This involves close collaboration with domain experts to create a ubiquitous language that accurately reflects the domain complexities and is shared among all team members.
2. Once the domain is thoroughly understood, the next step is to define bounded contexts. These are clear boundaries within the domain model that determine how the domain is partitioned into logical segments, each representing a different aspect of the business. In a CI/CD pipeline, these bounded contexts translate into independently deployable microservices, each with its own database and logic, allowing for isolated development, testing, and deployment.

3. The CI/CD pipeline should be designed to support DDD by automating the deployment of these microservices. Each code commit can trigger an automated build and test sequence, ensuring that changes do not break the domain model or the business logic. Automated testing should include unit tests, integration tests, and domain-specific tests that validate the business rules and behaviors defined by the ubiquitous language.
4. In addition to automated testing, the CI/CD pipeline should facilitate continuous deployment. This means that every successful build can be deployed to staging or production. Feature toggles can be used to manage the release of new features, allowing them to be selectively enabled or disabled without redeploying the entire service.
5. Also crucial are the monitoring and logging of a DDD-aligned CI/CD pipeline. They provide insights into how a system behaves in production and can inform domain model refinements. Observability tools can help track the flow of domain events and ensure that the system's behavior aligns with business expectations.
6. Furthermore, it's important to embrace a culture of continuous learning and improvement. Regularly scheduled domain events, such as workshops or retrospectives, can help a team reflect on the domain model and the CI/CD processes, adjusting as necessary to better serve the business goals.
7. Lastly, version control plays an important role in managing the evolution of the domain model. It allows you to track changes, collaborate on code, and maintain a history of the domain logic over time. Proper branch management strategies can help in managing features and fixes, ensuring that the main branch always reflects the most stable and up-to-date version of the domain model.

You can implement DDD in your CI/CD pipeline by following these steps, resulting in a robust, agile, and business-centric software delivery process. Remember, the key to successful DDD implementation is the continuous alignment of technology with business needs, facilitated by a well-structured CI/CD pipeline.

Summary

This chapter explored DDD patterns that can help product teams deal with the complexity and constraints of regulated domains. By applying DDD principles and patterns, product teams can create software solutions that accurately reflect the business domain, support effective communication between technical and non-technical stakeholders, and align with the business goals and vision. The integration of DDD with the CI/CD pipeline through a domain-driven CI/CD design pattern can help reduce software complexity and technical debt while improving communication and collaboration between developers, domain experts, and stakeholders.

In this chapter, you learned about DDD patterns for regulated sectors and best practices to implement CI/CD pipelines, with a focus on regulations. You also learned various skills, such as understanding the impact of regulations on CI/CD design, applying access control and policies to a CI/CD toolset, and

recognizing the correlation between regulations and building artifacts. You gained insights into the principles and patterns of DDD, including identifying and modeling the core domain and subdomains in a regulated context, using bounded contexts and context mapping, designing and implementing domain services, aggregates, entities, and value objects, using ubiquitous language and domain events, and leveraging strategic and tactical patterns to create a modern, loosely coupled architecture. You now have a better understanding of how to apply DDD in regulated sectors, as well as how to overcome common challenges and implement best practices in CI/CD pipelines.

In the next chapter, we will see how we can apply creation design patterns in combination with cloud technology.

Further reading

Here, you can find the sources to expand your knowledge about the specific concepts in this chapter:

- *Domain-Driven Design – Tackling Complexity in the Heart of Software* by Eric Evans: This book is considered the authoritative guide to DDD and provides a comprehensive overview of the principles and practices involved.
- *Implementing Domain-Driven Design* by Vaughn Vernon: This book provides a practical guide to implementing DDD in real-world scenarios, including examples of how to model complex domains and integrate DDD into existing systems.
- *Domain-Driven Design Distilled* by Vaughn Vernon: This book provides a more concise overview of DDD principles and practices and is a good starting point for those new to the concept.
- ComplianceAsCode documentation for more on security guidelines – *STIG* and *SCAP*: `https://complianceascode.readthedocs.io/en/latest/manual/user/01_introduction.html`.

Part 4: Creational CI/CD Design Patterns

In this part, you will get to understand the application of creational design patterns and their application in the CI/CD pipeline, followed by real-world examples from different cloud providers. Further on, you will learn how to apply measurements to understand the state and design your delivery pipeline accordingly.

This part has the following chapters:

- *Chapter 9*, *Applying Creational CI/CD Design Patterns*
- *Chapter 10*, *Understanding Deployment Strategies – Creational CI/CD with Cloud Providers*
- *Chapter 11*, *Auditing and Assessment of Design Patterns*

9
Applying Creational CI/CD Design Patterns

We briefly introduced the concepts of creational patterns in the *An overview of design patterns* section of *Chapter 1*. This chapter will delve deeper into these patterns and explain how they integrate **continuous integration** (**CI**) and **continuous deployment** (**CD**) techniques with cloud technologies.

This chapter delves into the transformative approach of applying **creational CI/CD patterns**, a methodology that integrates the principles of CI/CD with the power of cloud computing. You will go on a journey through the automation of the software development life cycle, exploring how code commits can seamlessly transition into production releases. The chapter highlights the important role of containerization, microservices, and serverless computing in enhancing efficiency and ensuring applications are robust enough to thrive in the cloud's dynamic environment. By adopting these innovative practices, development teams can not only speed up their development cycles but also improve system uptime and maintain the agility required to adapt to changing requirements. This results in a smoother path from development to deployment, optimizing applications that are suitable for the limitless potential of the cloud.

This chapter covers the following main topics:

- Creational design patterns – the concept
- Creational CI/CD design pattern principles – scalability and resilience
- Key components – containerization, microservices, and the serverless ecosystem
- Key components – ensuring security and compliance
- Implementing CI/CD pipelines based on creational design patterns

First, let's look into the concept of creational design patterns.

Creational design patterns – the concept

Creational design patterns are a fundamental concept in software development that looks at the process of object creation. These patterns have the ability to improve the flexibility and reusability of code by simplifying the instantiation process. By doing this, the system becomes less dependent on how objects are created, composed, and represented. They contain knowledge about the system's concrete classes and abstract the details of object creation and assembly. This approach is useful in scenarios where the system should not be dependent on the method used to create its objects.

One of the key benefits of creational patterns is their ability to reduce the complexities and instabilities associated with direct object creation, primarily through the use of the `new` operator, which can lead to scattered object creation throughout an application. By controlling the creation process, these patterns help avoid the tight coupling and inflexibility that can arise from hardcoded object creation.

There are several types of creational design patterns, each addressing different aspects of object creation:

- **Factory Method pattern**
- **Abstract Factory pattern**
- **Singleton pattern**
- **Prototype pattern**
- **Builder pattern**

Each design pattern has its own unique set of use cases and is selected based on the specific needs of the design problem at hand. In practice, these patterns can be implemented in various programming languages and are used to solve common problems in object-oriented design. They provide tested and proven solutions to complex issues, allowing developers to build more robust, flexible, and maintainable software systems. Creational design patterns are not just theoretical concepts but practical tools that have been applied successfully in numerous software projects to address the challenges of object creation and management. They are an essential part of the toolkit for any software developer looking to write high-quality, scalable, and efficient code.

Creational design patterns in the context of CI/CD are crucial for cloud-based software development. These patterns provide a blueprint for creating objects in a system, which is particularly beneficial when dealing with cloud environments' dynamic and scalable nature. In CI/CD, creational patterns can help manage the life cycle of services and applications by defining how instances are created, which is essential for automating deployment processes. For example, a Factory Method pattern is used to create objects without telling the exact class of object that will be made, allowing for greater flexibility and reuse of code. This aligns with the principles of CI/CD, where the goal is to automate the integration and deployment processes, enabling frequent and reliable releases into the cloud environment. By utilizing creational patterns, developers can ensure that their CI/CD pipelines are robust and capable of handling the complexities of cloud deployments, ultimately leading to more efficient and error-free releases.

Creational patterns can greatly enhance the modularity and scalability of the build pipelines. For instance, the Singleton pattern ensures only one instance per class and a global point of access to it, which can be very useful for managing configurations and states across a pipeline.

Moreover, the Abstract Factory pattern is used to create related or dependent objects without concrete classes. This is useful with multiple environments in CI/CD, such as staging and production, which may require different types of objects to be created. The Builder pattern also plays a crucial role in CI/CD by allowing the creation of complex objects step by step. This can be applied to the setup of a build environment, where various components need to be initialized and configured before a build can proceed.

Overall, creational design patterns facilitate the development of flexible and maintainable CI/CD pipelines by encapsulating the creation logic and promoting loose coupling and high cohesion. They enable developers to create more robust, scalable, and efficient automation processes, which are essential for the rapid and reliable delivery of software products.

Creational CI/CD design pattern principles – scalability and resilience

The principles of creational design patterns in CI/CD play a crucial role in achieving scalability and resilience. These patterns are the architectural blueprints that guide the instantiation of software components, ensuring that systems are not only robust but also flexible enough to adapt to changing demands. In this specific context, scalability refers to the ability of a CI/CD pipeline to handle increased loads gracefully by scaling resources up or down as needed. This is crucial for maintaining performance levels without incurring unnecessary costs or experiencing downtime.

Resilience, on the other hand, is the capacity of the CI/CD system to remain operational even when parts of it fail. This involves designing for failure, with patterns that allow for quick recovery and minimal disruption. For instance, implementing redundancy, where multiple instances of a service run in parallel, will prevent the entire system from coming down. Moreover, circuit breakers can halt cascading failures, letting a system continue functioning in a degraded state rather than failing completely.

In cloud computing, the principles of scalability and resilience are paramount, especially when considering the design patterns for CI/CD. Scalability ensures that a system can handle varying workloads efficiently by adjusting resources, while resilience refers to the system's ability to maintain functionality despite potential failures. For instance, Google Cloud suggests that a well-designed application should scale up or down as demand changes and be resilient enough to withstand service disruptions. This involves careful planning and design, incorporating autoscalers with virtual machines or Kubernetes clusters, and utilizing serverless platforms that manage compute and database services. Moreover, resilience is achieved through architectural considerations that span infrastructure, network design, and data storage strategies. It also extends to organizational culture, emphasizing the importance of learning from failures to enhance system robustness. These principles are theoretical and essential for modern architecture exercises, ensuring that applications remain operational and efficient under various conditions and demands.

Creational design patterns offer several benefits for cloud computing:

- **Consistency and configuration management**: The Singleton pattern ensures that cloud applications use a consistent set of configurations across different environments, which is critical for maintaining uniform behavior and reliability in the cloud.
- **Scalability**: The Factory Method and Abstract Factory patterns allow for the creation of objects without specifying their exact classes. This flexibility is essential in cloud environments where resources need to scale up or down based on demand.
- **Resource efficiency**: The Builder pattern provides a clear process for constructing complex objects, which can help optimize resource allocation and usage in the cloud, leading to more efficient performance.
- **Rapid prototyping**: The Prototype pattern enables the quick creation of object copies, facilitating faster development and deployment cycles in the cloud. This is essential for CI and continuous delivery workflows.

These patterns contribute to the robustness and agility of cloud applications, ensuring they are scalable, resilient, and maintainable.

The creational design patterns, discussed in the *Creational design patterns – the concept* section of this chapter, such as the Singleton, Factory Method, Abstract Factory, Builder, and Prototype patterns, provide a structured approach to object creation. These patterns encapsulate the logic of creating instances, which can be particularly beneficial in a CI/CD context where different environments might require different configurations.

Singleton pattern

When looking at these patterns, the Singleton pattern has classes with a single instance and a global point of access to it, which can be useful for managing shared resources across a pipeline.

Now let's look at a simplified example of it in this figure:

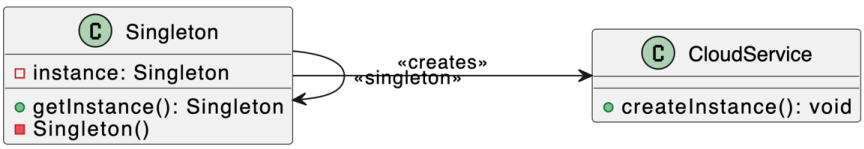

Figure 9.1 – A Singleton pattern applied in the context of creating a cloud instance

In this diagram, you can see the following:

- The `Singleton` class represents the single instance that is shared across the entire application.
- The `getInstance()` method ensures that only one instance of `Singleton` is created and returned.
- In a cloud CI/CD context, the `CloudService` class might be responsible for creating the `singleton` instance, ensuring consistency across different environments.

The Singleton pattern is important in a CI/CD pipeline for several reasons:

- **Consistency**: By ensuring there's only one instance, you can ensure that every part of your pipeline is working with the same, consistent set of data. This is particularly useful in a CI/CD pipeline where you might have a shared resource, such as a configuration file or a database connection. By ensuring there's only one instance, you can ensure that every part of your pipeline is working with the same, consistent set of data.
- **Controlled access**: Singleton allows controlled access to its sole instance, which can be useful for managing resources that are shared across a CI/CD pipeline. For example, if multiple processes need to write to a shared log file, the Singleton pattern can help ensure that access to the file is coordinated properly.
- **Lazy initialization**: Singleton objects are initialized only when needed. This can improve performance and resource usage in a CI/CD pipeline, especially when dealing with heavy resources.
- **Reduced overhead**: By limiting the number of instances of a particular class to one, we can save on resources and reduce overhead, which is especially important in a CI/CD pipeline where efficiency and speed are key.

Factory Method pattern

The Factory Method pattern allows a class to defer instantiation to subclasses, enabling the creation of objects without specifying their exact classes. This is especially useful when the system needs to be extended with new types of objects; the CI/CD pipeline can incorporate these without major changes to the existing code base. Now, let's apply this pattern to a diagram to elaborate more on this in the context of CI/CD:

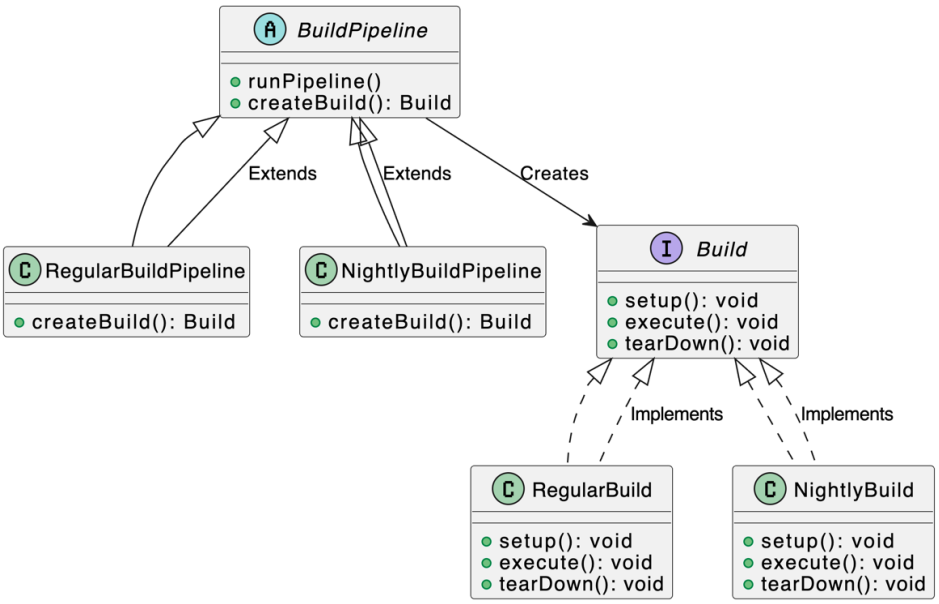

Figure 9.2 – A Factory Method pattern applied in the context of CI/CD

In this diagram, you can see the following components, which are important for this pattern:

- `BuildPipeline` is the abstract creator class and declares the factory method, returning an object of type `Build`.
- `RegularBuildPipeline` and `NightlyBuildPipeline` are concrete classes that implement the factory method to produce `RegularBuild` and `NightlyBuild` instances, respectively.
- `Build` is the abstract product interface that declares the operations that the build is able to do.
- `RegularBuild` and `NightlyBuild` are concrete classes that implement the `Build` interface.

The Factory Method pattern is important in a CI/CD pipeline for several reasons:

- **Flexibility**: The Factory Method pattern allows the system to be flexible when it comes to creating new types of objects. This is especially useful when the system needs to be extended with new types of objects. The CI/CD pipeline can incorporate these without major changes to the existing code base.
- **Loose coupling**: The Factory Method pattern promotes loose coupling, which is a design principle aimed at reducing the interdependencies between different parts of the program. This means that changes in one part of the system will have minimal impact on others.

- **Code reusability and maintenance**: By encapsulating the object creation code into a factory method, we can reuse this code, making the system easier to maintain and extend. When a new type of object needs to be added, we only need to create a new concrete factory class without changing the existing code.
- **Single-responsibility principle**: Each factory class has a single responsibility: to create objects of a specific type. This makes the system easier to understand, test, and maintain.

Abstract Factory pattern

The Abstract Factory pattern goes a step further by having an interface for the creation of family-related or dependent objects without specifying the concrete classes, promoting consistency across different environments. We can see this in the following diagram:

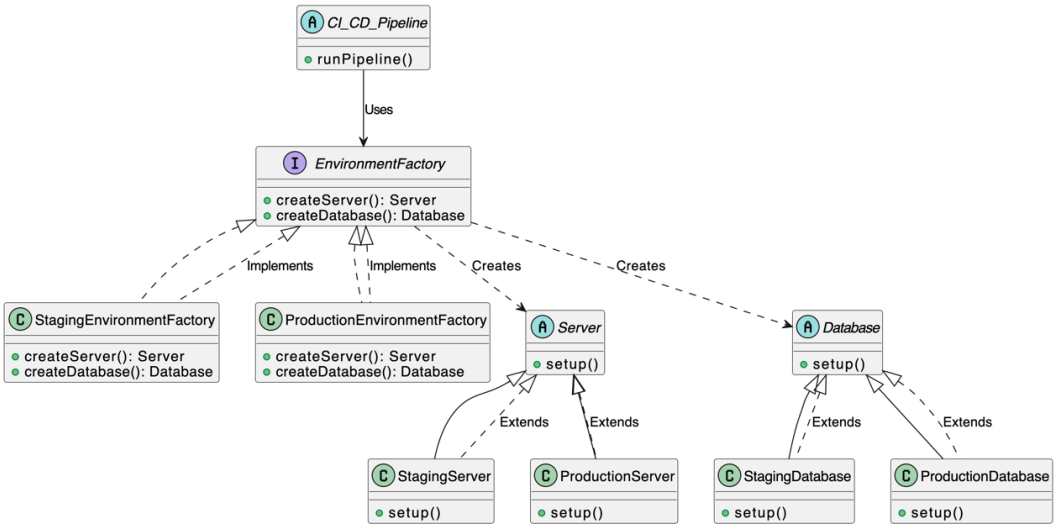

Figure 9.3 – The Abstract Factory pattern in the context of CI/CD

In this diagram, the following is represented:

- `CI_CD_Pipeline` is the client that uses an `EnvironmentFactory` method to create the `Server` and `Database` object.
- `EnvironmentFactory` is the Abstract Factory that declares the creation methods.
- `StagingEnvironmentFactory` and `ProductionEnvironmentFactory` are the concrete factories that implement the creation methods.

- `Server` and `Database` are the abstract products.
- `StagingServer`, `ProductionServer`, `StagingDatabase`, and `ProductionDatabase` are the concrete products.

In terms of CI/CD, the Abstract Factory pattern is beneficial for the following:

- **Environment setup**: It allows the creation of different environments (such as staging and production) with their own set of related objects (such as a server and database).
- **Pipeline configuration**: Different factories can be used to configure the pipeline based on the environment or other factors.
- **Consistent deployments**: It ensures that the pipeline uses consistent sets of objects that work well together.
- **Scalability**: New types of environments or components can be added with minor changes to the pipeline.
- **Code organization**: It encapsulates the creation logic, making the pipeline code cleaner and easier to maintain.

After looking at these benefits, let's continue looking a bit closer to the builder pattern.

Builder pattern

The Builder pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations. This can be applied to setting up different stages of a CI/CD pipeline, where each stage might have different requirements and configurations.

We can graphically display this in the following figure:

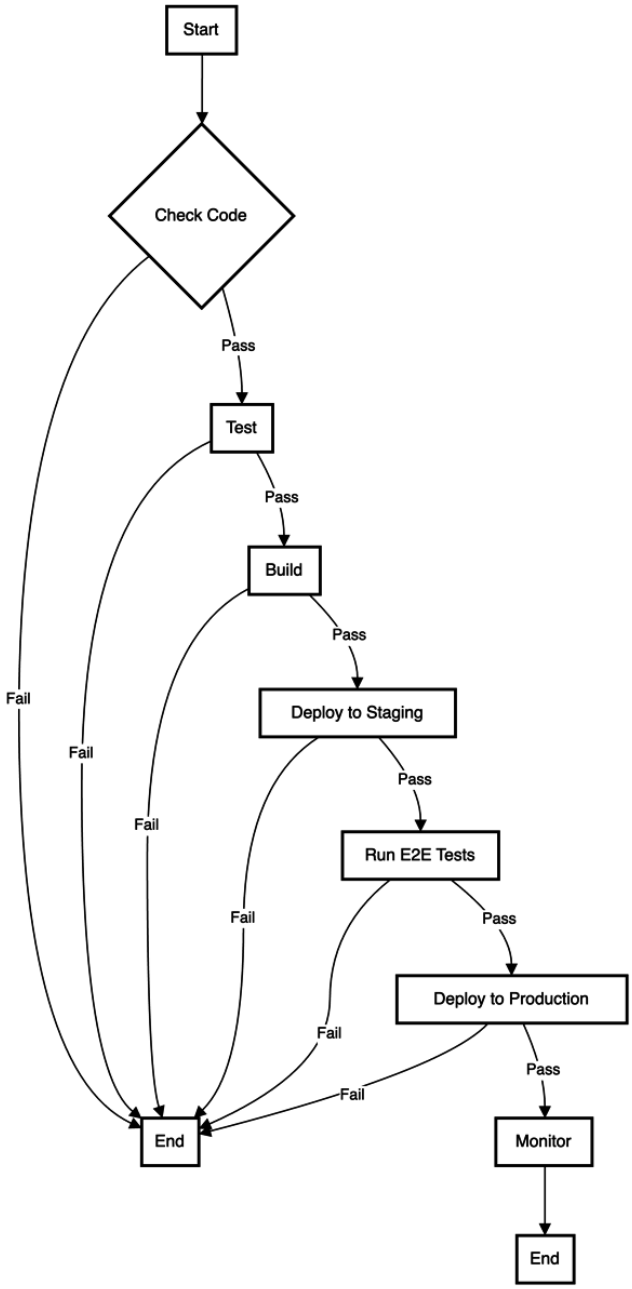

Figure 9.4 – The Builder pattern in CI/CD contexts

In this example, each stage of the pipeline can be seen as a build step in the Builder pattern:

1. Check code.
2. Test.
3. Build.
4. Deploy to staging.
5. Run E2E tests.
6. Deploy to production.
7. Monitor.

The pipeline starts at the beginning and progresses through each stage:

- It only moves to the next stage if the current stage passes
- If any stage fails, the pipeline ends

Next, we will look at the Prototype pattern.

Prototype pattern

Lastly, the Prototype pattern refers to a development approach where a prototype is built early in the software development life cycle. This prototype, which is a working model of the desired system, is then continuously improved and refined through iterative development and testing within the CI/CD pipelines. The advantage of this pattern is that it allows for early feedback and integration, ensuring that the final product is well aligned with user needs and system requirements. It also supports the CI/CD philosophy of making tiny, incremental changes that can be easily tested and deployed.

Let's have a look at it in this diagram:

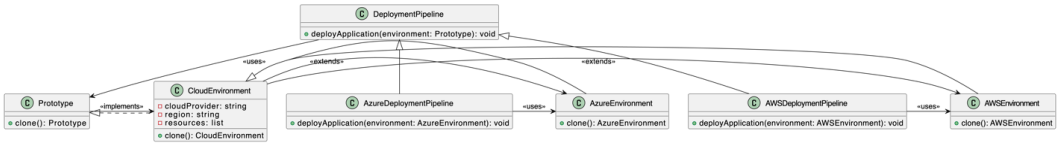

Figure 9.5 – An example of a Prototype pattern

In this figure, the following components are included:

- The `Prototype` class defines the interface for cloning objects.
- `CloudEnvironment` represents a cloud environment with properties such as `cloudProvider`, `region`, and `resources`.

- Concrete prototypes (`AzureEnvironment` and `AWSEnvironment`) extend `CloudEnvironment` and implement the `clone()` method.
- `DeploymentPipeline` uses the prototype to deploy applications to specific cloud environments.

The Prototype pattern is useful in CI/CD for several reasons:

- **Efficiency**: It allows the efficient creation of complex objects by cloning existing ones.
- **Consistency**: It ensures consistent configurations across different stages of the pipeline.
- **Flexibility**: It allows the easy customization of cloned objects based on specific needs.
- **Scalability**: New types of configurations can be added with minimal changes to the pipeline.
- **Resource management**: It helps manage resources effectively by reusing existing objects.

Now, all these patterns come from software development, but in the context of really building CI/CD, these creational patterns are meant to be more conceptual, rather than used in a practical sense. We've included them to illustrate the broader concepts.

Let us now look at some more relevant CI/CD creational patterns. In the context of CI/CD, *creational patterns* is not such a commonly used term. Nevertheless, there are certain practices and strategies that are foundational to setting up a CI/CD pipeline, which could be seen as *creational* in nature.

CI/CD cloud-native creational design patterns

Cloud-native CI/CD patterns represent a change in how software development and deployment processes are conceived and executed. These patterns are specifically designed to leverage the dynamic, scalable, and resilient nature of cloud environments.

Let's have a closer look at these patterns and how they are practically used:

- In the cloud, an important goal is to build on **cloud-native standards**, such as containers, microservices, orchestration, and immutable infrastructure, mentioned next.
- The use of **immutable infrastructure**, where changes are made by replacing components rather than altering existing ones, ensures consistency and reliability across environments, as each deployment is a replica of the previous one, mitigating *it works on my machine* syndrome. Another pattern is the microservices architecture, which permits individual services to be deployed and scaled independently, enhancing the agility of the CI/CD process.
- The **Infrastructure as Code** (**IaC**) principle enables the automatic provisioning and management of infrastructure through code, which can be version-controlled and reused. This not only accelerates the deployment process but also enhances the reproducibility and traceability of changes. Additionally, the adoption of containerization and orchestration tools such as Kubernetes facilitates the management of containerized applications, ensuring they are deployed consistently, irrespective of the underlying infrastructure.

- **Infrastructure agnostic**, which means CI/CD must be applicable to any system, regardless of the underlying infrastructure.
- **Infinite and dynamic scaling options**, which means solutions must land on infrastructure with dynamic scalability.
- **Configuration as code** is a pattern allowing teams to define the entire build and deployment process as code, which can be managed and versioned just like application code. This pattern aligns with the practice of CI, where developers frequently merge code changes into a central repository, followed by automated builds and tests. The continuous delivery aspect extends this by automatically deploying the application to a production-like environment, ensuring that the code is always in a deployable state.

  Looking at this pattern, the use of Helm charts can optimize Kubernetes and container components, making your CI/CD pipeline highly reliable. Another part of this pattern, **GitOps** is a method that utilizes familiar tools to automate infrastructure while applying the principles of Git, such as version control, collaboration, compliance, and CI/CD. This approach enhances the developer experience by ensuring that the production environment aligns with the state described in a Git repository. Not only does it streamline deployment, but it also simplifies error recovery and credential management, making it a powerful practice for managing cloud-native applications and infrastructure.
- **Observability** is also integral to cloud-native CI/CD, providing insights into the system's state and the impact of changes. This is achieved through comprehensive logging, monitoring, and alerting systems that enable the quick identification and resolution of issues.
- Finally, the pattern of **continuous feedback** is essential, where the CI/CD pipeline is designed to collect feedback from all stages of the development life cycle and use it to improve the process continuously. This includes feedback from automated tests, code quality checks, security scans, and user acceptance testing.

Let's make this clearer with the following diagram:

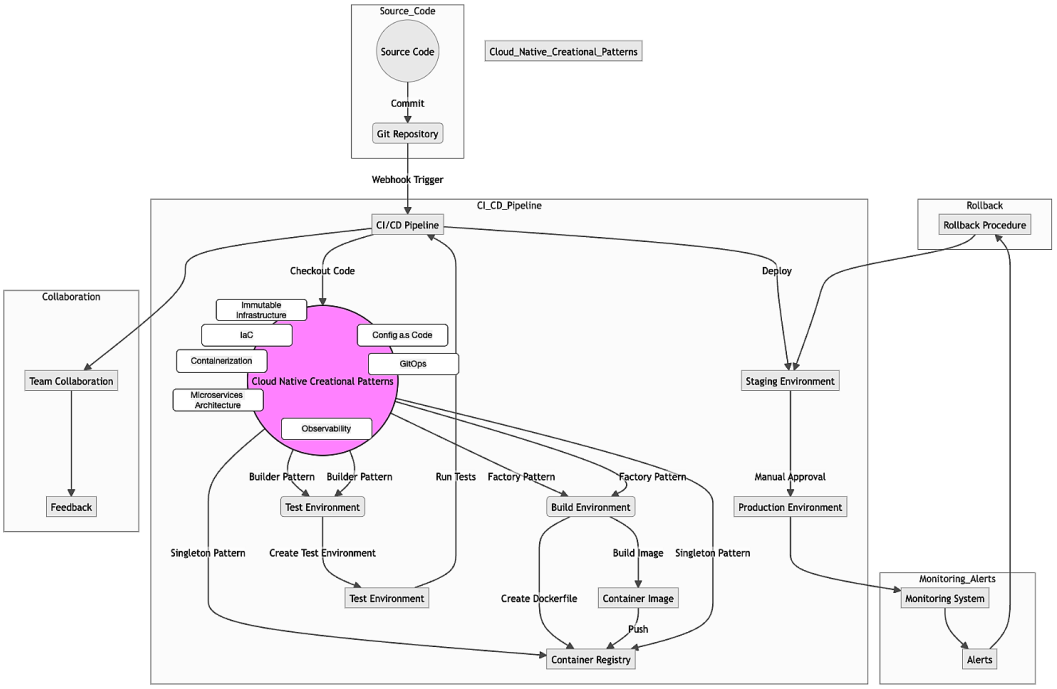

Figure 9.6 – Creational cloud-native CI/CD pattern

This diagram shows how a creational pattern can be used in the cloud-native world and consists of the following components:

- **Source control (repo)**:
  - **Feature branch**: Developers create feature branches to work on new features or bug fixes. These branches are isolated from the main code base to avoid conflicts.
  - **Release branch**: Once features are complete and tested, they are merged into the release branch, which is prepared for deployment.
- **CI/CD pipeline**:
  - **Continuous integration** (**CI**): When code is pushed to the repository, the CI process is triggered. This includes building the code and running automated tests in the build environment.
  - **Continuous deployment** (**CD**): After passing the CI checks, the code moves through various environments for further testing and validation.

- **Environments**:
  - **Build environment**: The initial stage where the code is compiled and built. If the build is successful, it moves to the next stage.
  - **Test environment**: Automated tests are run to ensure the code works as expected. This includes unit tests, integration tests, and so on.
  - **QA environment**: Quality assurance tests are conducted here. This environment is used for more extensive testing, including manual testing.
  - **Staging environment**: This is a pre-production environment that closely mirrors the production environment. Final tests are conducted here to ensure everything works correctly before going live.
  - **Production environment**: The live environment where the application is deployed for end users.
- **Monitoring tools**: Tools to monitor the application in the production environment. They provide alerts if any issues arise, allowing for quick responses to problems.

These components work as follows:

1. **Code development**: This starts in the feature branch.
2. **Code review and merge**: Code is reviewed and merged into the release branch.
3. **CI/CD pipeline**: The code goes through the CI/CD pipeline, moving through the build, test, QA, and staging environments.
4. **Deployment**: Finally, the code is deployed to the production environment.
5. **Monitoring**: The application is monitored for any issues post-deployment.

This flow ensures that code is thoroughly tested and validated at each stage before reaching the end users, maintaining high quality and reliability.

This diagram also emphasizes all the important parts of a cloud-native CI/CD pattern, such as the following:

- **Immutable infrastructure**: Creating components that are not modified after initial setup for consistency, reliability, and reproducibility
- **Microservices architecture**: Breaking down an application into smaller, loosely coupled services for agility, scalability, and fault isolation
- **IaC**: Treating infrastructure provisioning and management as code for faster deployment and improved traceability
- **Containerization and orchestration**: Packaging applications and managing container deployment, scaling, and networking for simplified maintenance

- **Configuration as code**: Defining and managing configuration settings as code alongside application code for consistent configuration across different environments
- **GitOps**: Automating infrastructure management using Git repositories for simplified error recovery and enhanced security
- **Observability**: Comprehensive logging, monitoring, and alerting systems for the quick identification of problems and efficient troubleshooting
- **Continuous feedback**: Collecting feedback from all stages of the development life cycle for continuous improvement in the CI/CD process

When discussing cloud-native CI/CD patterns, it's important to remember that they aim for a declarative approach to automation, decreasing complexity and errors in deployment. These patterns emphasize automation, consistency, and resilience, all of which are crucial for achieving high velocity and reliability in software delivery. Understanding key components of cloud native is also essential when utilizing these patterns.

Key components – containerization, microservices, and the serverless ecosystem

In the evolving landscape of cloud-native software development, key components such as **containerization**, **microservices**, and the **serverless ecosystem** stand at the forefront of architectural design, particularly in the realm of cloud native and CI/CD. Containerization encapsulates an application's code, configurations, and dependencies into a single object, fostering portability and consistency across environments. Microservices architecture decomposes complex applications into smaller, independent services, which enhances scalability and facilitates the CI/CD pipeline's agility. The serverless ecosystem further abstracts infrastructure management, allowing developers to focus on code and deploy it in a pay-as-you-go model, which seamlessly integrates with the CI/CD methodology. Together, these components embody the creational design patterns that are evident in modernizing applications, ensuring they are robust, scalable, and continuously delivered with efficiency and reliability. In this section, we'll take a deeper dive into these key components and what roles they play in CI/CD.

Containerization

Containerization plays an important role in the realm of the cloud and CI/CD, particularly when viewed through the lens of creational design patterns. You might have heard about containers and what they are used for, but let's look at a short overview of the concept.

A **container** is a lightweight, standalone unit of software with all the necessary components to run an application. These components consist of code, runtime, system tools, system libraries, and settings. Containers are self-contained and isolated from each other and the host system, making them easily portable across different cloud environments. This encapsulation streamlines the development, testing, and deployment processes, enabling more flexibility and scalability in application management.

Containers play a significant role in microservices architectures, where applications are constructed as a set of autonomous services that can be individually deployed and scaled.

In the context of CI/CD, containerization aligns closely with CI/CD design patterns by encapsulating the environment and dependencies of an application, much like how creational patterns encapsulate the instantiation process of objects.

The essence of containerization lies in its ability to create lightweight, reproducible, and portable containers. These containers serve as the perfect embodiment of the Prototype pattern, where a single container can serve as a prototype for creating subsequent containers quickly and efficiently. This is especially useful in CI/CD pipelines, where the need for rapid testing and deployment is paramount. Each container can be seen as a *prototype* that is cloned, modified, and deployed across various stages of the pipeline, ensuring consistency and reliability.

Moreover, containerization can be likened to the Factory Method pattern, where the container runtime acts as a factory that produces containers. This method abstracts the complexities of creating objects, similar to how container runtimes abstract the complexities of creating and running containers. The client, in this case, the CI/CD pipeline, is not concerned with the intricate details of container creation but is assured that the container produced will be in a usable state.

The following figure illustrates this graphically:

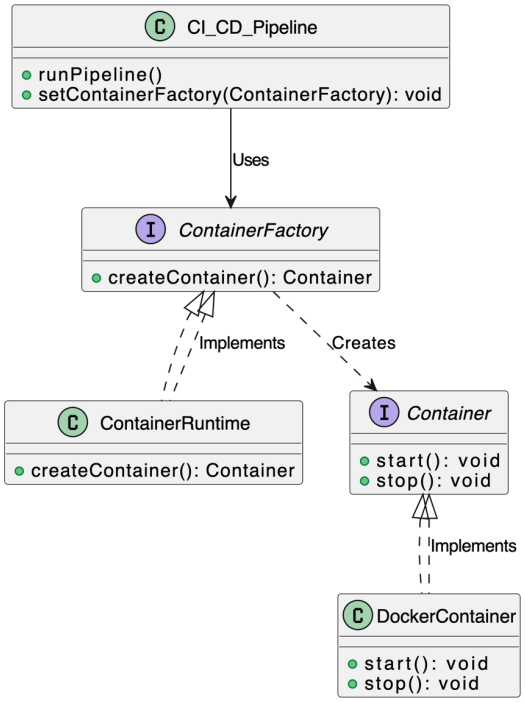

Figure 9.7 – Factory Method pattern creates containers

The components of the diagram can be understood as follows:

- `ContainerFactory` is the abstract factory that declares the creation method, `ContainerRuntime` is a concrete class that implements the factory method to produce `Container` instances, and `Container` is the abstract product interface that declares the operations that the container is able to do.
- `DockerContainer` is a concrete class that implements the `Container` interface.
- `CI_CD_Pipeline` is the client that uses a `ContainerFactory` class to create the `Container` objects.

A simple but more practical example of the Factory Method pattern in a CI/CD pipeline is using Docker, which handles different types of builds that require different configurations. For instance, you might have a `JavaBuild` method for Java applications and a `PythonBuild` method for Python applications. Each of these builds requires a different Docker image.

Here's a simplified Java example using the Docker SDK:

```
import java.io.IOException;

abstract class Build {
    abstract Process createContainer() throws IOException;
}

class JavaBuild extends Build {
    @Override
    Process createContainer() throws IOException {
        return new ProcessBuilder("docker", "run", "-d", "openjdk:8-
jdk-alpine").start();
    }
}

class PythonBuild extends Build {
    @Override
    Process createContainer() throws IOException {
        return new ProcessBuilder("docker", "run", "-d", "python:3.7-
alpine").start();
    }
}

public class Main {
    static void runPipeline(Build build) throws IOException {
        Process container = build.createContainer();
        // Now we can interact with the container
    }
```

```
    public static void main(String[] args) throws IOException {
        runPipeline(new JavaBuild());
        runPipeline(new PythonBuild());
    }
}
```

In this example, `Build` is the *creator* class that declares the `create_container` factory method. `JavaBuild` and `PythonBuild` are *concrete creators* that implement the factory method to produce specific types of containers. The `run_pipeline` function is agnostic to the type of build it's running – it simply calls the `create_container` method on the build object it's given. This is the essence of the Factory Method pattern: it encapsulates the object creation code in a method (the factory method), allowing subclasses to decide which class to instantiate. This makes the code more flexible and easier to extend – for example, to support new types of builds, you can simply create new subclasses of `Build`.

The Builder pattern also finds its parallel in containerization. Just as the Builder pattern separates the construction of a complex object from its representation, containerization allows for the separation of an application's configuration from its execution environment. This separation enables developers to define container specifications in a container build file or similar configuration files, which can then be used to build the container image.

Lastly, the Singleton pattern is reflected in the orchestration of containers. Container orchestration tools such as Kubernetes ensure that the desired state of the application, often defined as a single source of truth in a configuration file, is maintained across all containers. This ensures that there is a consistent environment for the application, having a single instance in the Singleton pattern.

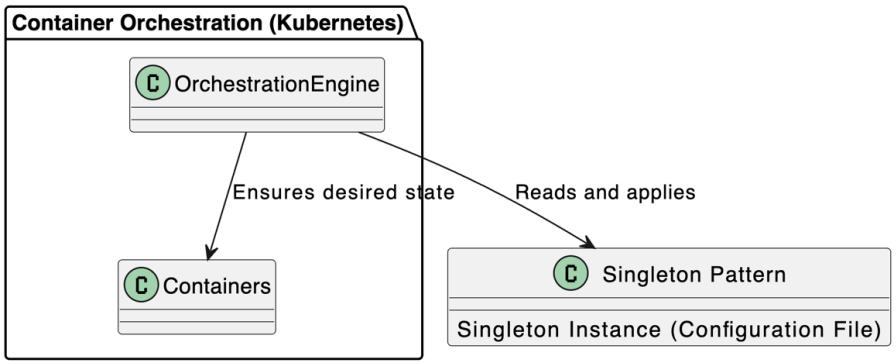

Figure 9.8 – The Singleton pattern in a container orchestration such as Kubernetes

In *Figure 9.9*, `Singleton Instance (Configuration File)` represents the single source of truth for the application's desired state. `OrchestrationEngine` reads from this configuration file and ensures that the containers maintain this desired state.

To show the Singleton pattern in a code example, look at the following Singleton pattern, which can be applied to manage shared resources. For instance, consider a logging service used across multiple services in the pipeline. This logging service should ideally be a Singleton, ensuring all services are writing to the same log source.

Here's an example using a Kubernetes ConfigMap:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: logging-config
data:
  log_level: "Warning"
```

In this example, `logging-config` is a ConfigMap that acts as a Singleton. It provides a single, global configuration (`log_level`) that all containers in the Kubernetes cluster can access. This ensures consistency across the system, as all containers will be using the same logging level.

In conclusion, containerization is not just a technology that facilitates the deployment of applications but also a manifestation of creational design patterns within the CI/CD process. It encapsulates best practices of object creation and systematizes the deployment process, ensuring that applications are developed, tested, and deployed in a consistent, efficient, and scalable manner. By understanding the synergy between containerization and creational design patterns, organizations can leverage both to enhance their CI/CD pipelines and, ultimately, their delivery of software products. This understanding is crucial for creating robust, scalable, and maintainable CI/CD systems that can adapt to the ever-evolving demands of software development.

Microservices

Microservices architecture is a methodology for software development that emphasizes breaking down applications into more minor, independently deployable services. Each service operates its own process and communication using lightweight mechanisms, usually an HTTP resource API.

The CI/CD pipeline is of importance for the microservices deployment process, enabling CI/CD. It automates the building, testing, and deployment of microservices, facilitating a faster and more reliable delivery process. This pipeline is often integrated with cloud-native services, further enhancing the agility and scalability of the microservices architecture.

This architectural style of microservices aligns perfectly with the principles of CI/CD design patterns, which focus on frequent, automated, and reliable software delivery processes. Each microservice can be integrated and delivered continuously, without waiting for a complete, monolithic application to be finalized. This means that new features, updates, and fixes can be rolled out rapidly and more safely. Containerization supports this by packaging services with their dependencies, ensuring that they work uniformly across various environments. This is crucial for CI/CD as it eliminates the *it works on my machine* problem, enabling predictable deployments.

Service discovery and API gateways are pivotal in a microservices CI/CD pipeline. Service discovery ensures that services can dynamically discover and communicate with each other in a cloud environment, which is essential for the automated deployment processes in CI/CD. API gateways facilitate this communication and provide a single point of entry for all service requests, which simplifies the complexity of interactions between microservices and external clients.

Data domains ensure that each microservice can own its database schema and logic, which aligns with the CI/CD principle of independent service deployment. Changes to a service's data domain do not impact other services, allowing for non-disruptive, continuous delivery. The serverless ecosystem takes this a step further by abstracting the infrastructure management away from developers, which fits the CI/CD model of focusing on code and business logic rather than on servers and hardware.

The orchestration of all these components within a CI/CD pipeline enables a smooth flow from development to production. Automated testing is integrated at various stages, ensuring that each service is verified before it is deployed. This reduces the risk of errors in production and allows for a faster feedback loop. The CI/CD pipeline also facilitates rollback mechanisms, so if a new service version causes issues, it can be quickly reverted to a stable state.

Serverless

Serverless architecture represents a new approach to building, deploying, and managing applications. It's a more efficient model because it abstracts the server layer, providing a more streamlined and cost-effective way to manage computing resources.

The key components of serverless architecture include containerization, microservices, and the serverless ecosystem, all of which work together to enable the creational CI/CD pattern. Containerization is a way to package an application's code along with its dependencies, ensuring consistency across different computing environments and simplifying portability. Microservices architecture breaks down applications into smaller, autonomous services that can be developed, deployed, and scaled individually, making them more agile and resilient. The serverless ecosystem comprises cloud services that manage server provisioning and scaling, allowing developers to focus on writing code that serves business logic.

In the context of CI/CD, serverless architectures streamline the development pipeline by automating the deployment process, enabling CI and continuous delivery with minimal configuration. This approach aligns with the creational pattern in CI/CD, which emphasizes the importance of creating resources on demand, ensuring that the infrastructure and services are available exactly when needed, without unnecessary overhead.

By leveraging serverless components, organizations can achieve faster time to market, reduce operational costs, and increase the reliability of their applications. For example, serverless computing can reduce the need for in-house IT staff, as it simplifies the management of computing resources and infrastructure. This makes it easier and more cost-effective to scale and maintain applications as business needs change.

The integration of these elements forms a cohesive and modern approach to software development that aligns with the principles of agile and DevOps practices. Serverless computing is about revolutionizing

the developer experience and operational efficiency. It enables developers to focus on writing code that meets business needs, rather than worrying about the underlying infrastructure.

Now, let us explore some specific characteristics of serverless in a CI/CD pipeline. When we break down the specifics of a serverless CI/CD pipeline, we can discover the following:

- **Function-based deployment**: Traditional applications are usually deployed as a whole. In contrast, serverless applications are composed of individual functions that can be deployed and scaled independently. This means that the CI/CD pipeline needs to handle the deployment of these individual functions.
- **Event-driven testing**: Serverless functions are typically triggered by events. Therefore, testing in a serverless CI/CD pipeline often involves simulating these events to trigger the functions.
- **IaC**: Serverless architectures often rely heavily on IaC tools such as AWS CloudFormation or the serverless framework. These tools allow developers to define their serverless infrastructure in code, which can be version-controlled and deployed through the CI/CD pipeline.
- **Cold starts**: Serverless functions are stateless and can be shut down when not in use, leading to *cold starts* when they are invoked again. This is a unique aspect of serverless applications that might need to be considered in the CI/CD pipeline.
- **Integration with serverless platforms**: The CI/CD pipeline needs to integrate with the serverless platform (such as AWS Lambda, Google Cloud Functions, or Azure Functions) that is used to run the functions.
- **Managing artifacts across multiple accounts and regions**: Serverless CI/CD pipelines often need to manage artifacts across multiple accounts and regions.
- **Using the minimum number of permissions required to deploy the application**: Serverless CI/CD pipelines should use the minimum amount of permissions required to deploy the application.

These specifics make the CI/CD pipeline for serverless applications different from traditional application deployment. The pipeline needs to handle the deployment, testing, and scaling of individual functions, manage cold starts, integrate with serverless platforms, and more. These aspects introduce new challenges and considerations in the CI/CD process.

A real-life example of microservices in a CI/CD ecosystem

Imagine a hotel booking application designed using a microservices architecture. The application comprises several independent services, each responsible for a specific function within the booking process. For instance, one service handles user authentication, another manages room inventory, a third processes payments, and another takes care of reservations. The way the implementation is done is as follows:

- Each service is a self-contained unit running in its own container, which includes all the necessary code, runtime, system tools, libraries, and settings. Containerization ensures that each service

can be deployed, scaled, and managed independently across various environments – from a developer's perspective, to a production server – without compatibility issues.

- Kubernetes will manage these containers. It also aids in service discovery, allowing each service in the hotel booking application to dynamically discover and communicate with other services via a service registry to track all the available services and their endpoints.
- An API gateway is implemented at the front of this architecture, acting as the single entry point for all client requests. It routes each request to the appropriate microservice and facilitates a layer of abstraction that decouples the client interface from the backend services. This gateway also offers additional security measures, such as authentication and rate-limiting, to protect the microservices from unauthorized access or overuse.
- Within the business domain, data domains are defined to encapsulate the business logic and data for specific capabilities. For example, the reservation service has its data domain, which includes the logic for booking rooms and maintaining the reservation data. This separation ensures that each service can operate independently and maintain its data integrity without being affected by changes in other services.
- The serverless ecosystem, a key part of the architecture, allows for certain functionalities, such as sending confirmation emails or processing analytics, to be handled by serverless functions. These functions are event-driven and only consume resources when triggered, making the application cost-effective and highly scalable. This scalability instills confidence in the system's capabilities, knowing it can handle any workload efficiently.

This implementation can be better understood with the following diagram:

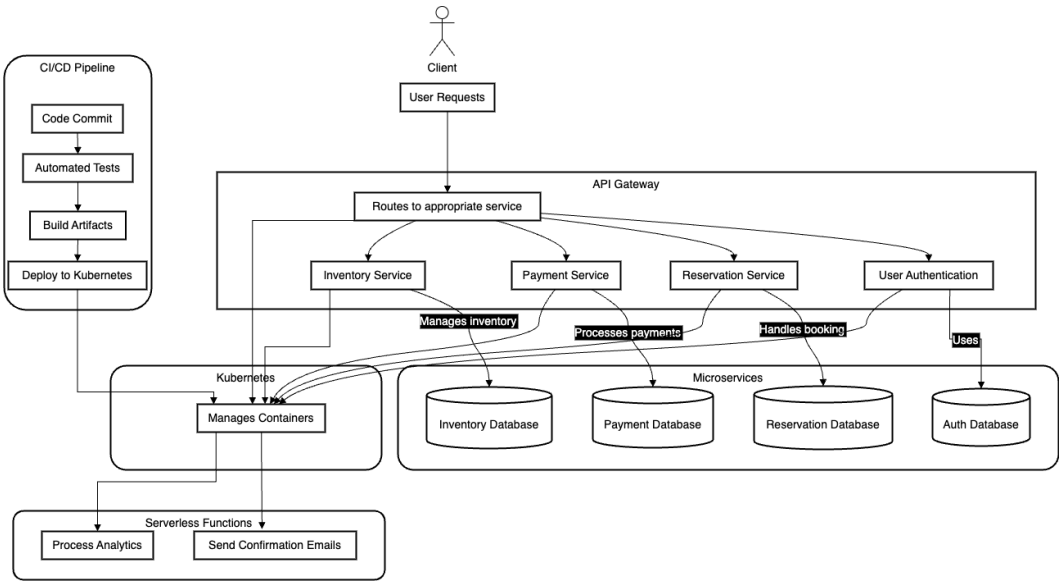

Figure 9.9 – Microservices and serverless ecosystem implementation in CI/CD

Ultimately, the microservices architecture empowers the hotel booking application with resilience, flexibility, and scalability. It paves the way for CI/CD, enabling the development team to roll out updates swiftly and efficiently without disrupting the entire application. This architecture not only simplifies the development and maintenance of complex applications but also aligns with modern business needs for agility and speed, making the team adaptable and responsive.

In summary, the microservices architecture offers numerous benefits, including improved scalability, flexibility, and speed of deployment. However, it also presents challenges such as increased complexity and the need for a robust infrastructure. Understanding and implementing these components effectively is crucial for organizations looking to adopt a microservices architecture in their software development life cycle.

Next, we will have a look at components such as security and compliance.

Key components – ensuring security and compliance

Ensuring security and compliance in CI/CD pipelines is crucial for maintaining the integrity and reliability of software development and deployment processes. Key components include implementing DevSecOps principles, which integrate security practices within the DevOps process. This approach emphasizes the *shift-left* security paradigm, advocating for security measures to be considered early in the development cycle. It also encourages a collaborative approach between development, security, and operations teams to foster a culture of shared responsibility for security.

Code scanning and analysis are also vital, utilizing tools to identify vulnerabilities. Additionally, threat modeling, code provenance, and **software composition analysis** (**SCA**) are employed to ensure the security of both the code and its dependencies. Integrating these tools and their results provides a comprehensive view of the security posture throughout the CI/CD pipeline.

Container security and IaC security are increasingly important as they deal with the security of the infrastructure itself. Proper secrets management is another crucial aspect, involving the implementation of access controls and separation of duties to enforce permissions and prevent unauthorized access to sensitive data.

On the compliance side, policy enforcement is essential, with practices such as policy as code and the use of policy enforcement engines to automate and enforce compliance rules. Compliance adherence can be further strengthened by automating compliance checks and implementing compliance as code, which ensures that compliance requirements are met programmatically.

So, how would a CI/CD pipeline look when implementing these components? Let's have a look at this flow:

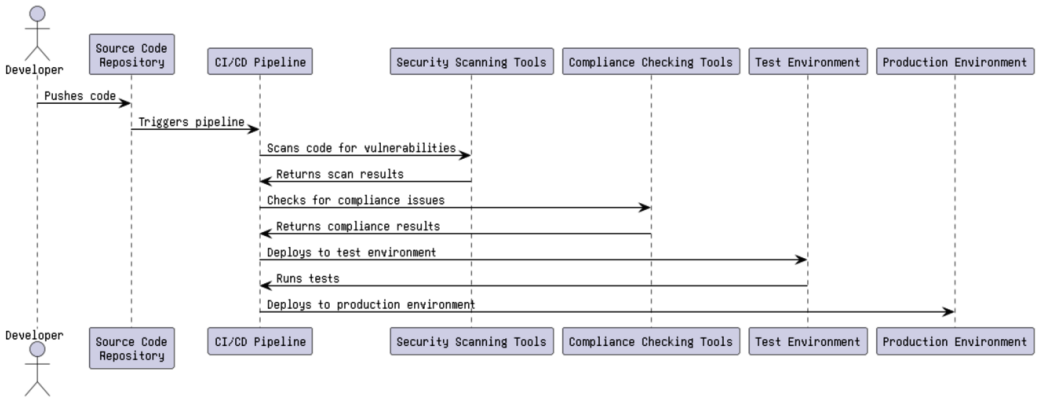

Figure 9.10 – Security and compliance integrated into a CI/CD pipeline

In this diagram, the following is shown:

1. The developer pushes code to the source code repository.
2. This triggers the CI/CD pipeline.
3. The pipeline uses security scanning tools to scan the code for vulnerabilities.
4. The security scanning tools return the scan results to the pipeline.
5. The pipeline uses compliance-checking tools to check for compliance issues.
6. The compliance-checking tools return the compliance results to the pipeline.
7. If the code passes the security scans and compliance checks, the pipeline deploys the code to the test environment and runs tests.
8. If the tests pass, the pipeline deploys the code to the production environment.

This diagram shows how security and compliance checks can be integrated into the CI/CD pipeline to ensure that the code is secure and compliant before it is deployed.

Auditing and attestation round out the security and compliance measures, providing a means to verify and validate the security and compliance status of the CI/CD pipeline. Regular audits and the ability to attest to the compliance of the software being delivered are fundamental to maintaining trust and meeting regulatory requirements.

In summary, the key components for ensuring security and compliance in CI/CD involve a combination of cultural shifts, automated tools, and rigorous processes that work together to protect the software delivery pipeline from potential threats and ensure that the software meets the necessary compliance standards.

In the following section, we are going to have a look at the implementation of CI/CD based on creational design patterns.

Implementing CI/CD pipelines based on creational design patterns

Now, having learned, in the previous sections and subsections, about these creational design patterns, we will look at the implementation part of it, including some examples and scenarios.

Implementing CI/CD pipelines in the cloud using creational design patterns can significantly enhance the automation, efficiency, and scalability of software deployment processes. Creational patterns, which focus on the object creation mechanism, can provide a blueprint for creating instances in a manner suited to the situation. In the context of CI/CD, this could mean dynamically generating pipeline components or environments based on the specific needs of a deployment. For instance, a Factory Method pattern could be used to create different types of automated tests or deployment strategies depending on the target environment. Similarly, a Builder pattern might be employed to construct complex, customizable pipelines that can handle various types of software projects with distinct build and deployment requirements. By leveraging these patterns, teams can create robust CI/CD pipelines that are both flexible and maintainable, ensuring that the software delivery process is as streamlined and error-free as possible. Additionally, understanding and applying these patterns can lead to more predictable and reliable deployments, which is crucial for CI/CD practices.

Implementing effective CI/CD pipelines can enhance the software development process and utilizing creational design patterns can provide a structured approach to managing the creation of objects within these pipelines. Creational design patterns, such as Factory Method, Abstract Factory, Builder, Prototype, and Singleton, offer various ways to construct objects that can handle different stages of a CI/CD pipeline, from code compilation to deployment.

For instance, a Builder pattern could be used to create a complex pipeline by separating the construction process, permitting the same construction process to create different representations. Similarly, a Singleton pattern could ensure that a pipeline has a single instance of a service. This is particularly useful for services that manage state or resources, such as a logging service.

By applying these patterns, developers can create pipelines that are more maintainable, scalable, and easier to understand. Moreover, patterns can help encapsulate the creation logic and promote consistency across the pipeline, which is crucial for the CI/CD process, which relies on predictable and repeatable builds and deployments.

In the next subsection, we will have a closer look at how to implement these patterns in a CI/CD pipeline.

Factory Method implementation

Let us look into some examples of the Factory Method pattern implemented.

As we already explained before, the Factory Method is a pattern that defines an interface for creating objects in a superclass and permits subclasses to change the type of objects that will be created. With a CI/CD pipeline, you could use the Factory Method to define a framework for creating deployment jobs without specifying the exact classes that will be instantiated.

Imagine a scenario where your software needs to be deployed to different environments, such as testing, staging, and production. Each environment requires slightly different deployment steps, configurations, or middleware services. You could define an abstract class called `DeploymentJob` with a method called `createDeploymentSteps()`. This method acts as the Factory Method.

Subclasses of `DeploymentJob` would then override `createDeploymentSteps()` to instantiate environment-specific deployment steps. For example, `TestingDeploymentJob`, `StagingDeploymentJob`, and `ProductionDeploymentJob` could be subclasses that provide the specific steps needed for each environment.

When the CI/CD pipeline runs, it doesn't need to know which type of deployment job it's creating. It just calls the `createDeploymentSteps()` method, which has been overridden by the subclass to provide the correct steps. This way, the pipeline is decoupled from the specific classes that perform the deployment, which makes the system easier to extend and maintain.

Now, you can write this, for instance, in Python code, but using a tool such as Argo CD, it might be better to use a configuration language or **domain-specific language** (**DSL**) supported by your CI/CD tool to define your pipeline and deployment steps. Let's have a sneak peek at how that works.

To give you an example of how this would look, we will use a well-known cloud-native CI/CD tool called Argo CD, which is a continuous delivery tool for Kubernetes that automates application deployments. It works based on GitOps principles, meaning that it syncs the deployment process with the application specifications described in Git repositories. This synchronization is performed at specified intervals. Argo CD comes with a graphical user interface that allows for the easy management and visualization of applications. Moreover, it supports multiple configuration management tools, such as Kustomize, Helm, and Jsonnet, among others.

In this context, you can apply the scenario you described by using different configurations for each environment (testing, staging, and production).

In Argo CD, you can create deployment configurations using Kubernetes manifests along with Kustomize or Helm to manage differences between environments. To define your Argo CD applications, you can organize your repository in a structure similar to this:

```
.
├── base
│   ├── deployment.yaml
│   └── service.yaml
├── overlays
│   ├── dev
│   │   ├── kustomization.yaml
│   │   └── patch.yaml
│   ├── prod
│   │   ├── kustomization.yaml
│   │   └── patch.yaml
│   └── stage
```

```
|       ├── kustomization.yaml
|       └── patch.yaml
└── kustomization.yaml
```

The proposed methodology involves the segregation of the base Kubernetes manifests and environment-specific configurations into the respective directories of `base` and `overlays`. The `kustomization.yaml` file is then created for each environment (`dev`, `stage`, and `prod`), which references the base manifests and applies any necessary patches.

For each environment, an Argo CD application is created by specifying the path to the corresponding overlay directory in the source repository. After syncing the application in Argo CD, the manifests from the specified path are applied to the destination cluster.

This approach enables encapsulation of the differences between each environment into separate configurations, akin to separate classes in the original Python example. The appropriate configuration (similar to the Factory Method pattern) can then be used to create the appropriate deployment steps based on the target environment.

The following is an example of how an Argo CD application can be defined:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: my-app-dev
spec:
  project: default
  source:
    repoURL: https://github.com/my-org/my-repo.git
    targetRevision: HEAD
    path: overlays/dev
  destination:
    server: https://kubernetes.default.svc
    namespace: dev
```

The `path` field in the present instance directs toward the `overlays/dev` directory, which houses the configuration settings tailored to the developmental environment. Similar application definitions, with the path field indicating the corresponding overlay directory, must be created for the stage and production environments.

Now that we've seen this example, let's have a look at the implementation of the Abstract Factory pattern.

Abstract Factory pattern implementation

The Abstract Factory pattern implementation, in the context of cloud computing and CI/CD, can facilitate the management of cloud resources and services across various environments.

In a CI/CD pipeline, the Abstract Factory pattern can help automate the provisioning and deployment of these resources. It allows for the creation of a suite of objects that are required for a deployment, such as virtual machines, containers, storage, and other configurations, tailored to the needs of each stage in the pipeline. By abstracting the creation process, developers can focus on the logic of their applications rather than the intricacies of the cloud environment. This pattern also promotes consistency and reduces the potential for errors during the deployment process, as the factory ensures that each resource is created following the predefined specifications suitable for the target environment.

Moreover, when it comes to scaling or updating the cloud infrastructure, the Abstract Factory pattern provides a systematic approach to handle these changes. It can be extended to accommodate new types of resources or configurations, making the CI/CD process more adaptable and maintainable over time. Overall, the Abstract Factory pattern is a powerful tool in the realm of cloud-based CI/CD, enabling more efficient, reliable, and scalable software delivery workflows.

This pattern allows for avoiding specifying concrete classes when creating families of related objects. It creates objects that are related or dependent on each other. Imagine a scenario where the pipeline needs to support multiple types of source code repositories, such as Git. An abstract factory could provide a uniform interface to interact with these various repository types. For instance, the factory could have methods for common actions such as `commit`, `merge`, and `revert`, which would be implemented differently depending on the repository type. This pattern allows the CI/CD pipeline to remain flexible and extensible, as new repository types can be added without altering the existing pipeline's code structure. Essentially, the Abstract Factory pattern helps in creating an architecture that is adaptable and scalable to the evolving needs of a CI/CD process.

The best way to explain this pattern is to show you the following diagram:

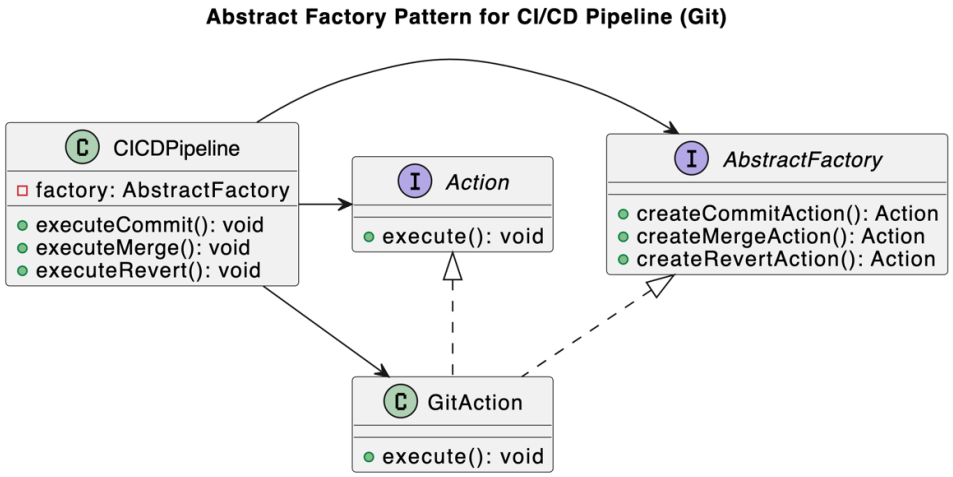

Figure 9.11 – Implementation of the Abstract Factory pattern

The diagram shows the following components:

- `AbstractFactory`:
  - An interface that defines methods for creating different types of actions (such as `commit`, `merge`, and `revert`)
  - Acts as a factory for creating related objects
- `Action`:
  - Another interface representing the common behavior for all actions
  - Contains a single method, `execute()`, which performs the specific action
- `GitAction`:
  - Concrete classes that implement the `Action` interface
  - Each class provides its own implementation of the `execute()` method
  - These represent specific actions related to Git repositories
- `CICDPipeline`:
  - A class that interacts with the abstract factory and executes actions
  - Contains references to the factory and methods such as `executeCommit()`, `executeMerge()`, and `executeRevert()`
  - The implementation depends on the concrete factory used
- Connections:
  - The `CICDPipeline` class interacts with `AbstractFactory` to create instances of `GitAction`
  - The abstract factory ensures that the correct type of action is created based on the context (e.g., the type of repository)

In summary, the Abstract Factory pattern allows the CI/CD pipeline to create different types of actions (for various repository types) without specifying their concrete classes directly. It promotes flexibility and extensibility in handling diverse repositories.

In the following section, we will have a look at the Builder pattern implementation.

Builder pattern implementation

The implementation of the Builder pattern allows for the construction of complex objects step by step. It is particularly useful in scenarios where an object needs to be created with many possible configurations, and it can be implemented in a CI/CD pipeline to ensure that software builds are consistent and automated. In terms of CI/CD, the Builder pattern can be used to define a process for compiling code, running tests, and deploying applications in a controlled sequence of steps, much like constructing an object piece by piece.

You can see this construct in the following figure:

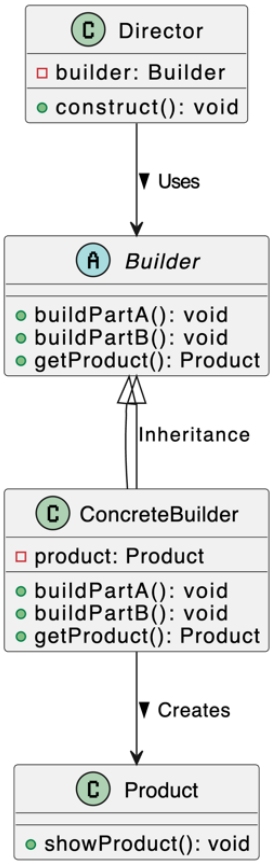

Figure 9.12 – Implementation of the Builder pattern

Let's break the components of this figure down:

- The `Director` class is responsible for executing the building steps in a particular sequence. It's only responsible for executing the building steps and not for producing the product.

- The `Builder` class creates parts of a `Product` object using an abstract interface.
- `ConcreteBuilder` implements the `Builder` interface to construct and assemble parts of the product. It keeps track of the representation it creates and provides an interface to retrieve the final product.

In a CI/CD pipeline scenario, `Product` could represent a software build, `Builder` could represent the steps needed to create the build (such as compiling code and running tests), and `Director` could represent the automation server controlling the build process.

Next, we'll dive into some implementations of these patterns.

Prototype pattern implementation

The Prototype pattern can be particularly useful in a CI/CD pipeline by enabling the creation of expensive objects to create from scratch. In the context of CI/CD, this pattern can help manage configurations and dependencies required for deploying software artifacts. By using prototypes or pre-configured templates, teams can quickly spin up or tear down environments, which is essential for testing in CI/CD workflows. This ensures that each integration and deployment cycle is consistent and reliable, reducing the risk of errors that can occur with manual setups. Implementing the Prototype pattern within CI/CD can lead to more efficient, error-free releases and a smoother development process overall.

Let's have a look at a graphical representation of the Prototype pattern:

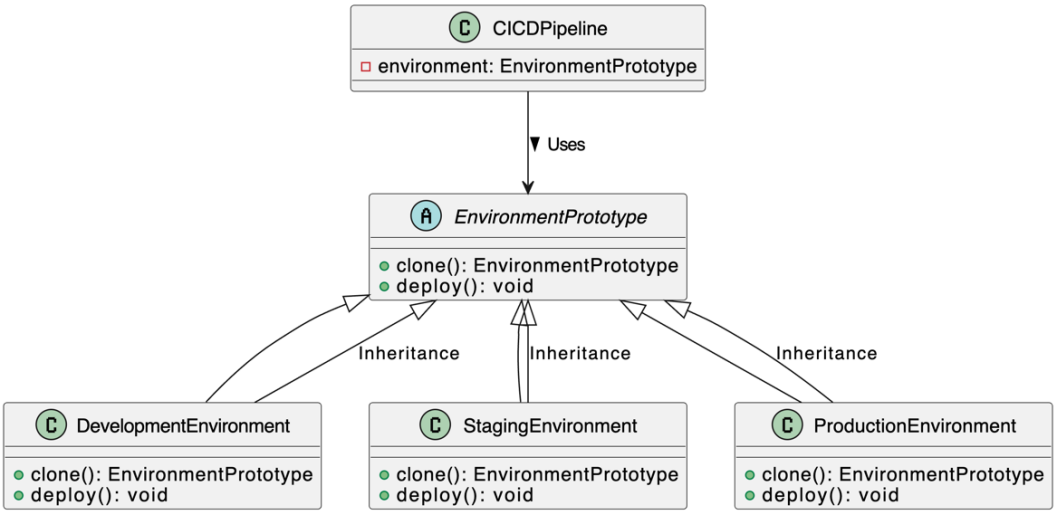

Figure 9.13 – Implementation of the Prototype pattern

What you see in this diagram is the following:

- The `EnvironmentPrototype` class declares an interface for cloning itself and deploying the application.
- The `DevelopmentEnvironment`, `StagingEnvironment`, and `ProductionEnvironment` classes implement the `EnvironmentPrototype` interface. Each of these classes represents a specific configuration of an environment.
- The `CICDPipeline` class represents the CI/CD pipeline, which clones the appropriate environment prototype and deploys the application.

In a real-world scenario, the deploy method in each environment class would handle the specifics of deploying the application to that environment. This could include tasks such as setting environment variables, configuring databases, and starting servers.

The next implementation to look at is the Singleton pattern.

Singleton pattern implementation

The Singleton pattern ensures a class has a single instance with a global point of access to it. In the context of CI/CD, implementing the Singleton pattern can be particularly useful for managing shared resources such as configuration settings, connection pools, or logging. For instance, during the CI process, a Singleton can ensure that only one example of a build tool or a test suite is running at a time, preventing conflicts and ensuring consistency. Similarly, a Singleton might manage deployment configurations in CD, ensuring that all deployment tasks reference a single source of truth. It's crucial to implement the Singleton pattern in a thread-safe manner to avoid issues in a multi-threaded environment, which is common in CI/CD pipelines. This often involves synchronization mechanisms or lazy initialization techniques to ensure that the Singleton instance is created safely without unnecessary resource consumption. Proper implementation of the Singleton pattern in CI/CD pipelines can lead to more efficient resource utilization, better management of shared resources, and a reduced likelihood of errors during the build and deployment processes.

Now, we have seen a simple example of a Singleton pattern in this section, but let's have a look at another example where the Singleton pattern can be applied.

Imagine you're working on a large-scale web application that needs to be tested across multiple environments before it's deployed to production. These environments could include development, staging, and production, each with its own unique configuration.

Here's a simplified diagram representing this scenario:

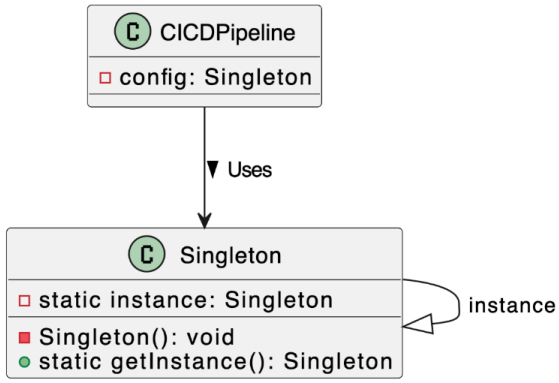

Figure 9.14 – Implementation of the Singleton pattern

The diagram shows you the following:

- The `Singleton` class represents a shared resource, such as a configuration setting, a connection pool, or a logging service. The `getInstance()` method ensures that only one instance of this resource is used throughout the CI/CD pipeline, preventing conflicts and ensuring consistency.
- The `CICDPipeline` class represents the CI/CD pipeline that uses the Singleton instance.

In a real-world scenario, Singleton might manage deployment configurations in CD, ensuring that all deployment tasks reference a single source of truth. It's crucial to implement the Singleton pattern in a thread-safe manner to avoid issues in a multi-threaded environment, which is common in CI/CD pipelines. This often involves synchronization mechanisms or lazy initialization techniques to ensure that the Singleton instance is created safely without unnecessary resource consumption. Proper implementation of the Singleton pattern in CI/CD pipelines can lead to more efficient resource utilization, better management of shared resources, and a reduced likelihood of errors during the build and deployment processes.

Summarizing the implementation of creational design patterns

In summary, integrating creational design patterns into CI/CD pipelines can lead to more robust and flexible automation processes. These are essential for CI and continuous delivery in today's software development environments. Implementing creational design patterns, which offer a structured approach to creating objects within the pipelines, can make CI/CD pipelines more efficient and reliable.

By keeping these patterns in mind, developers can create pipelines that are more maintainable, scalable, and easier to understand. These patterns help to encapsulate the creation logic and promote consistency throughout the pipeline. But as said before, they are conceptual and need to be translated to real-life scenarios using the tools and techniques that exist in the cloud-native world.

Last in this chapter, let's have a look at the scaling and optimization of CI/CD pipelines, including the prerequisites of it.

Prerequisites of scaling and optimization

Scaling and optimization in the context of CI/CD pipelines when using creational design patterns involve several key prerequisites. Firstly, a thorough understanding of the specific creational pattern being implemented is crucial, as each pattern has its own implications for instantiation and object creation processes that can affect scalability and performance. For instance, the Singleton pattern may introduce bottlenecks due to its nature of restricting object creation to a single instance.

Moreover, it's essential to ensure that the system's architecture is designed with scalability in mind from the outset. This includes using microservices architectures or serverless computing platforms to allow for the independent scaling of different components of the application. Containerization and orchestration tools can also play a meaningful role in managing and scaling containerized applications efficiently. That translates into the following key aspects:

- In terms of optimization, leveraging AI-powered tools can aid in code review, quality assurance, and performance optimization, ensuring that the CI/CD pipeline runs smoothly and efficiently. Tools such as DeepCode and Kite can detect bugs and security vulnerabilities, while machine learning-based tools such as Akamas can optimize software configurations autonomously.
- Another important aspect is the standardization and automation of the development pipeline, which facilitates the management of the pipeline and ensures the quality of the code being delivered. This standardization can be achieved through IaC tools such as Terraform (or OpenTofu if you want to step away from the commercial tools), Ansible, or Chef, which enable rapid, consistent, and repeatable deployments.
- Scaling and optimization in creational design patterns are essential for ensuring that cloud-native applications can efficiently manage resources and maintain performance during CI/CD processes. Scaling and optimizing a CI/CD pipeline requires careful planning and execution. Here are some of the considerations around scaling and optimizing a CI/CD pipeline:
    - Understand the current state and identify issues and bottlenecks
    - Use microservices architecture
    - Automate and standardize processes
    - Optimize resources and costs
    - Implement feedback loops for continuous improvement
    - Choose the right tools based on compatibility, scalability, performance, security, and usability
    - Avoid single points of failure by proactively scaling infrastructure

- Problem-solving and optimization involve a continuous cycle of integrating and delivering code changes. This process requires a robust setup that can handle the dynamic nature of cloud-native applications, which often rely on microservices and containerization. Effective CI/CD pipelines must accommodate rapid scaling and frequent deployments, all while minimizing downtime and resource wastage. Techniques for identifying and resolving issues include implementing automated testing to catch bugs early, using monitoring tools to track performance bottlenecks, and applying IaC to streamline environment provisioning.
- Optimization in CI/CD also entails refining the pipeline to reduce build times, improve testing accuracy, and ensure security compliance. This can be achieved by parallelizing tasks where possible, caching dependencies, and using container orchestration platforms such as Kubernetes to manage the deployment of services efficiently. Additionally, cloud-native applications benefit from a microservices architecture, which allows individual components to be scaled independently based on demand, resulting in adequate resource utilization and performance.
- Lastly, observability and monitoring are critical for scaling and optimization. Implementing AI-powered monitoring and logging tools provides insights into the system's performance and helps in anticipating potential failures in the development process or the software itself, allowing for proactive problem-solving and resource optimization.

In summary, the prerequisites for scaling and optimization in creational design patterns within the CI/CD context for cloud-native applications involve understanding the design patterns that facilitate efficient object creation, implementing a CI/CD pipeline that supports rapid and reliable delivery, and employing strategies to monitor and optimize resource usage. By focusing on these areas, developers can ensure that their applications are scalable, performant, and resilient in the face of changing demands and complex cloud environments.

Now that we have completed the last topic of this chapter, let's take a look at a brief summary of what you've learned.

Summary

This chapter offered a guide to integrating CI/CD with cloud technologies. The guide emphasized the importance of automating the software development life cycle to enhance efficiency and reliability. Adopting containerization, microservices, and serverless computing is crucial for leveraging cloud infrastructure, ensuring scalability and resilience. These elements form the backbone of the creational CI/CD design patterns, which prioritize principles such as scalability and resilience. Security and compliance are also integral, ensuring that the implemented systems are robust against threats while adhering to regulatory standards. Implementing CI/CD pipelines based on these design patterns is a strategic approach to accelerate development cycles, minimize downtime, and maintain agility. This ensures that applications are not only optimized for the cloud's dynamic nature but are also future-proofed against evolving technological demands. Finally, understanding the prerequisites of scaling and optimization is essential for teams to effectively apply these patterns and achieve a streamlined development process with optimal outcomes. This chapter serves as a call to action for software

development teams to embrace the power of creational CI/CD design patterns and elevate their software development process to new heights.

Now that we've concluded this chapter, another interesting topic comes up in the next chapter, that is, understanding deployment strategies and creational CI/CD with cloud providers. Have fun!

10

Understanding Deployment Strategies – Creational CI/CD with Cloud Providers

In this chapter, we will explore cloud-native architectural and deployment strategies, delving into key concepts and practices optimized for cloud environments. These concepts will give you insights into containerization, microservices, and serverless computing, along with their practical applications. Additionally, this chapter will highlight leading cloud providers, showcasing how various platforms support cloud-native approaches. At the end of this chapter, we will be armed with a deep understanding of modern cloud-native paradigms and how to harness them effectively for scalable and resilient software deployments.

The following topics will be covered in this chapter:

- Creational design pattern used in CI/CD
- Cloud native development and its influence on CI/CD patterns
- Cloud provider offerings
- Cloud offerings for CI/CD toolset
- Deployment strategies
- Team topologies
- Practical examples

In this chapter, we will discuss practical applications of the design patterns we described in *Chapters 5* to *9*, and we will implement them in a cloud environment – specifically, AWS. We will go deeper into different architectures and their specific configurations required in the scope of CI/CD. We will also check how to understand different creational CI/CD patterns in the scope of different infrastructure/application architectures.

Creational design pattern used in CI/CD

In *Chapter 9*, we discussed the theory and implementation of creational patterns. We also mentioned IaC and configuration, so here, we will focus on this aspect through the lens of creational patterns. There is another reason, too. In this chapter, we will navigate through cloud providers and while the application deployed on any cloud will be relatively similar, the underlying infrastructure, configuration, and so on will be significantly different.

In *Chapter 4*, we presented various approaches that are the foundation of creational design patterns mentioned in the previous chapter. To remind you, these patterns are as follows: the **factory method**, **abstract method**, **singleton**, **prototype**, and **builder** patterns. Repo design, dependencies management, modularity (*Chapter 4*), and so on are key elements of consistency, scalability, efficiency, and rapid prototyping (*Chapter 9*).

In this chapter, we will combine the knowledge we have learned and apply it to real-world scenarios and problems. We will see how proper design can drive organizational growth and enforce processes, quality, and responsibilities. By understanding creational design patterns at a meta level, we can apply them to various elements. In fact, when discussing application architecture and CI/CD architecture, we find common ground as both use similar architectural patterns.

Before we continue, we first need to clear out an element called **landing zone**, which we will do in the following subsection.

Landing zones

The element that is not covered by this book is the concept of a landing zone. The definition of landing zone provided by AWS (`https://docs.aws.amazon.com/prescriptive-guidance/latest/migration-aws-environment/understanding-landing-zones.html`) determines the following:

> *"A landing zone is a well-architected, multi-account AWS environment that is scalable and secure. This is a starting point from which your organization can quickly launch and deploy workloads and applications with confidence in your security and infrastructure environment."*

Why do we mention this element? That's because it determines the way in which we design CI/CD in terms of the scope of architecture and responsibilities. We will say more about this in the *Team topologies* section of this chapter.

The landing zone approach not only determines but also enforces some decisions and patterns. One of them is the way in which we design workflows in terms of the scope of creational patterns.

Code repositories as a singleton

In *Chapter 4*, we deeply discussed monorepos and polyrepos. For the example ahead, we have decided to utilize the monorepo pattern, so we will create the repo and its structure for multiple microservices in one repository.

Going with a monorepo per service of the application means that each repository contains microservices for one service only. Sometimes, it might be one microservice, and sometimes it might be many. What we want to achieve is a simplistic and standardized way of working on the code.

We will now move to the next section for designing the code repository, adhering to best practices for separating services and functionalities.

Repository structure example

Let's check the structure of the repository:

```
 .
|- .github
| |- workflows
|- Infrastructure
| |- Common
| |- Service1
| |- Service2
| |- Service3
|- Configuration
| |- Common
| |- Service1
| |- Service2
| |- Service3
|- Application
| |- Service1
|…
```

In the following figure, we see how templating (and thus, the singleton approach) allows us to standardize the creation of the project's repositories.

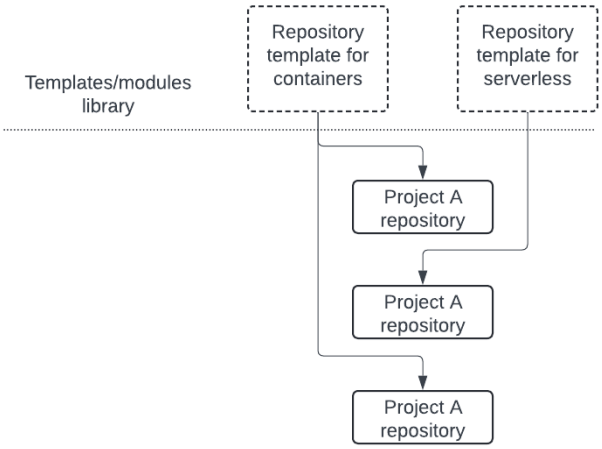

Figure 10.1 – Repository standardization and templating

The reasons to use templating even in repository structures are quite simple. This way, we can ensure the standardization of code structure. This enables us to use standardization further, including in pipelines, builds, and so on, without analyzing the code structure, directories, locations, scripts, and so on. The template repository can contain predefined scripts, data, and so on. Therefore, there is no need to write those predefined artifacts again. What should be the principle, though, this template should be created with modularization in mind. This means that we shouldn't provide the scripts' code, but rather the location of the scripts.

Singleton creational pattern in a microservices world

We have already created a repo, and the structure is standardized. We delivered a code to distinct directories; now, it is time to build our containers.

Now, you may ask, how do you create the standardized Dockerfile? Although you can try to standardize the Dockerfile itself, you should instead focus on a standardized container build from Dockerfile.

Dockerfile can be different for every service. Of course, for any programming language, the code will be more or less similar. However, there will be differences, as the components play different roles.

The build process, however, might be standardized. In fact, in the simplest case, it will be something similar to the following:

```
$ docker build -t name:tag -t name:tag -t tag .
```

Of course, the command only shows the build process, but your build block can contain more actions. In the interest of keeping the agreed standards and enforcing them, you can encapsulate it as a standardized block, delivered separately from another place (such as a registry of standardized components).

The same goes for other actions, such as pushing the image to a repository, as well as running tests, running security tests, generating SBOM, and many more.

Singleton pattern in pipelines

Let's go through the figures that follow. The first one presents the extension of *Figure 10.1*. We add build pipelines in the projects, using templating (or modules).

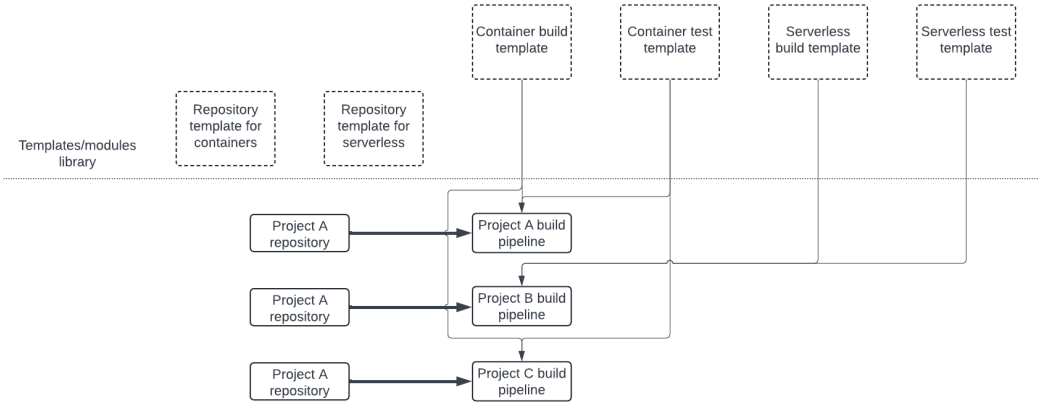

Figure 10.2 – First extension using templating – build pipeline

For clarity, we will not show the previous relations between templates and components.

The next stage adds the deployment templating:

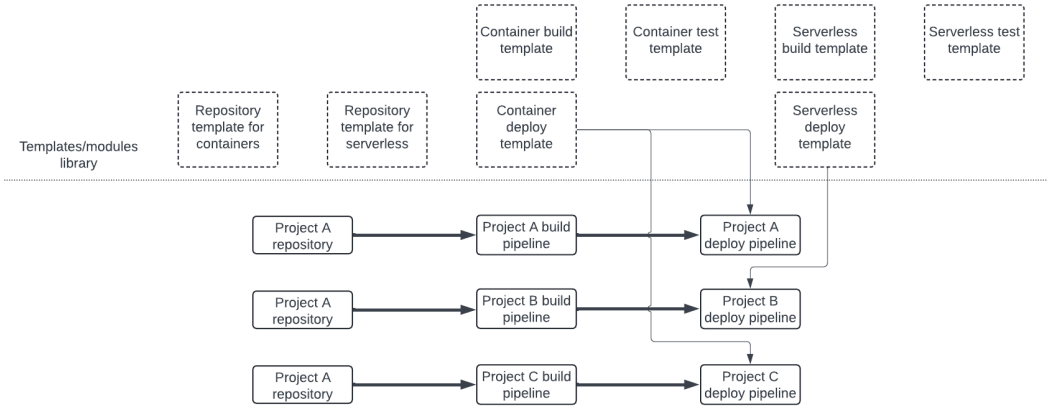

Figure 10.3 – Further extension with deployment templates

Finally, we add tests:

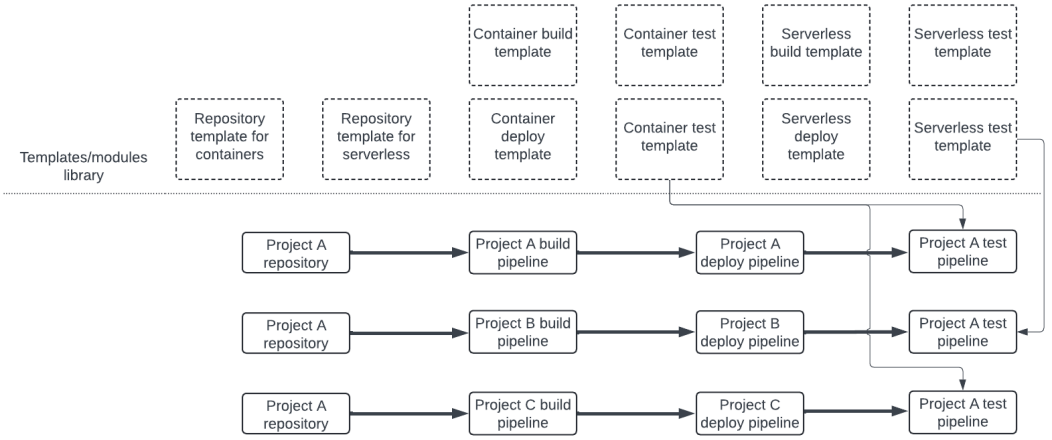

Figure 10.4 – Completed templating

Of course, at this stage, we have more templates than pipelines. However, let's remind ourselves why we want to standardize the approach. The main reasons are to achieve consistency, scalability, efficiency, and rapid prototyping. When the number of pipelines grows, the benefit will be more visible. It is there already, however. Instead of making changes in all separated repos multiple times, you can modify the code in the module only once.

We know that we have shown you this approach before. However, we believe that it is crucial for proper DevOps adoption to remember the importance of these patterns.

Singletons in clouds

Translating the singleton creational pattern into cloud environments is easy. Especially with infrastructure, we can enforce standardization with a singleton approach, using one IaC tool (such as Terraform) and defining modules with it (for example, a module to create a network).

Let's look at the example that follows:

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.8.1"
}
```

This way, we can treat the module as singleton and reuse it multiple times in many projects.

So far in this chapter, we have discussed three concepts of singletons in CI/CD and IaC. We need this translation to understand how to deal with cloud services through automation.

Cloud-native development and its influence on CI/CD patterns

As we have established pattern examples, we are ready to dive deeper into CI/CD for the cloud-native world. However, first, let's explain what the cloud-native approach is.

Cloud-native is an approach to building, deploying, and maintaining software in cloud environments, using all necessary and available services provided by cloud vendors.

The first element of influence of the cloud-native approach in the context of CI/CD is the foundation of this pattern. The cloud-native approach comes with three benefits:

- **Increased efficiency**: This comes from delivering faster and smaller changes to applications. This principle stimulates the microservices approach. Please bear in mind that we do not separate containerization and serverless here. In principle, both approaches decouple applications.
- **Increased availability**: This comes from building resilient, small, and highly available components of applications.
- **Reduced cost**: This comes from decoupling the application. By doing this, we can scale the elements that should be scaled.

The list of benefits is based on the AWS documentation (`https://aws.amazon.com/what-is/cloud-native/`). In fact, many sources might give different benefits, but we find these to be the most important and common ones.

AWS also defines what the principles of cloud-native architecture are. The most important ones are immutable infrastructure, microservices, and API. There are more, but from the perspective of CI/CD and its impact on pipelines, these three are the most important. We add **Infrastructure as Code** (**IaC**) to this list as a separate pillar, yet one that is also very important in terms of working in a cloud-native way.

Let's depict these pillars and check how pipelines are influenced by this architecture.

Immutable infrastructure

We will start with this principle. However, this one is not always fully implemented, especially if we select specific approaches, such as serverless.

Immutable infrastructure defines that when infrastructure is created, it is not changed. In other words, if we deploy a server with an application on it, we do not deploy a new version of that application on this server. Instead, we deploy the whole server with a new application version on it.

This approach allows us to have the option to bring back the previous server if a new one has some issues. However, it is a more costly and time-consuming process.

Immutable infrastructure, or immutability in this matter, should be treated as a concept that is part of the process and delivery area.

Microservices

The main component of the cloud-native approach, microservices are small, independent components that perform a specific function of the application. Thanks to their small size and minimal complexity, the development, testing, and delivery processes are fast and simple.

Microservices as the element of architecture and workload

Here, we can combine the concept of immutability with microservices. We combine the approach to processes with the approach to architecture. For example, we never update an application in a container – we simply build a new one. This means that our CI/CD must acknowledge this combination and be prepared in the expected way.

API

Microservices architecture enforces changes in communication between services. We can list many design patterns here, but from a cloud-native perspective, the preferred way is through API. API is a gate that brings the loosely coupled microservices together. With the API definition and API contracts, the CI/CD process must recognize these gates, especially in the test phases:

- **Unit tests phase**: Similarly to the rest of the code, the code of API should be the subject of unit tests.
- **Contract tests in the functional tests phase**: In this step, we test whether a defined contract is an exact match to a coded one.
- **Integration tests phase**: Any API's integration should be tested. All functions of an API can be tested in this phase; we don't really care whether the connection is made to or from a mocked component or a real one.
- **End-to-end test phase**: This is similar to the integration phase, except that we should use as few mocked systems as possible.
- **Performance and security tests**: These are distinct elements that require specific approaches. For performance testing, we need to determine how much traffic our API can handle. For security testing, we need to assess how secure our API is against common threats, such as SQL injection attacks.

IaC

We have added this element to the list, although the cloud-native definition doesn't say anything about IaC directly. However, IaC is an integral part of the cloud-native approach as quality, reliability, and short time of delivery are some of the benefits of IaC.

Especially for cloud-based solutions, IaC plays a very important role. Pulumi (`https://www.pulumi.com/whitepapers/delivering-cloud-native-infrastructure-as-code/`) mentioned the following:

"The shift from Infrastructure at Rest, to Infrastructure in Motion."

This shows the main mindset shift in the cloud-native approach in terms of infrastructure.

Cloud infrastructure is in continuous motion. Especially with proper scaling, and the use of SaaS services, such as serverless, the resources are created or triggered when needed. We do not define the number of instances of our application. We define the thresholds for scaling, or the triggers for executions.

This leads to a very important conclusion. Delivery in the cloud-native approach is not done on the defined servers. It is done on the defined service. This means that CI/CD does not control how many nodes, servers, clusters, or deployments must be done or how many servers must be created. This is controlled by infrastructure and application orchestrators. CI/CD is responsible for delivering applications or infrastructure to the systems here, as many times (instances) as are defined by the orchestrator.

Cloud provider offerings

Each cloud provider has its own set of services and approaches, both to workloads and delivery. In this section, we will explore a few concepts that must be understood to properly define delivery strategies for cloud-native applications.

API-driven communication for microservices

In this example, we simplify the architecture to show the high-level concept.

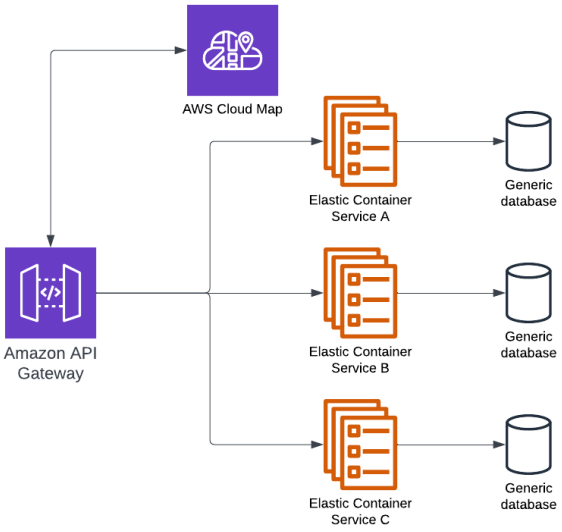

Figure 10.5 – Simplistic example of AWS-based solution for API-driven microservices

The preceding example shows a very simplistic scenario, where we have three different microservices (**A**, **B**, and **C**) behind one API service (in AWS, it is called AWS API Gateway). Elastic Container Service is responsible for scaling the containers (like HorizontalPodAutoscaler in Kubernetes). Finally, at the end, we have some databases.

The interesting things that we see in this diagram are happening in this example, where we use AWS Cloud Map, the cloud resource discovery service. AWS API Gateway needs to know exactly where to send the request. Cloud Map assures exactly that and shows what the IPs of the containers in the service are to send the requests there.

Just to complete the picture, very often, we can see the architecture where we see load balancers, especially AWS Application Load Balancers, which operate on layer 7 of the OSI model (for example, using the HTTPS protocol). In the case of this example and with the implementation of AWS Cloud Map, there is no need to implement a load balancer that has an impact on costs and performance.

Let's think about the CI/CD approach here. There are many scenarios to consider, such as the following:

- Using monorepo for all services
- Extracting API and Cloud Map definitions outside applications' repos
- Multi-repo, in which every single microservice is stored in separate repositories and infrastructure is also separated

How can we select the proper approach? We need to consider a few elements:

- Is this part of a closed component of the system?
- Is the API only an entry point to the microservices defined on the diagram?
- Is Cloud Map only used by these microservices?
- Can two microservices use the same database?
- How can we deploy the changes to this setup?
- How can we test the setup? Please consider the testing pyramid (or diamond) here and plan your test strategy accordingly.

If you answered *yes* to the first four questions, it means that the natural choice will be monorepo, with infrastructure and configurations inside. With this approach, you ensure the separation of components of the application, but you also allow the process to effectively provide the updates using the complete testing scenarios.

API-driven communication for serverless

In principle, the serverless architecture might be seen as very similar to containers. However, in recent years, we have seen more and more dedicated services, which makes the serverless more robust, functional, and effective, as well as different from classic microservices.

The most significant difference is related to the code composition. The serverless approach was always called serverless functions. This term shows the difference. While the microservice is responsible for one service, or one functionality, the serverless function is responsible for one function. The code of serverless functions should be small, effective, and created strictly for serverless. As we go through AWS examples, we need to remember that AWS Lambda (which is an AWS service for serverless compute) expects a specific code structure. Serverless is called *serverless* for a reason. The vendor delivers the whole infrastructure, including compute power, scaling capabilities, monitoring functionality, and so on. For the developers, the vendor only leaves one responsibility: delivering properly designed code. Let's check the JavaScript code here:

```
export const handler = async (event) => {
  const response = {
    statusCode: 200,
    body: JSON.stringify('Hello, world!'),
  };
  return response;
};
```

The second example here shows the same behavior, but the code is written using Python:

```
import json
def handler(event, context):
  return {
    'statusCode': 200,
    'body': json.dumps('Hello, world!')
  }
```

Although these languages have different syntaxes, we can easily spot a common element. It is a function called `handler`. This is an element that AWS Lambda needs to be able to execute the provided code.

As this is not a book about serverless, we only have to understand one more element of the serverless approach. Let's analyze the following diagram:

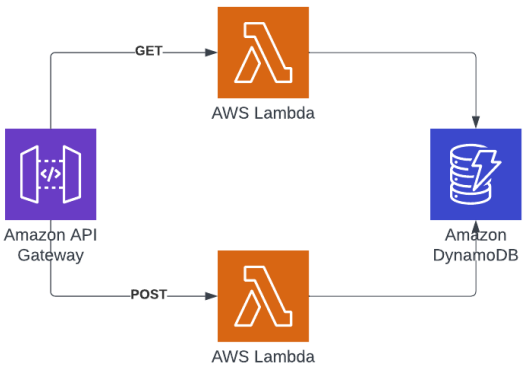

Figure 10.6 – Simple example of serverless architecture

In the example, we see two serverless functions that are triggered by API Gateway and, in the end, connected to one database. However, the difference here is significant compared to the microservices example.

We said before that serverless functions are exactly that – functions. Therefore, we construct the infrastructure differently. As we see in the preceding diagram, there are two lambda functions, but the fact is that the functions are separated by HTTP methods.

If you're working with CI/CD in the context of serverless, you must consider a few elements:

- Testing is harder than in any other architecture. This is caused by the fact that the code needs a special function to be executed.
- Decoupling of the application is deeper, which leads to more connection patterns.
- This decoupling also makes the construction of CI/CD harder to properly establish.

The debate (`https://securityboulevard.com/2020/08/mono-or-multi-repository-a-dilemma-in-the-serverless-world/`; you can read more about it using the links in the *Further reading* section) about whether mono- or polyrepo is better began among experts many years ago and has still not ended. That's for good reasons, as both approaches have some advantages over the other.

To select the correct approach, you have to answer this set of questions:

- How many functions will be created in the service?
- Do you use a serverless framework, such as serverless.com (`https://www.serverless.com/`), serverless.tf (`https://serverless.tf/`), or AWS SAM (`https://aws.amazon.com/serverless/sam/`)?
- Do you want to incorporate in the project all required elements, such as databases, queues, storage, or APIs?
- How decoupled from other services is this service?

This set of questions is just the initial one. You should create a list related to your project.

The hybrid approach to CI/CD seems to be the optimal one. Hybrid, in this case, means that we use polyrepo as the solution for the whole application and monorepos for services with corresponding resources, such as storage, databases, APIs, queues, and so on.

The element that enables this approach is the way in which we can construct the deployment approach using the mentioned frameworks. For example, in AWS SAM, we can build the configuration for many serverless functions, APIs, roles and policies, storages, triggers, and so on. This strongly drives the approach to monorepos.

Let's take a look at the example of the AWS SAM template in the GitHub repository of this book in the `Chapter10` folder, in the file named `SAM-template-for-monorepo-app.yml` (`https://github.com/PacktPublishing/CI-CD-Design-Patterns`).

The rather long example in the repository shows the configuration of the following:

- Two AWS Lambda functions
- AWS DynamoDB table
- AWS API Gateway
- AWS S3 Bucket
- One layer that is used by both functions

This setup encapsulates all the resources needed for the service to run autonomously in any application setup.

In the *Practical examples* section of this chapter, we will discuss the CI/CD for AWS SAM templates.

Cloud offerings for CI/CD toolset

Each cloud vendor has its own approach to CI/CD and provides some tooling for it. In this section, we discuss two different approaches, one from Microsoft and the second from AWS. GCP and other vendors also are present in this field. However, this book is not about discussing tools, but about how to use tools for CI/CD design.

Microsoft Azure – Azure DevOps

Microsoft offers a fully independent CI/CD platform called **Azure DevOps** (**ADO**). Although the name can suggest deep integration with Azure, or even the service being in Azure cloud, it is an independent tool. It offers the full CI/CD process, including project management (Azure Boards). We can of course discuss the quality and usefulness of it, but the fact is that Azure DevOps offers this tool.

Azure DevOps also incorporates testing capabilities (Azure Test Plans) and artifact storage capabilities (Azure Artifacts). Obviously, the most obvious CI/CD capabilities are covered by repositories (Azure Repos) and CI and CD (Azure pipelines). In addition, we have advanced security, as well as the possibility to create a registry for programming languages such as JS, dotnet, Java, or Python. Additionally, we can create a registry for custom artifacts.

Azure DevOps should be treated as a standalone SaaS CI/CD tool. It can easily be integrated with any platform and is a fully functional, very powerful tool to use.

AWS CodePipeline

The AWS tool for CI/CD is much simpler than the one from Azure. It is the collection of tools in one service, called **AWS CodePipeline**. We can find repositories there (AWS CodeCommit), as well as artifact storage (AWS Code Artifacts), where we can build registries for Java, JS, dotnet, Python, Ruby, Rust, and Swift.

Of course, we have a CI tool (AWS CodeBuild) and a CD tool (AWS CodeDeploy). It is all orchestrated through AWS CodePipeline.

AWS solution is strongly incorporated into the AWS ecosystem. All security aspects, encryption, access, and so on are based on AWS services, such as CodeGuru, KMS, IAM, and so on. It might be very powerful in many AWS workloads, but it is well-known that experts are not big fans of this tool.

It is time to switch our focus to the ways in which we can deploy the code. We learned the patterns, tools, and (some) architecture specifics. Now, we will see how these elements influence the proper selection of delivery and deployment patterns.

Deployment strategies

Although deployment strategies are very well described and are not exclusive to clouds, it is the cloud-native environment that enables most of these. This is the reason why we define them here again; we will also look at their potential in terms of cloud environments.

In this section, we will go through three of the most popular strategies, and we will start with the **blue-green deployment** strategy.

Blue-green deployment strategy

Blue-green is probably the most well-known, but also the most mistakenly understood, strategy in terms what its subject. In other words, people often misunderstand what part of the application and infrastructure is part of the blue-green deployment. That is why we will start with diagrams that present the strategy at a high-level view. Please follow these two diagrams and observe the change.

The first diagram shows the used environment (v.1) and the prepared and ready environment (v.2). The whole point here is that the prepared environment is a full copy, or the mirror of the running environment.

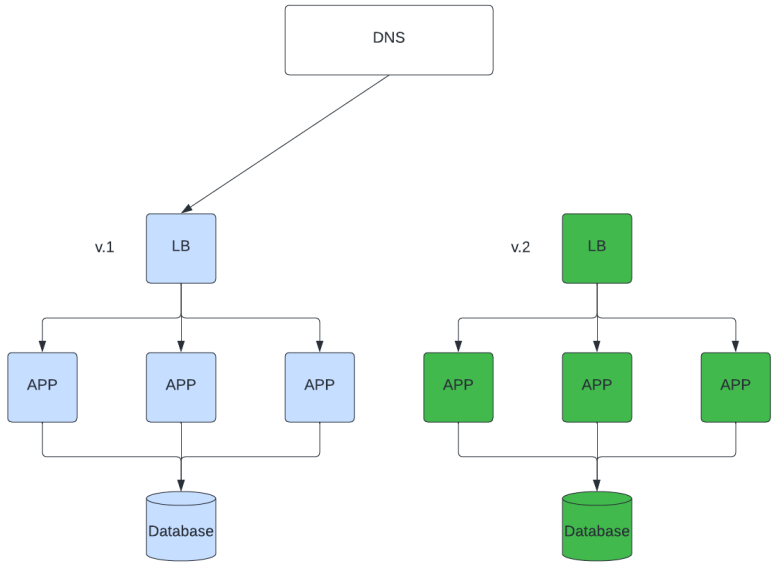

Figure 10.7 – Blue-green before the switch

When the deployment is performed, it really changes the top-level DNS entries and redirects the traffic to standby, or prepared environment, as in the diagram that follows:

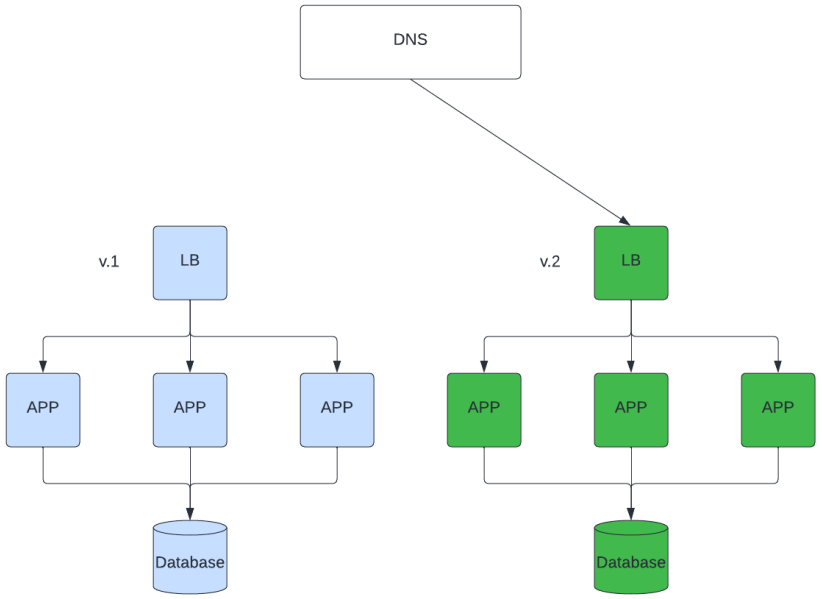

Figure 10.8 – Blue-green after deployment

This approach has some significant advantages, but at the same time, it is very costly. Pros and cons are presented in the table that follows:

Advantages	**Disadvantages**
It comes with a very easy rollback – simply switch on the DNS level.	Complexity: This approach requires us to create and manage the mirror of existing infrastructure and applications.
It creates the possibility to test every aspect of the new version.	Costs: Two separate but identical systems are twice as expensive. It is simple math. However, we need to keep maintenance, people's time, and so on in mind.
It's the perfect solution for a Release approach (where the multiple changes are packed into one release).	Databases: As you have to keep two separate databases, there is another step to perform during the switch. We need to ensure the consistency between data in both databases.
It brings reduced disruption – the production *leg* is not touched until the last moment.	Administration: Maintaining two environments means doubling the effort required for administration such as security management. Both environments need to be kept secure, patched, and compliant.

Table 10.1 – Advantages and disadvantages of blue-green deployment strategy in the delivery process in the organization

Let's define additional pros and cons of this strategy, this time exclusively for cloud-based environments:

Advantages	**Disadvantages**
It's easy to manage with IaC.	One of the biggest advantages of cloud computing is the ability to run resources on demand. However, blue-green deployments do not fit neatly into this definition.
Many tools are designed to have an option for blue-green deployment.	This approach is the *heaviest* for CI/CD, IaC, and configuration management tools.
It allows for seamless rollbacks, minimizing downtime and reducing the risk of deployment failures.	This has the complexity of managing and synchronizing data between the two environments in the cloud.

Table 10.2 – Advantages and disadvantages of blue-green strategy performed in a cloud environment

Blue-green modification – blue-violet

As we have established, the blue-green approach is heavy in terms of costs and complexity. This is the reason why modifications to it were developed, called blue-violet, red-black, and a few more. In this section, we will look at blue-violet.

> **Fun fact**
>
> Blue always stays in this modification's name. However, violet might be changed to a different color. The most common seems to be blue-yellow or blue-red. However, this doesn't affect anything in the pattern; it is just the way some engineers call it.

Let's look at the diagrams that follow. The following figure shows the blue-violet approach before deployment:

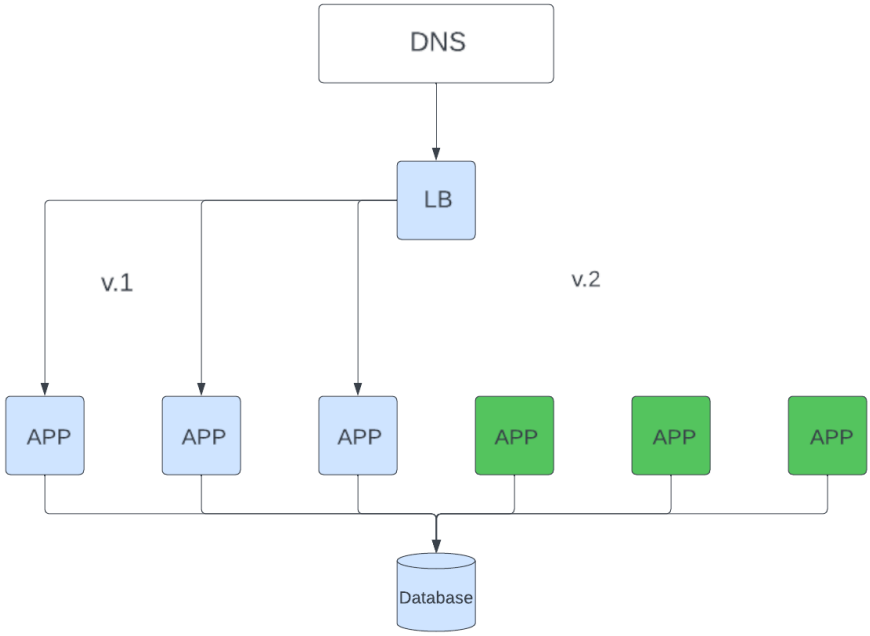

Figure 10.9 – Blue-violet before deployment

Now, let's see how it looks after deployment.

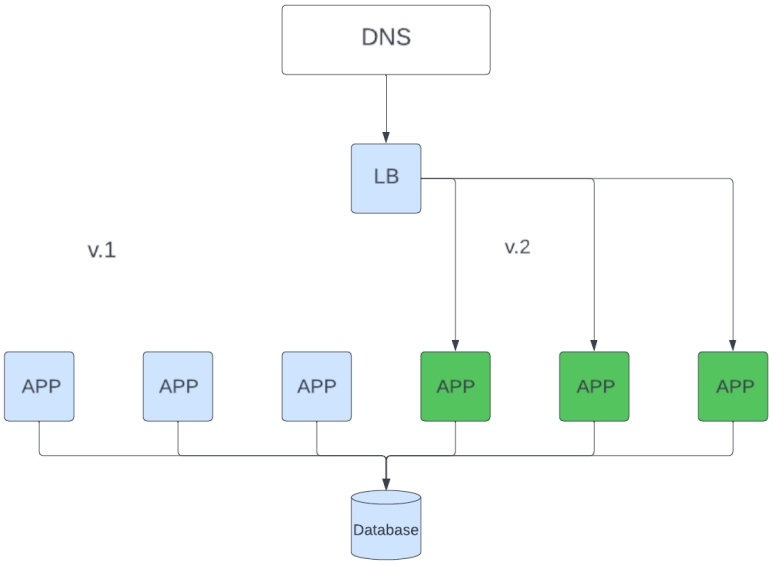

Figure 10.10 – Blue-violet after deployment

It is clear that in this modification, we care about part of the infrastructure and application. What's more, we apply changes to the part between the load balancer and database, so these two stay untouched (from the perspective of the creation of new resources).

What are the benefits of this approach? Let's see a list:

- **Less complexity locally**: In theory, this approach is less complicated than normal blue-green, as we only care about part of the system.
- **Faster than blue-green**: It is faster, as it touches only the part that belongs to one team (in most cases). This means that no additional scheduling and communication is needed.

However, this approach has also issues:

- **Bigger complexity globally**: Imagine a very complex system, where every team or part of the application is deployed this way. It requires, in fact, very complex communication patterns and channels.
- **Duplication**: The costs are lower, but it still requires the duplication of part of the system.

So, these issues lead us to another modification of blue-green, which is closer to the modern approaches that we will discuss soon.

Blue-green modification – red-black

Let's immediately start with the diagrams:

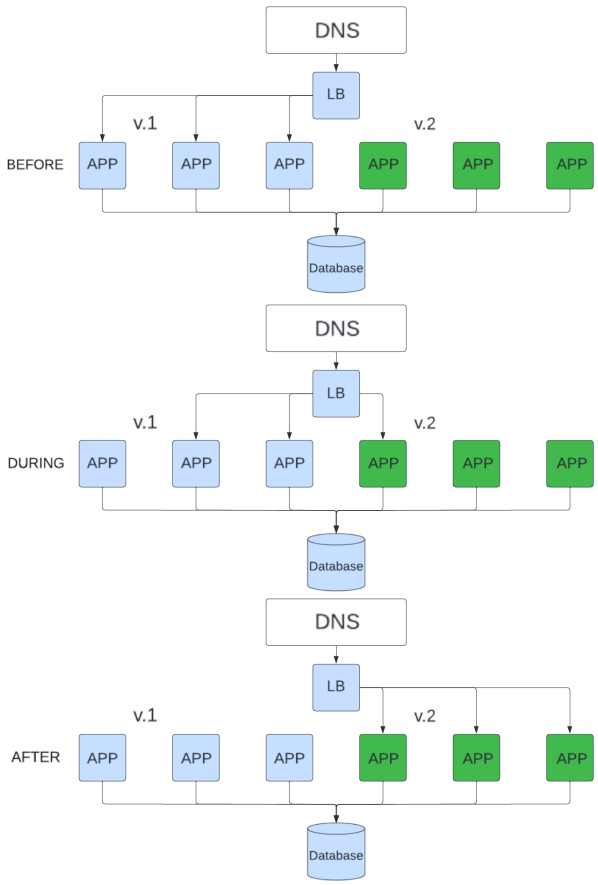

Figure 10. 11 – Red-black modification

In this approach, we can see two approaches combined. It includes both blue-violet, which we already discussed, and canary deployment, which we will discuss in detail in the next section.

As we will shortly go into detail on canary, let's clarify the middle step for now. In this step, we allow traffic to be split into specific percentages between two versions of our application. In this way, we prevent the situation when all traffic will be diverted to v.2 of our application, and the errors that would appear for every request.

This modification is a natural step after blue-violet. This is because blue-violet doesn't give us the full potential of using an inactive environment to test everything. It is part of an active application. The traffic is not diverted yet, yes, but all elements before and after the application are in use.

All three approaches presented so far are still very popular. However, in the modern world of microservices, two other patterns are more favored. We will look at them now.

Canary deployment

The name of this pattern was taken from the real approach used by miners years ago. Back then, miners didn't have equipment to measure the concentration of carbon monoxide in mines. Therefore, it was very common for them to die because of that. Canaries were less resistant to carbon monoxide, so the poor birds died faster than people. This was the signal to miners to escape from the dangerous areas.

As terrible as this comparison probably sounds to you now, canary deployment utilizes the same *technique*. We deploy one container with the new application. If the container dies, it means we need to roll back to the previous state.

Canary deployment is more suited for cloud-native workloads. We have already established that cloud-native workloads favor microservices, so many tools support this type of deployment natively.

Let's look at the following diagram to see how the process looks:

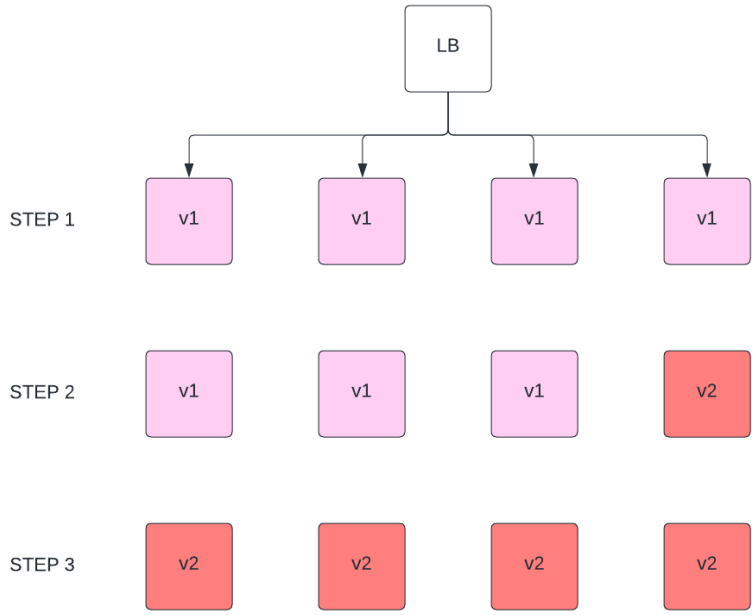

Figure 10.12 – Phases of canary deployment

Let's take a closer look at some elements and steps presented in this figure:

- **STEP 1**: Before deployment, v.1 of the application is active. We can see that the application is only using one version of the application.

- **STEP 2**: Deployment adds one container where v.2 of the application is already deployed. Orchestration of the service will root traffic to the new container too. Now, the role of CI/CD is to check whether the new container performs successfully. If yes, we can go to **STEP 3**.
- **STEP 3**: All containers are switched to v.2. If not, we need to go back to **STEP 1** and roll back to the workable version of the application.

This approach sounds easy, but we have to dig deeper into two patterns for rollbacks.

Rollback back

Rollback back is a technique or pattern that defines that if, during or after deployment, the application starts to log more errors than earlier, or worse, stops responding, the previous version of the application must be restored. This gives the team time to troubleshoot and fix the issue.

Rollback forward

Even if the name sounds a little controversial, this pattern is quite popular in canary and rolling deployment types.

In this pattern, we do not bring back the previous version of the application. Our task is to provide the fixes and deploy newer, already-fixed versions of the code.

Of course, in this case, the fix must be provided as quickly as possible, but through an established CI/CD process. This brings a lot of stress and anxiety to the team. This is something that managers must take into consideration when establishing the ways of work and delivery.

Now, let's look at the pros and cons of canary deployment:

Advantages	**Disadvantages**
There's no investment in architectural changes.	Our canary might be unreliable. In the mining analogy, there might be a pocket of good air where the canary is, but elsewhere, there might not be clean air. In the case of our canary deployment, there is a chance that during the relatively short time of the canary check, everything might be OK, but the real traffic might bring issues.
It's a quick way to evaluate the correctness of the deployment.	More effort is needed to stabilize the application in case of issues.
It's very easy to automate.	It has the complexity of managing multiple versions of your application simultaneously. This includes handling different versions of APIs, database schemas, and other components, which can add significant overhead to the deployment process.

Table 10.3 – Pros and cons of canary deployment strategy

The final deployment pattern will show you how to fix the disadvantages of canary deployment. However, as usual, new disadvantages will be brought into the picture.

Rolling deployment

We start this section with a diagram as well:

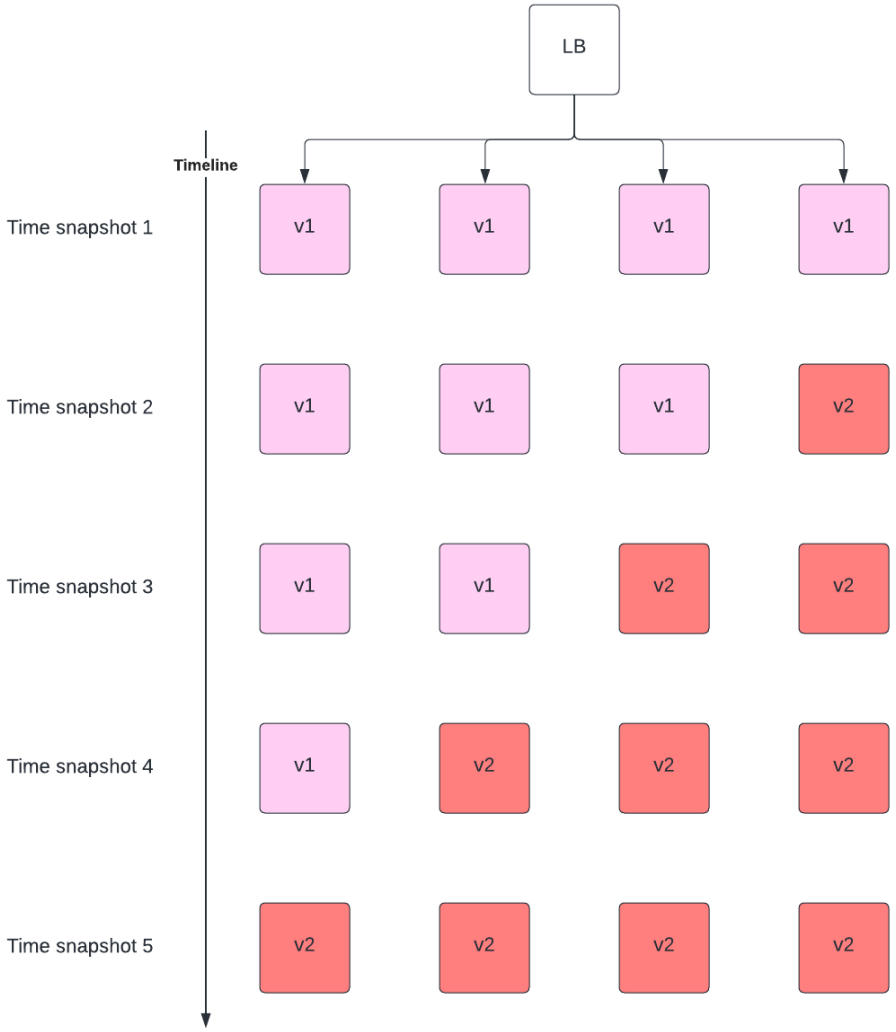

Figure 10.13 – Rolling deployment

In the rolling pattern, we define how much traffic will be directed to the old version of the application and to the new. The second parameter is how much time we need to ensure that the newest change is well-tested, and that the components can execute the next shift.

The preceding figure shows the approach where we roll 25% of the workload in the defined time periods.

What are the pros and cons of this pattern? Let's look at the table that follows:

Advantages	**Disadvantages**
Gradual deployment with many checkpoints	Longer deployment than in the canary pattern
A quick way to evaluate the deployment	Potential impact on CI/CD pipelines duration time
Lower risk of interruption of the service	Potential for service disruption during the update process. Since rolling deployments update a few instances at a time, there can be a period where different versions of the application are running simultaneously.

Table 10.4 – Pros and cons of rollback deployment strategy

Of course, rolling and canary patterns also have some modifications, but this is not the place to list them.

The cloud-native approach and cloud providers are in favor of quick changes, fast deployments, resource minimalization, and a use-what-you-need approach. The presented deployment patterns incorporate these elements into the defined process.

It is very important to analyze and select the proper deployment strategy, considering not only branching strategies and application architecture but also the platform where the deployment will be executed.

Team topologies

The cloud-native approach requires changes to the way in which organizations manage teams. In this section, we will strongly relay the must-read book, written by Matthew Skelton and Manuel Pais, called *Team Topologies* (`https://teamtopologies.com/book`). This book sets the stage for modern approaches such as Platform Teams but also organizes the knowledge of DevOps transition and management very well.

First, let us get to know the team topologies landscape a little bit better. The whole concept presented in the mentioned book is raised around Conway's Law (`https://martinfowler.com/bliki/ConwaysLaw.html`). Skelton and Pais created the concept of dynamic relations between teams involved in DevOps transformation.

In short, Conway's Law says that the design of the system mirrors the communication structure in the organization. This means that the communication structure plays a vital role in the way the systems are designed, and also in how the system's component cooperates.

The authors of *Team Topologies* created a set of four topologies that should be implemented in organizations. All of these are covered by three interaction modes, which connect these topologies.

Those four topologies are as follows:

- Stream-aligned team
- Enabling team
- Complicated subsystem team
- Platform team

These are the three interaction modes:

- X-as-a-service
- Collaboration
- Facilitation

We strongly recommend (again) reading the book to learn more about this concept.

Example – Landing zones, CI/CD, and team topologies

We can come back to the concept of landing zones, and more specifically, to the concept of delivering them with CI/CD and team topologies.

The best practice in this case is to transfer all responsibilities for a landing zone to the Platform Team (one of the four topologies in *Team Topologies*). This means that this team is fully responsible for the infrastructure of the landing zone, as well as the CI/CD process of this landing zone. At the same time, they are responsible for preparing the toolset that will the use landing zone.

Let's analyze the diagram that follows:

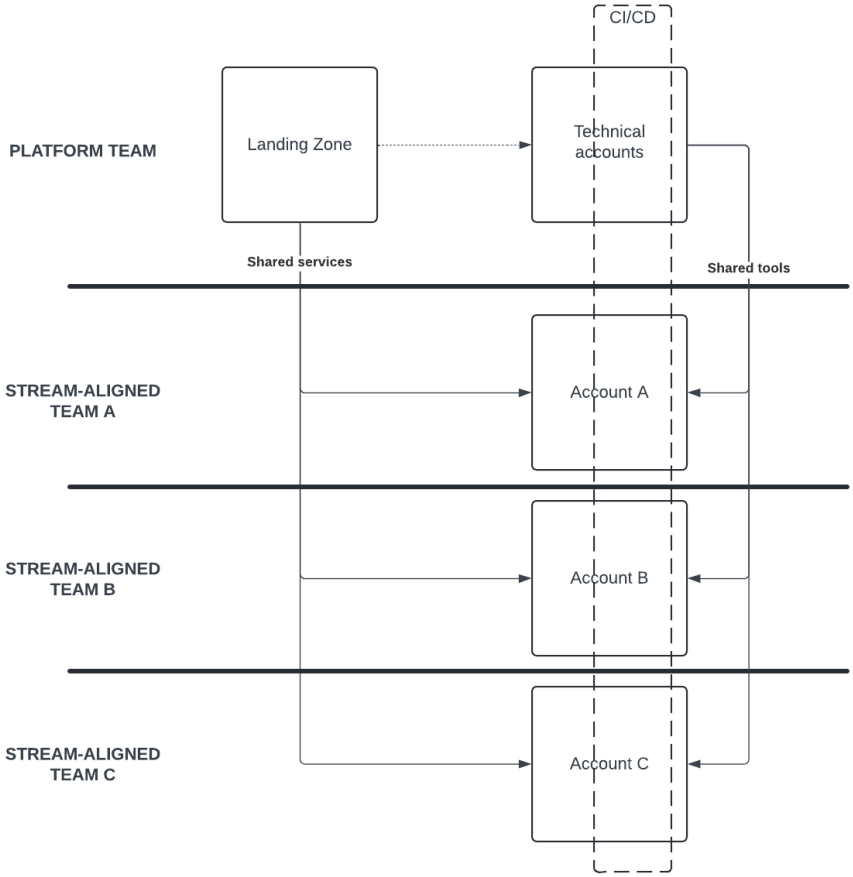

Figure 10.14 – Simplistic perspective of the setup

As we see in the diagram, the Platform Team is responsible for a few elements, including the landing zone and critical resources (such as network, accesses, etc.), as well as for the main (central) part of CI/CD. This means that the Platform Team not only manages the CI/CD tool (infrastructure, application, etc.) but is also responsible for the preparation of the core skeleton of pipelines.

Right now, it should be easy for us to circle back to topics that we covered earlier, such as standardization, scalability, modularization, and so on. At this point, we should see the importance of these elements and patterns. To implement a proper landing zone approach and, most importantly, a proper Platform Team strategy, we need to enforce these patterns. Without them, the effectiveness of the Platform Team will be close to none very quickly.

Practical examples

In this final section, we will go through the CI/CD pipeline for the serverless example we used earlier in the *API-driven communication for serverless* section.

For this case, we will use AWS CodePipeline. Why should we use this tool for the example? AWS CodePipeline is deeply integrated with many AWS services, including a serverless ecosystem. We would also like to bring your attention to one bad practice that is used very often.

CodePipeline has two distinct CI/CD components. It contains AWS CodeBuild for building the artifacts (the CI part), and AWS CodeDeploy for deployments (the CD part). However, we very often see that CodeDeploy is not used and that all steps in the pipeline are built with CodeBuild.

There are a few reasons why:

- **Use of CodeDeploy is not possible**: In many cases, we cannot use CodeDeploy, as this service does not allow us to prepare the scripts or commands.
- **Too much complexity**: CodeBuild is simple in its construction. It's very similar to everything we know from other CI/CD systems, such as Jenkins, Travis, and GitHub Actions. CodeDeploy has its own approach.

In the example that follows, we use AWS CloudFormation to create CodeBuild for building the artifact and CodeDeploy to deploy it. We do not include the whole pipeline, as it would be too long, but we focus on the snippets that are in our interest.

CodePipeline Build block – the CI part of the process

Let's look into a snippet for CodeBuild creation:

```
lambdaBuild:
     (1)
    Type: AWS::CodeBuild::Project
     (2)
    Properties:
      Name: lambdaPipeline_Build
      Description: Build the artifact
      ServiceRole: !GetAtt lambdaBuildRole.Arn
     (3)
      Artifacts:
        Type: CODEPIPELINE
     (4)
      Environment:
        Type: LINUX_CONTAINER
        ComputeType: BUILD_GENERAL1_SMALL
```

```
        Image: aws/codebuild/amazonlinux2-x86_64-standard:3.0
      Source:
        Type: CODEPIPELINE
        BuildSpec: !Sub |
    (5)
          version: 0.2
          env:
            shell: bash
          phases:
            install:
              runtime-versions:
                python: 3.8
            build:
              commands:
    (6)
                - !Sub aws cloudformation package --template-file
template.yml --s3-bucket ${artifactsBucket} --output-template-file
outputTemplate.yml
          artifacts:
            type: zip
            files:
              - template.yml
              - outputTemplate.yml
      TimeoutInMinutes: 5
```

This book is not about AWS or AWS CloudFormation. However, we can see the need to explain a few elements here:

- Like in other IaC tools, this is the logical name of the resource, which is used to create dependencies inside the template.
- We define the resource that will be created. In this case, it is a CodeBuild Project. This will be later available in the **CodeBuild** tab in the CodePipeline service.
- AWS, like other cloud vendors, heavily relies on proper access models. In this case, we need to provide the IAM Role that is responsible for this component.
- We mentioned earlier how artifacts can be stored. In this case, we store the artifact in the scope of the pipeline.
- Here, the `BuildSpec` is defined. This is the section where we can provide all configurations, actions, commands, scripts, and parameters for the pipeline.
- We need to determine the exact place where we build and store our serverless artifact.

CodePipeline Deploy – the CD part

After we have created the build phase, it is time to deploy the artifact:

```
        - Name: DeployLambda
          Actions:
          - Name: CreateChangeSet
      (1)
            ActionTypeId:
              Category: Deploy
              Owner: AWS
              Version: 1
              Provider: CloudFormation
            InputArtifacts:
            - Name: lambdaBuildOutput
            OutputArtifacts: []
            RunOrder: 1
      (2)
            Configuration:
              ActionMode: CHANGE_SET_REPLACE
              StackName: lambdaDemo
              ChangeSetName: simpleLambdaChangeSet
              TemplatePath: lambdaBuildOutput::outputTemplate.yml
              Capabilities: CAPABILITY_IAM,CAPABILITY_AUTO_EXPAND
              RoleArn: !GetAtt lambdaDeployRole.Arn
          - Name: ManualApproval
      (1)
            ActionTypeId:
              Category: Approval
              Owner: AWS
              Version: 1
              Provider: Manual
            RunOrder: 2
      (2)
            Configuration:
              NotificationArn: !Ref notificationTopic
              CustomData: 'Approve or reject the pipeline.'
          - Name: ExecuteChangeSet
      (1)
            ActionTypeId:
              Category: Deploy
              Owner: AWS
              Version: 1
              Provider: CloudFormation
```

```
            InputArtifacts:
            - Name: lambdaBuildOutput
            OutputArtifacts: []
            RunOrder: 3
      (2)
            Configuration:
              ActionMode: CHANGE_SET_EXECUTE
              StackName: lambdaDemo
              ChangeSetName: simpleLambdaChangeSet
              RoleArn: !GetAtt lambdaDeployRole.Arn
       ArtifactStore:
         Type: S3
         Location: !Ref artifactsBucket
```

The preceding code defines the deployment in three phases (labeled with `(1)`). The best practice for running CloudFormation templates is to execute updates through Changesets. This allows us to control the deployment, review the deployment, and document the changes.

Therefore, we have prepared the Changeset (with the `CreateChangeSet` action) and executed it (with the `ExecuteChangeSet` action). Additionally, we have a second phase (between the mentioned two) called `ManualApproval`. These steps are set in their order of execution (labeled with `(2)`) to avoid parallel runs. These steps must be run in sequence, one by one.

Canary example for AWS Lambda

The best way for a deeper understanding of building a canary deployment for AWS Lambda is to test the code available in the AWS Samples repo (`https://github.com/aws-samples/aws-lambda-deploy/tree/master`). This covers the canary deployment with rollback.

Summary

In this chapter, we learned how to adapt CI/CD design patterns to cloud vendors' processes (in our case, it was AWS) and cloud-native architecture.

The knowledge of deployment patterns and how to incorporate them into cloud processes enables the proper approach to CI/CD. Modularity, scalability, and standardization – all these elements are crucial to enable good design of CI/CD.

This knowledge, combined with applying these patterns, allows the organization to keep consistency and control over the delivery process, without dispatching a huge group of engineers to manage the service.

In the next chapter, we will see how we can audit and assess CI/CD. We have to remember that creating something new is one side of the coin, but there is a second side – maintenance and control. Without this second side, we will not be able to really maintain the effective process and all our efforts to create the best solution will be wasted.

Further reading

Use the following links to read about the dilemma between using mono-repositories and multi-repositories in serverless architectures:

- https://dev.to/bitdev_/monorepo-poly-repo-or-no-repo-at-all-4nel
- https://www.simform.com/blog/monorepo-vs-polyrepo/
- https://www.linkedin.com/pulse/monorepo-vspolyrepo-rafael-vargas/
- https://buildkite.com/blog/monorepo-polyrepo-choosing

11
Auditing and Assessment of Design Patterns

CI/CD design patterns need to be ready for audits and controls to mitigate the risks of rapid integration of new technology. Additionally, Audits help override ambiguity and confusion. In this chapter, we will discuss the need to ensure proper monitoring, what should be implemented, and how to build measurements and controls into the process.

We will discuss the importance of audits and assessments for the CI/CD workflow. With CI/CD, auditing and assessments have evolved significantly. Understanding the various tools and techniques available today for auditing is also critical for using them. Gathering evidence, observing the system, embedding controls, and even documentation are integral parts of the audits. This chapter provides an in-depth understanding of the auditing process and the importance of assessments.

We will cover the following topics in this chapter:

- Overview – taxonomy of assessment and audits
- Conducting a comprehensive assessment of CI/CD design patterns
- Shifting the auditing process left
- Common audit checklist for the CI/CD design patterns
- Impact of audits and assessments
- Maturity model of design patterns

Overview – taxonomy of assessment and audits

In this chapter, we will discuss the assessment and auditing of CI/CD design patterns. Assessments and audits can be beneficial to improve the operational performance of CI/CD. There are other outcomes of audits and assessments, such as ensuring compliance and maintaining the right security posture for the pipeline. Before we go into the details, let's first get familiar with the key terms:

- **Taxonomy**: This is a science that deals with naming, describing, and classification. In the context of auditing and assessments, we will attempt to characterize different kinds of audits and assessments. We will also discuss the different methods, tools, and techniques for performing these activities for CI/CD.
- **Assessments**: Using surveys, questionnaires, and so on to come up with recommendations and actionable guidance to improve the state of a particular system. Assessments can range from a broad scope to a specific scope depending on the required outcome. Some types of assessment often performed for CI/CD include but are not limited to the following:
    - Performance assessment
    - Risk assessment
    - CI/CD portfolio assessment
    - Financial assessment
    - Prescriptive assessment

    The assessment services are often offered as complementary to existing or new customers with the intent to optimize and improve the efficiency and resiliency of the system.
- **Audits**: As per the **International Organization for Standardization** (**ISO**), an audit is defined as the *"systematic, independent, and documented process for obtaining objective evidence and evaluating it objectively to determine the extent to which audit criteria are fulfilled."* An audit can be conducted for a range of criteria such as the following:
    - Security and compliance requirements.
    - Requirements specified by an organization for a particular project, or system-related guidelines
    - Demonstrating evidence for policy adherence to build trust and enhance brand reputation

Audits can differ in scale, purpose, and process. Sometimes, audits are conducted by internal organizations while others are done through third-party organizations. The audits can be technology-driven or done manually. We will discuss more details about auditing a CI/CD design pattern throughout this chapter.

In the next section, we will discuss in detail the process, tools, and outcome of conducting a comprehensive assessment of CI/CD design patterns.

Conducting a comprehensive assessment of CI/CD design patterns

In this section, we aim to provide some guidance on assessments. Assessments are often done through surveys or questionnaires, observational tools, or software. The process of performing assessments can differ depending on who is performing the assessment. The assessment can be done through a software or observability tool, but the following steps must be performed to complete it:

1. Start the assessment by defining the key objectives and outcomes. The assessment can be either a performance assessment, portfolio assessment, or financial assessment.
2. Planning the assessment is also an important step as it will include the frequency of the assessment, duration of the assessment, tools that will be used for performing this activity, and so on.
3. After the planning stage, the actual execution starts, and the required data is collected through the tools that were chosen to be used.
4. These assessments can be broad, starting from simple interpretation to complex exploration. For example, a financial assessment can be quite complicated and might require sophisticated tools for rightsizing recommendations, licensing recommendations, and optimizing infrastructure.
5. The final step is producing recommendations based on the data analysis, specific insights, and key actions as recommendations.

The following figure shows a simple depiction of the assessment workflow:

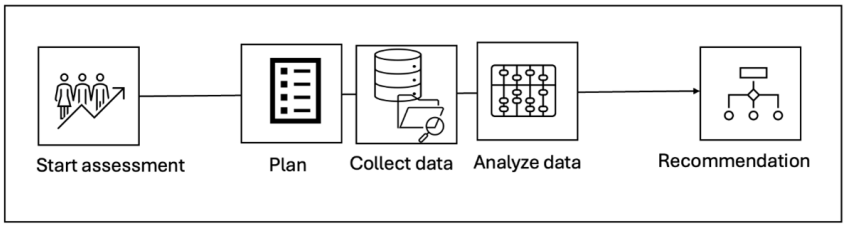

Figure 11.1 – How the process of assessment works

In the next section, we will explore some tools and specific practices that can be used for audits and assessments in general for CI/CD design patterns.

Tools and techniques for comprehensive assessments

Often, organizations prefer to use specific tools and techniques for assessments.

One of the popular techniques for performance management of CI/CD deployments is monitoring and measurement through **DevOps Research and Assessment** (**DORA**) metrics. DORA is a program by Google Cloud that identified four primary key metrics used to indicate the velocity and stability of software development. DORA metrics are discussed in the *Introduction to CI/CD measurements* section in *Chapter 4*.

Another technique to use is gamification or remuneration, which also helps in changing behavior. This is a tier-based scoring system that motivates/encourages teams/practitioners/systems to do better.

Yet another method is using assessment tools, including discovery tools, to analyze the topology of the existing CI/CD approach and provide strategic recommendations. In the following list, we present some examples of tools that can be used for different assessments. Note that at the time we wrote this book, these tools were used for assessment purposes. Please ensure that the tools are not obsolete by checking the links:

- **Txture**: This tool is used for different types of assessment, the most popular being infrastructure mapping. More information can be found at `https://txture.io/en`.
- **Matilda**: This tool is used for agentless discovery of application topology and associated software and hardware infrastructure. More information can be found at `https://www.matildacloud.com/`.
- **Apache DevLake™ (Incubating)**: This tool ingests, analyzes, and visualizes the fragmented data from DevOps tools to distill insights for engineering excellence. More information can be found at `https://devlake.apache.org/`.

Checklist for specific design patterns

Every chapter provides detailed guidance to create a checklist for each of the specific design patterns described in those chapters and it serves as a guideline on how these design patterns can be implemented in a standardized way.

In the next section, we will discuss audits, which are broader in scope and complexity, and often used for performing big-picture analysis.

Conducting comprehensive audits on CI/CD design patterns

Audits are a more formal process to address any potential gaps in the overall CI/CD posture. They involve policies and data security, encompass compliance and regulatory requirements, and follow a defined documentation process.

Audits are often done for third-party assurance. Regular internal audits and health checks are done to facilitate third-party audits, keeping a close eye on activities and bringing the right level of management control, examining changes, and demonstrating that regulatory and compliance standards are followed.

Many advocates talk about shifting the audit process due to the complexity of the activity. Auditors might require information and data points regarding deployment success rates, for example. We also highlighted the need for embedding policy as code specifically for regular sectors in *Chapter 8*.

The auditability of CI/CD design patterns is directly associated with the traceability of the pattern. Several traceability tools are available to produce artifacts to support audits. The audit process can be divided into the following phases:

- **Planning phase**: This phase typically consists of gaining an understanding of the system. This is done through traceability tools, available evidence, artifacts, policy definitions, and so on.
- **Developing findings**: This phase includes investigations, testing, vulnerability assessments, and data analysis. To process and correlate the findings from the different types of activities like testing, etc., various types of evidence-gathering techniques are used including deploying audit tools and filters.
- **Reporting phase**: During the reporting phase, reports are developed, and findings are presented.
- **Follow-up phase**: This phase mainly focuses on the corrective actions management has taken to rectify the reported gaps and anomalies and how many of the reported findings have been resolved.

Now, to start secure at an early phase, **Shift-left** is the keyword today. Let's have a look at it.

Shifting the auditing process left

With rapid advancement in technology, it is evident that the approach toward audits and assessment will have to be reevaluated. The bright spot or optimized and efficient way is to implement audit controls into the code, which should be part of the development process from the inception stages of the CI/CD workflow. Shift-left is a practice that helps software practitioners detect inconsistencies early in the SDLC process by addressing the audit requirements from the beginning through code. Traditionally, auditing processes are dealt with post-implementation. However, if you look at the design pattern of CI/CD, it is valuable to treat auditing with a shift-left mindset. The shift-left approach to auditing within the CI/CD pipeline is a transformative strategy that enhances the efficiency and effectiveness of the development process. By embedding audit mechanisms early in the development cycle, organizations can detect issues sooner, reduce the risk of late-stage failures, and ensure compliance with regulatory standards. This proactive stance on auditing not only streamlines the implementation of CI/CD but also drives a culture of continuous improvement and accountability.

Integrating auditing-related code into the initial phases allows for the automation of compliance checks, making the process less labor-intensive and more reliable. Templates and checklists serve as standardized tools that guide developers in adhering to best practices, while guidelines ensure that everyone is aligned with the organization's audit requirements. These resources, when incorporated early, contribute to a more cohesive and secure development life cycle.

Moreover, the use of well-defined audit trails and telemetry within the CI/CD pipeline provides a transparent view of the development process, enabling real-time monitoring and reporting. This visibility is crucial for identifying deviations from expected outcomes and for providing evidence of

compliance during external audits. The immediate availability of comprehensive audit data through automation reduces the time and resources typically required for post-implementation audits.

In essence, the shift-left auditing approach aligns with the agile nature of CI/CD, promoting rapid iterations, quick feedback loops, and incremental improvements. It empowers teams to take corrective actions promptly, ensuring that the end product is of excellent quality and compliant with all necessary standards. As organizations continue to embrace this methodology, they will likely see a reduction in delays, cost overruns, and compliance issues, leading to a more robust and resilient software development process.

Now, let's have a look at the common audit checklist. Now, let's have a look at the common audit checklist.

Common audit checklist for the CI/CD design patterns

In this section, we will highlight some of the common checklist items to audit the CI/CD design pattern. These checklist items can be integrated into the CI/CD workflows to make the design patterns traceable, increase visibility, and improve auditability.

The checklist to be used can be divided into **general audit points** and **specific audit points**. Let's first have a look at the general audit points.

General audit points

In this section, we will discuss the general audit points for CI/CD design patterns. These help organizations streamline the implementation at the foundational level but also help reduce general risks in the implementation of the CI/CD workflow. They are as follows:

- **Source code management**: Ensures all code changes are tracked in a **version control system** (**VCS**), with proper branch management and code review processes in place.
- **Build automation**: Verifies that builds are automated and reproducible, and that build scripts are versioned and maintained. Ensures build failures are reported and addressed promptly.
- **Continuous integration** (**CI**): Confirms that automated tests are run on every code commit, including unit, integration, and regression tests. Ensures code quality checks are integrated into the CI process.

  Continuous delivery (**CD**): Ensures deployment to staging and production environments is automated, with versioned deployment scripts and tested rollback mechanisms.
- **Security**: Checks for the integration of security testing tools in the CI/CD pipeline, secure management of secrets, **software bill of materials** (**SBOM**) production, and proper access controls for CI/CD tools and environments.
- **Monitoring and logging**: Ensures build and deployment logs are retained and accessible, with monitoring tools in place to track the health of the CI/CD pipeline and alerts configured for critical failures.

- **Compliance and documentation**: Verifies that the CI/CD process complies with relevant industry standards and regulations, with up-to-date and accessible documentation and maintained audit trails for all CI/CD activities.

These points are important to ensure consistency, enhance security, improve quality, and many more benefits. The following specific audit points can help further improvement.

Specific audit points

In this section, we will discuss the specific audit points for CI/CD design patterns. These help organizations not only optimize the implementation but also reduce specific risks in the implementation of the CI/CD workflow. They are as follows:

- **Review controls**: This checklist item includes auditing the version control flow, role-based access controls, pipeline-based access controls, branching and release protection rules, and credential control and management. These aspects have been discussed in the previous chapters in detail.
- **Artifact management**: The resource generated, or consumed, by the CI/CD process should be audited for integrity and security. Auditing practices such as artifact verification, code signing, and configuration drift usually applicable to environmental resources are recommended as a checklist item.
- **Identity management**: This includes an inventory of external, shared, and local identities, as well as a catalog of stale identities.
- **Analysis and mapping of identities**: This checklist item provides recommendations for removing unnecessary roles, permissions, and identities.
- **Inventory of tools and applications**: This activity manages supply chain dependencies, optimizes the total number of tools and applications involved across the CI/CD ecosystem, and helps in license management and other related improvement opportunities.
- **Automation workflow**: This checklist item ensures how the CI/CD performs in runtime, by auditing build automation, testing automation, and deployment automation.
- **Assessing risks**: While risks differ from one design pattern to another, some of the commonly auditable risks are related to pipeline attacks, or security flaws in the CI/CD software stack, resulting in the compromise of the entire system.
- **Third-party services**: Nowadays, third-party services are an integral part of the pipeline. For example, GitHub **SaaS** (short for **Software as a Service**) offers many integration points and interfaces. Often, these types of integration result in a complex mesh of third-party offerings, and full visibility is required for secure deployments.

How can we implement these audit points into our CI/CD pipelines? The answer is quality gates.

Implementing audit points with quality gates

There are several steps needed to set up quality gates to implement the audit points. These are the steps to be taken:

1. **Define quality criteria**: Identify the key quality metrics and standards you want to enforce, such as code quality, security, compliance, and performance.
2. **Select tools and integrations**: Choose tools that can help you enforce these criteria. Examples include static code analysis tools (e.g., SonarQube), security testing tools (e.g., Snyk), and performance testing tools (e.g., JMeter).
3. **Integrate tools into the CI/CD pipeline**: Integrate the selected tools into your CI/CD pipeline. This can be done using plugins or APIs provided by your CI/CD platform (e.g., Jenkins, GitLab CI, or GitHub Actions).
4. **Configure quality gates**: Set up quality gates within the tools to define the thresholds and conditions that must be met. For example, you can configure SonarQube to fail the build if code coverage is below a certain percentage or if critical vulnerabilities are found.
5. **Automate quality checks**: Ensure that quality checks are automated and run at specific stages of the pipeline, such as during code commits, pull requests, or before deployment.
6. **Monitor and review results**: Continuously monitor the results of the quality gates and review any issues that are flagged. Ensure that the team addresses these issues promptly.
7. **Iterate and improve**: Regularly review and update the quality criteria and thresholds based on feedback and evolving standards. Continuously improve the quality gates to adapt to new challenges and requirements.

To make this a little bit clearer, we are going to see how we can implement a quality gate into popular tools such as Jenkins and SonarQube.

Example – Implementing a quality gate with SonarQube in Jenkins

The following steps show how to implement a quality gate with Jenkins and SonarQube:

1. **Install the SonarQube plugin**: Install the SonarQube plugin in Jenkins.
2. **Configure SonarQube Server**: Configure the SonarQube server details in Jenkins (**Manage Jenkins** | **Configure System** | **SonarQube Servers**).
3. **Create a SonarQube project**: Create a new project in SonarQube and generate a project key and token.
4. **Add SonarQube analysis to the Jenkins pipeline**: Add a SonarQube analysis stage to your Jenkins pipeline script:

```
pipeline {
    agent any
```

```
    stages {
        stage('Build') {
            steps {
                // Build steps
            }
        }
        stage('SonarQube Analysis') {
            steps {
                script {
                    def scannerHome = tool 'SonarQube Scanner'
                    withSonarQubeEnv('SonarQube') {
                        sh "${scannerHome}/bin/sonar-scanner
-Dsonar.projectKey=my-project -Dsonar.sources=src"
                    }
                }
            }
        }
        stage('Quality Gate') {
            steps {
                timeout(time: 1, unit: 'HOURS') {
                    waitForQualityGate abortPipeline: true
                }
            }
        }
    }
}
```

5. **Review results**: Review the results in SonarQube and ensure that the build passes the quality gate criteria.

In the preceding code snippet of the Jenkins pipeline, it is evident that it ensures the code is built, analyzed by SonarQube for topics such as code quality and security issues, and then checked against the quality gate criteria. If the code does not meet the quality standards, the pipeline will be aborted, preventing further deployment.

By following these steps, you can implement quality gates in your CI/CD pipeline to enforce quality standards and improve the overall security and reliability of your software. So, you can use this approach to implement the audit points.

In the next section, we will discuss the impact of audits and assessments and why it is important to make the CI/CD design pattern auditable.

Impact of audits and assessments

In this section, we will discuss the importance of audits and assessments. In the recent past, we have seen several cases of data breaches, security breaches, and systems getting adversely affected by vulnerabilities and even an unauthorized person or process deploying releases in production. These affect not only the reputation of the company but also cause substantial monetary loss. The business value of auditing and assessing the CI/CD design patterns is immense. In modern times, with rapid technology adoption, CI/CD enables organizations to scale software delivery with speed and it is important to ensure the following features are embedded in the CI/CD workflows:

- Traceability and control of the CI/CD workflow
- Keeping pace with changing policies
- Configuring guardrails
- Injecting security
- Managing access control

Let's look at some cases, including the lessons learned, which will make it clearer what impact audits and assessments have on an organization's CI/CD software stack:

- **Case 1 – Data and security breaches due to missing security controls**: A large international company should have conducted thorough security assessments, plus penetration testing and code reviews, of its CI/CD pipelines but failed to do so. This inevitably resulted in a malicious actor infiltrating their production servers and deploying malware. The attack wasn't picked up due to the lack of traceability and control within the CI/CD process. This resulted in a massive data breach, reputational damage, and huge customer claims:
    - **Lessons learned**: A well-designed audit of the security configuration in their CI/CD processes could prevent these from happening. They should have implemented stronger security measures, which would have resulted in preventing unauthorized access.
- **Case 2 – Non-compliance with changing policies and regulations**: A software company was punished severely for failing to comply with new data security regulations. The issue arose because their CI/CD workflows were not updated quickly enough to reflect the changing regulatory landscape. Additionally, their audit programs were infrequent and insufficiently thorough to keep up with evolving compliance demands:
    - **Lessons learned**: This case emphasizes the critical importance of aligning CI/CD workflows with regulatory updates. Regular, thorough assessments and audits are essential for keeping up with policy changes and ensuring timely compliance. Regular compliance checks should be implemented, which must be frequently updated with changed regulations.

- **Case 3 – Unauthorized production deployment due to improper access**: In this case, an organization granted a developer access to the production environment due to inadequate access controls, which was a huge mistake. The developer deployed an untested release with a lot of bugs, causing a widespread system outage and data loss. Why this could happen lies in the absence of regular access audits:
    - **Lessons learned**: This case once more shows the importance of routine access management audits in preventing unauthorized deployment. Establishing clear, controlled access mechanisms to ensure only authorized personnel have access to production environments is essential for system security.

Business value impact

The importance of audits and assessments extends beyond risk management; they are critical for maintaining customer trust, ensuring regulatory compliance, and protecting a company's reputation. By regularly auditing CI/CD designs and processes, businesses can identify and address issues early, providing secure and reliable software delivery. This underscores the need for leadership to promote a culture of security and compliance, ensuring that audits and assessments are not just a formality but also a fundamental part of the CI/CD process.

The cases mentioned in this section highlight the strategic importance of audits and assessments in securing and enhancing CI/CD workflows. By ensuring traceability, adapting to policy changes, strengthening security measures, and managing access, businesses can significantly mitigate risks and prevent them on the forehand, instead of solving them afterward.

In the next part, we will further discuss the next steps to further improve the quality of design patterns through the maturity model approach.

Maturity model of design patterns

In the previous section, we discussed the audits and assessments of design patterns. These techniques are useful for maturing the design patterns and their adoption as they are also a measure of the effectiveness of a design pattern.

If we talk about the maturity model of design patterns, it can be one way to improve the quality of the design patterns discussed in this book. The design patterns for CI/CD are still evolving. In most cases, if design patterns are adopted, they enable organizations to stay out of the overhead of coordinating, maintaining, and sustaining several different postures for CI/CD, removing inconsistencies and improving the effectiveness of CI/CD in general.

Now let's try to answer this question: How is CI/CD implemented in your organization? Once you have answered the question, the stages of the maturity model of the design pattern implementation of CI/CD can indicate where you and your organization are in the journey. Also, it will help you identify

the next steps to mature the adoption of CI/CD. We can describe the maturity model of the design patterns of CI/CD with the help of the following figure:

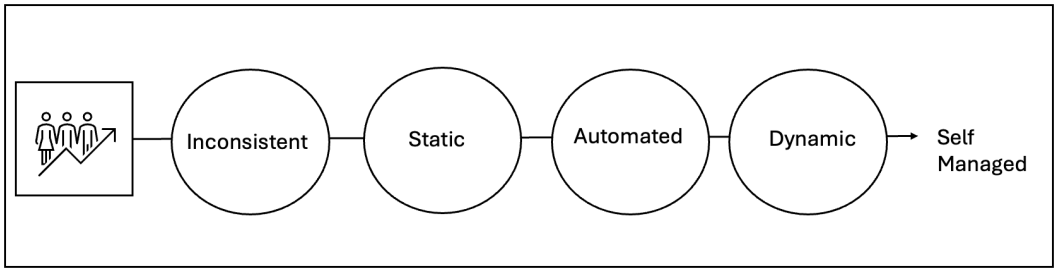

Figure 11.2 – Phases of maturity in CI/CD design patterns

Design patterns for CI/CD is a relatively new concept. It will take time to transform into a mature offering. In the following list, we highlight the different phases of the maturity of CI/CD design patterns by organizations:

- **Inconsistent**: This reflects a stage where the design pattern for CI/CD is mostly absent. Organizations have not even tried to unify the commonalities in different implementations.
- **Static**: At this stage, organizations start to realize the benefit of using common design patterns for similar CI/CD needs. Organizations try to create static guidelines or checklists. This approach is good in the beginning to build a common vocabulary but can get obsolete with time.
- **Automated**: As the system scales, it is not possible to create the guidelines and checklist manually through assessments and audits. Organizations will begin to automate parts of the process through modern tools and applications. As a next step, we will see that design patterns can automatically be generated by feeding the requirements, consistently.
- **Dynamic**: At some stage, the design patterns will start to evolve and there will be a need to induce observability tools to assess, recommend, and implement continuous improvement in the patterns. At this stage, organizations are on a self-evolution journey, with pattern libraries, code snippets, and code examples for developers to use. The checklist and guidelines can be dynamically extracted through a dynamic system.
- **Self-managed**: A self-managed inventory of design patterns with associated artifacts will be the highest level of maturity for a CI/CD system. In this stage, multiple teams can pull the pattern library to make use of it.

The maturity model for design patterns for CI/CD requires more than a technology outlook. Organizations need to put some serious effort into keeping things together. In the next subsection, we will explore the common challenges that organizations face in the assessment and auditing of the CI/CD design patterns.

Common challenges for comprehensive audits and assessments

Today, organizations are facing many challenges with CI/CD including audits and assessments. Some of the common challenges that we must overcome in this process are as follows:

- **Over-standardization**: The design patterns are for creating synergies and finding the reusable aspects from various similar implementations into a common one. The key is that these design patterns remain flexible, and do not result in slowing down the system.
- **How companies organize the teams**: Avoid creating an ivory tower; the design pattern's objective is to simplify the approach of software delivery. Let the practitioner decide how the design pattern evolves, and a more federated approach is required to deal with such evolution.
- **Documentation and communication**: Clear documentation and guidelines on how to contribute to the evolution with an array of practitioners working together can create some overhead. However, the overhead of creating and maintaining documentation is worth sustaining the design patterns, associated templates, code, libraries, and so on.
- **Linear progression**: The maturity models traditionally evolve in a linear fashion, which creates constraints on how the organizations adopt and move the needle to the next stage of maturity The maturity model phases described previously are only indicative and not meant to restrict the creative thinking of the teams.

In the next section, we will discuss some proposed next steps for comprehensive audits and assessments.

Next steps for comprehensive audits and assessments

Now, we have come to the end of this chapter. Before we close this discussion, we would like to highlight the next steps in the journey of design patterns for CI/CD.

As more and more applications are leveraging AI-based features, it becomes extremely critical to ensure the trustworthiness and accountability of software workflows. CI/CD audits will apply more progressive approaches to audit and assess the workflows.

Generative AI-based assessment can also be leveraged to speed up surveys and questionnaire writing and make assessments and internal audits more accessible.

Summary

In this chapter, we looked into the importance of audits and how assessment of the CI/CD design pattern can benefit organizations and practitioners. Furthermore, we discussed different tools and techniques to perform the audits and assessments. These practices not only ensure the robustness and efficiency of CI/CD pipelines but also enhance the learning and development of teams by identifying areas for improvement and embracing a culture of continuous improvement.

In the next chapter, we will discuss adopting advanced design patterns for CI/CD covering self-learning design patterns and further evolution of design patterns and roadmap for adoption of these advanced design patterns.

Further reading

Here, you can find the sources to expand your knowledge about the specific concepts in this chapter:

- *Role of internal audit in DevOps* – This article discusses the key challenges and associated controls related to DevOps including CI/CD: `https://kpmg.com/kpmg-us/content/dam/kpmg/pdf/2020/role-of-internal-audit-in-devops.pdf`.
- This article outlines the audit controls for pipelines, and how to produce automated audit reports: `https://www.devopsinstitute.com/wp-content/uploads/2021/07/CICD-21-AndersWallgren-SlideDeck.pdf`.
- Bajpai, G., and Schuetz, T. (2023). *Strategizing continuous delivery in the cloud: Implement continuous delivery using modern cloud-native technology*. Packt Publishing Ltd.

Part 5: Advanced Design Patterns and Anti-Patterns for CI/CD

In this part, you will learn about common anti-patterns or scenarios that need to be avoided while designing a CI/CD pipeline. In addition to this, you will also learn how to create pipelines for advanced use cases, such as machine learning pipelines for training or inference purposes.

This part has the following chapters:

- *Chapter 12*, *Advanced CI/CD Design Patterns and Use Cases*
- *Chapter 13*, *Exploring Anti-Patterns for CI/CD Design Pattern Deployments*

12
Advanced CI/CD Design Patterns and Use Cases

Integrating advanced design patterns with machine learning and generative AI is revolutionizing how we approach software development and deployment. When combined with the agility of near-real-time, utility-based CI/CD pipelines, these patterns enable systems to be more adaptive, resilient, and aligned with business needs. Machine learning algorithms can analyze huge volumes of data to improve decision-making processes. At the same time, generative AI can automate code creation, reducing the time and effort required for development tasks. Together, they facilitate a more dynamic and responsive design pattern that can be provisioned in real time, ensuring that applications are continuously integrated and delivered with minimal downtime. This approach not only accelerates the development cycle but also enhances the quality of the software, as it allows immediate feedback and iterative improvements.

As we get deeper into this chapter, we will explore the complexities of these patterns, their practical use, plus the benefits they bring to modern software engineering practices. We will examine case studies that demonstrate the successful implementation of these patterns in various industries. This will provide a comprehensive understanding of their potential and the best practices for their integration. The goal is to provide you with the knowledge and tools to use these advanced design patterns, ultimately leading to more efficient, effective, and innovative software solutions.

The following topics will be covered in this chapter:

- Self-learning CI/CD design patterns
- Understanding real-time utility-based CI/CD design patterns
- Further evolution and roadmap considerations

Let's look at the first topic in this chapter.

Self-learning CI/CD design practices

The concept of self-learning in CI/CD refers to the ability of systems to adapt and improve their deployment patterns based on feedback and data-driven insights. This involves the use of machine learning algorithms to analyze past deployments, identify patterns, and predict outcomes to enhance the CI/CD pipeline's efficiency.

Design patterns in CI/CD provide, as mentioned several times in this book, a structured approach to solving common problems encountered during software delivery.

Anti-patterns, on the other hand, are common responses to recurring problems that are ineffective and counterproductive. Recognizing these anti-patterns is crucial as they can lead to inefficiencies and increased risk in the software delivery process. For example, the *Manual Intervention* anti-pattern, where manual steps are required in the deployment process, can lead to errors and delays.

Self-learning CI/CD systems aim to minimize these anti-patterns by learning from historical data and progressively refining the deployment process. By incorporating analytics and machine learning, these systems can forecast potential issues and suggest improvements, making the CI/CD process more robust and less error-prone.

In practice, implementing self-learning in CI/CD requires a combination of tools and practices. VCS, automated testing, monitoring tools, and feedback loops are integral components. These elements work together to collect data, provide insights, and facilitate the continuous improvement of the CI/CD pipeline.

Moreover, embracing a culture of continuous learning and experimentation is vital. Teams and organizations should be encouraged to explore new tools, techniques, and processes to enhance their CI/CD practices, without following every fad that comes by in the IT-world. This culture fosters innovation and helps organizations stay up-to-date in the dynamic world of software development.

So, self-learning CI/CD design practices represent the intersection of software engineering best practices and machine learning technologies. By making use of these patterns, organizations can create more efficient, reliable, and adaptive CI/CD pipelines, ultimately leading to better-quality software and faster delivery cycles. As the field evolves, we can expect to see more sophisticated self-learning mechanisms being integrated into CI/CD systems, further revolutionizing the way we think about software delivery.

Maybe it's better to visualize this with a simple use case of a workflow of a self-learning CI /CD system:

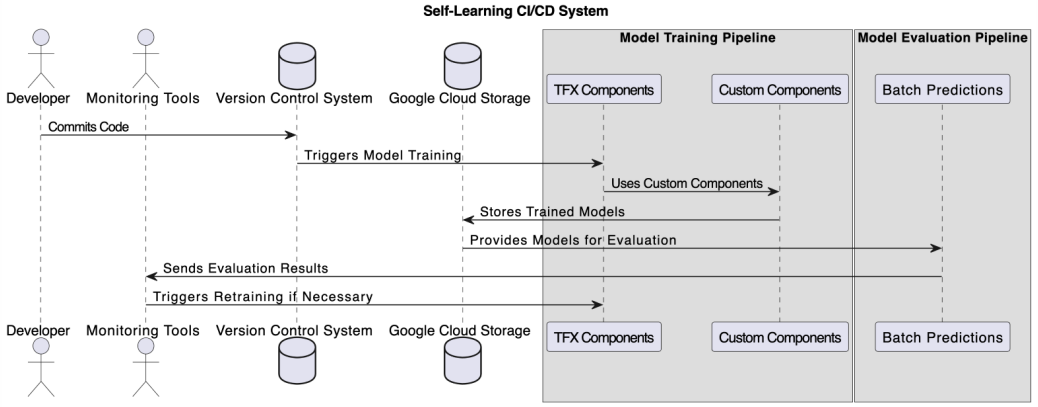

Figure 12.1 – Self-learning CI/CD system

This diagram illustrates the following steps:

1. The developer commits code to the **Version Control System**.
2. The VCS triggers the **Model Training Pipeline**.
3. The **Model Training Pipeline** uses **TensorFlow Extended** (**TFX**) components and custom components.

> **What is TensorFlow?**
>
> TensorFlow is a library for machine learning and **artificial intelligence** (**AI**), it's open source and particularly known for its applications in deep neural networks. Developed by the Google Brain team, TensorFlow allows researchers and developers to create complex machine learning models with ease. Its flexible architecture enables deployment across various platforms, from desktops to servers, and on mobile devices. With a focus on both research and production, TensorFlow supports a wide range of programming languages, making it a versatile tool in the field of AI.

4. Put the elements in the right order.
5. The trained models are stored in **Google Cloud Storage** (**GCS**).
6. GCS provides the models for evaluation in the Model Evaluation Pipeline.
7. The Model Evaluation Pipeline evaluates the models based on batch predictions and sends the results to the monitoring tools.
8. If the model's effectiveness declines due to data changes, the monitoring tools trigger retraining in the Model Training Pipeline.

What we see here is the concept of self-learning CI/CD systems in the context of **Machine Learning Operations** (**MLOps**) that can adapt to changes in data and real-world context. The system uses continuous evaluation and retraining through two main pipelines – the Model Training Pipeline and the Model Evaluation Pipeline – as well as VCDs, monitoring tools, and feedback loops. The technology stack includes **TensorFlow Extended** (**TFX**), **Keras**, and **Google Cloud Platform** services, just to give you an idea of what could be involved. Overall, the approach is more important, which ensures robust and error-resistant deployment processes for smarter and more efficient pipelines.

Besides using AI and machine learning technologies, organizations should also address ethics and develop a company policy around it. Why?

Organizations must approach the integration of AI into CI/CD processes with a robust ethical framework that prioritizes transparency, accountability, and fairness. It is essential to establish clear guidelines that recognize the potential risks and ethical considerations associated with AI deployment in these areas. This includes ensuring that AI systems are designed and implemented to support meaningful work, rather than replace it, thereby enhancing the skills and contributions of human workers. Furthermore, organizations should commit to continuous ethical training and awareness for all stakeholders involved in the CI/CD pipeline to cherish an environment of ethical vigilance.

AI's role in CI/CD can lead to significant improvements in efficiency and quality assurance through intelligent testing and optimization of pipelines. However, it is crucial to monitor these systems for bias and errors that could lead to unfair outcomes or discrimination. Regular audits and reviews of AI systems, conducted by diverse and interdisciplinary teams, can help identify and mitigate these risks. Additionally, organizations should engage with external experts and stakeholders to gain broader perspectives on the ethical implications of AI in CI/CD processes.

Given the fact that AI technologies are evolving rapidly, adaptive governance structures are necessary to respond to new ethical challenges as they arise. This includes the creation of ethical advisory boards and the implementation of AI ethics principles, such as those suggested by leading research and industry groups. By doing so, organizations can ensure that their use of AI in CI/CD not only adheres to current ethical standards but also evolves alongside emerging best practices and societal expectations.

Ultimately, the goal is to use AI in a way that respects and enhances human dignity, promotes equitable outcomes, and contributes positively to organizational and societal well-being. This proactive and thoughtful approach to the ethical integration of AI into CI/CD processes will be a defining factor in the responsible advancement of technology within the industry.

To get to that point, it is important to look at the concept of **real-time utility-based CI/CD design patterns** for maximizing your practices in real time.

Understanding real-time utility-based CI/CD design patterns

That term seems quite a mouthful... furthermore, as we are exploring new ground in this book, you might never have heard it before.

In fact, this is not a set of patterns, but rather a concept that involves CI/CD practices.

When it comes to real-time utility, a set of practices can be applied to optimize the value delivered by software at any given moment. This process entails dynamically adjusting CI/CD practices based on real-time feedback, performance metrics, user behavior, and other relevant factors. The specifics of this approach will depend on the particular project or organization's needs and constraints. It is imperative to note that this process can be key to enhancing the effectiveness of software systems, particularly in the context of real-time applications. By leveraging real-time feedback and data, organizations can make informed decisions that improve their software's overall performance and efficacy.

Real-time utility-based CI/CD design patterns are crucial for modern software development, where quick integration and deployment are essential. They provide a framework for automating the software delivery process, ensuring that code changes are integrated, tested, and deployed efficiently and reliably.

One of the main patterns is the automation of the build and test process, which allows for instant feedback and quick identification of issues. This is complemented by the practice of CD, where automated tests are used to validate changes before they are deployed to production.

Another critical pattern is the use of microservices architecture, which enables independent deployment of service components, enhancing scalability and fault isolation.

Monitoring and logging are also essential, providing real-time insights into system performance and aiding in quick troubleshooting. By adopting these utility-based design patterns, organizations can achieve a more streamlined and robust CI/CD pipeline, leading to improved software quality and faster time-to-market.

We can look at this in the following graphical representation:

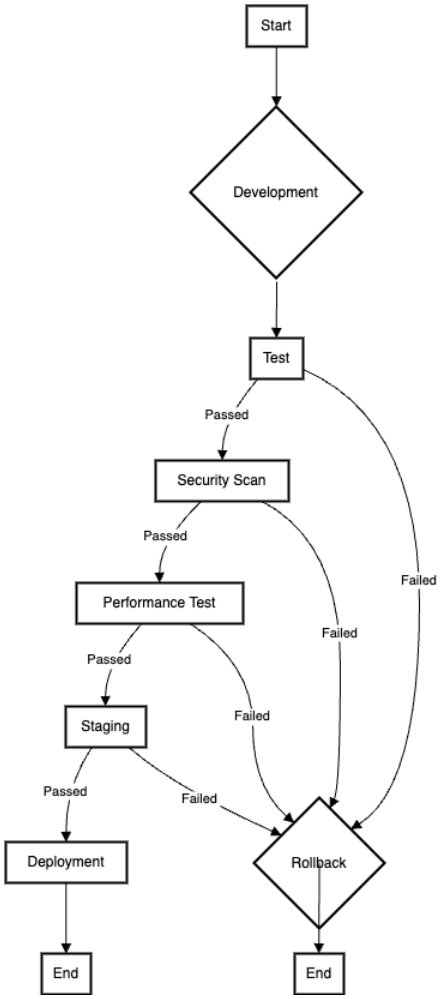

Figure 12.2 – Real-time utility pattern

In this diagram, we can see the following:

1. **Start**: The process begins.
2. **Development**: Developers write and commit code to the repository.
3. **Test**: Automated tests are run to ensure the code meets quality standards and doesn't break existing functionality.
4. **Security Scan**: The code is scanned for security vulnerabilities.
5. **Performance Test**: The code is tested for performance issues.

6. **Staging**: The code is deployed to a staging environment where it's tested in a production-like setting.
7. **Deployment**: If all tests pass, the code is deployed to the production environment.
8. **Rollback**: If any stage fails, the changes are rolled back.
9. **End**: The process ends until new code is committed.

So, why is this a real-time utility pattern? For the following reasons:

- **It's real time**: The CI/CD pipeline is triggered automatically whenever code is committed to the repository. This means that the pipeline provides real-time feedback to developers about the health of their code. If any stage of the pipeline fails, developers are alerted immediately so they can fix the issue.
- **It's utility-based**: The pipeline is designed to deliver a utility (such as a web application) from development to production. Each stage of the pipeline (Development, Testing, Security Scanning, Performance Testing, Staging, and Deployment) adds value to the utility and ensures that it meets the necessary quality standards before delivering it to the end-users.
- **It follows the CI/CD design pattern**: The pipeline follows the CI/CD design pattern. This pattern emphasizes the importance of automating the software delivery process, from integrating code changes to delivering ready-to-use software products.

In summary, a real-time utility-based CI/CD design pattern is a design pattern that uses automated, real-time processes to deliver a utility from development to production. The CI/CD pipeline examples provided follow this pattern. They are designed to automate the process of integrating code changes, testing the utility, and deploying it to a production environment, providing real-time feedback to developers throughout the process. This ensures that the utility is always in a deliverable state and meets the necessary quality standards. To have a clearer view of how this would look in the real world, we will have a look at an example.

Picture a fintech company's CI/CD pipeline that is designed as a real-time utility-based CI/CD design pattern. The fintech company is developing a web-based financial management platform. This platform provides users with real-time data on their financial transactions, investments, and budgets. The company wants to ensure that the platform is always up to date, secure, and performs well under high traffic.

The implementation of a real-time utility-based design pattern involves a range of steps to make it work, which are as follows:

1. **Development Stage (Code Integration)**:
    - Real-time: Developers continuously work on new features or bug fixes. Each time a developer commits code to the Git repository, a CI/CD pipeline is automatically triggered.
    - Utility-based: The focus is on delivering a functional utility—an updated version of the financial platform with the latest features or fixes.

2. **Testing Stage (Unit and Integration Testing)**:
    - Real-time: The pipeline immediately runs unit and integration tests after the code is committed. If any tests fail, developers are notified in real time so they can address issues immediately.
    - Utility-based: This stage ensures that the utility (the financial platform) remains functional and that new code does not break existing features.
3. **Security Scanning**:
    - Real-time: The pipeline automatically scans the code for security vulnerabilities as soon as the tests pass. Developers are alerted immediately if vulnerabilities are found, allowing them to fix issues before deployment.
    - Utility-based: Security scanning adds value to the utility by ensuring that the platform is secure and complies with industry standards.
4. **Performance Testing**:
    - Real-time: The pipeline then runs performance tests to simulate high traffic and ensure the platform can handle real-time data processing under load.
    - Utility-based: Performance testing ensures that the utility (financial platform) can deliver real-time data to users efficiently, even under peak usage.
5. **Staging Environment**:
    - Real-time: If all previous stages pass, the code is automatically deployed to a staging environment that mirrors the production environment.
    - Utility-based: This stage allows the team to verify that the utility behaves as expected in a real-world scenario before going live.
6. **Deployment to Production**:
    - Real-time: After final approval, the pipeline deploys the utility to the production environment. This deployment is automated and occurs in real time, to be sure users always have access to the latest version of the platform.
    - Utility-based: The deployment stage completes the delivery of the utility to end-users, ensuring that the financial platform is always in a deliverable state and meets the quality standards.

Each stage of the pipeline is automated, providing real-time feedback to developers and ensuring that the financial management platform (the utility) is always secure, performant, and ready for production. This pattern ensures that the platform is continuously improved and delivered to users in the most efficient way possible, meeting the high standards required for a real-time financial application.

Let's have a look at some implementation challenges in the next section.

The challenges of implementing real-time utility-based CI/CD systems

Implementing real-time utility-based CI/CD systems can be challenging. Organizations must navigate several obstacles to ensure smooth and efficient software delivery. Let's discuss these challenges one by one:

- One of the main challenges is managing performance issues. These can lead to slow page loading, sluggish server responses, or suboptimal memory usage. This affects the speed, responsiveness, stability, and scalability of the software.
- Effective team communication is also crucial. CI/CD involves various stakeholders, and any misalignment can lead to significant setbacks. Version control complexities arise when multiple developers merge changes into a shared repository. This necessitates robust conflict resolution strategies.
- Flawed automated testing can result in unreliable builds. This can happen if the test suites are not comprehensive or if they fail to cover critical paths in the code base. Security vulnerabilities are a constant concern. Therefore, it is necessary to integrate security checks into the CI/CD pipeline to detect and address issues early in the development cycle.
- Scaling up the testing infrastructure to match the pace of development and deployment is another hurdle. It requires substantial investment in resources and tooling. Complex debugging reports can overwhelm teams if not managed properly. They make it difficult to pinpoint the root causes of issues quickly.
- Integrating third-party services and tools often requires custom configurations. This can introduce dependencies that complicate the CI/CD process. Ensuring consistent environments across development, testing, and production stages is another challenge that needs attention. This is to prevent the *it works on my machine* syndrome.
- The adoption of microservices architectures adds complexity to the CI/CD pipeline. This is due to the increased number of components and interactions to manage. Balancing the speed of deployment with the need for thorough testing and quality assurance is a delicate task. It requires careful planning and execution.

To create a resilient and adaptable CI/CD pipeline, organizations must adopt best practices and use the right set of tools. This includes automating as much of the process as possible, implementing robust monitoring and alerting systems, and fostering a culture of continuous improvement. Feedback must be actively sought and acted upon. By addressing these common challenges, teams can realize the full potential of real-time CI/CD and maintain a competitive edge in the world of software development. Professionals can refer to comprehensive guides and discussions available in the field for a deeper understanding of these challenges and their solutions.

Perhaps it's better to look at a real-life scenario to see the implications discussed in this section.

A real-life scenario of implementation challenges

Consider the scenario of a financial services company that decided to improve its online banking platform by adopting a microservices architecture. The company implemented a real-time CI/CD system to streamline its software delivery process. However, the development team faced several challenges when transitioning to this system. The next section describes the challenges faced, but also the solutions to these challenges.

The challenges

Initially, the team experienced performance issues, with the new services responding slowly during high-traffic periods. This led to customer complaints about the online banking experience. To address this, the team implemented performance monitoring tools to identify and resolve bottlenecks, optimizing the system for better load handling.

Communication hurdles emerged as the project involved multiple teams, including development, operations, and quality assurance. Misunderstandings regarding deployment schedules and feature releases caused delays. The company addressed these issues by establishing clear communication protocols and utilizing collaboration tools to keep all stakeholders informed and aligned.

Version control complexities arose when developers worked on different microservices concurrently. Merging code changes into the main repository led to conflicts that disrupted the development workflow.

Adopting solutions to the challenges

The team adopted a feature-branching strategy with peer reviews to manage code integration more effectively. This kind of strategy is graphically represented in the following diagram:

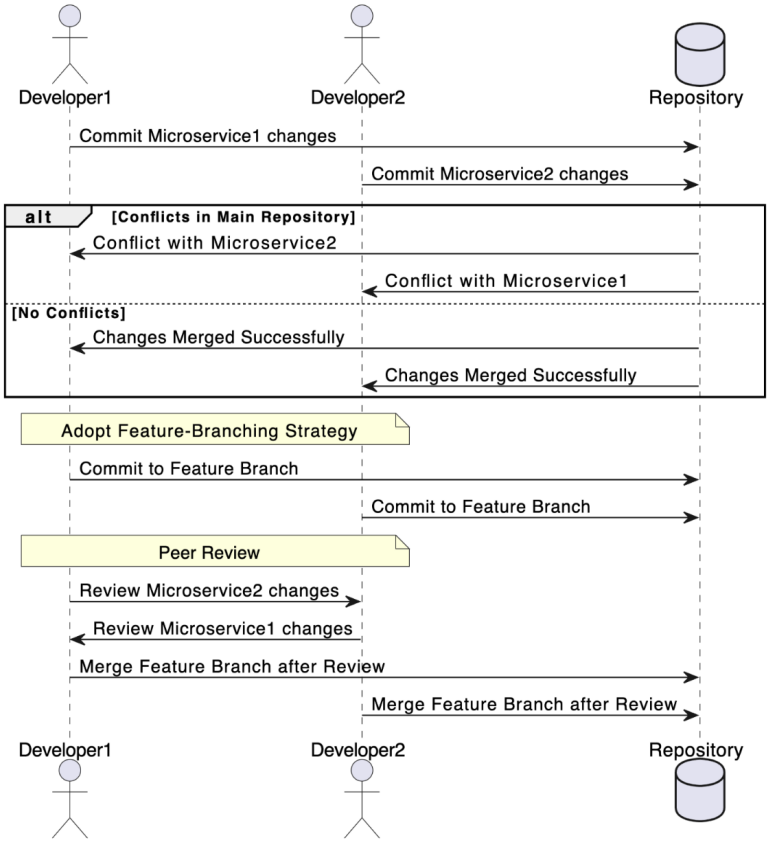

Figure 12.3 – Implemented feature branch strategy

In this flow, the developers are working on multiple microservices as a part of a larger project. As the development progresses, some of the team members start experiencing conflicts due to multiple code changes happening simultaneously. In order to manage this situation effectively, the team decides to adopt a feature-branching strategy with peer reviews.

With this approach, each developer works on a separate branch, implementing a specific feature or fixing a bug. Once the changes are ready, the developers create a pull request, which triggers a peer review process. Other members of the team review the code changes and provide feedback. The pull request is merged into the main branch only after it has been approved by the reviewers.

This strategy helps to minimize conflicts and ensures that the code changes are thoroughly reviewed before being incorporated into the main branch. It also helps the team to track the progress of each feature and identify any issues early in the development cycle.

Another challenge to be solved

The automated testing framework initially failed to catch critical bugs, resulting in unstable builds being deployed to production. The team revised their testing strategy, ensuring that test suites were comprehensive and covered all critical paths. They also integrated security checks into the CI/CD pipeline, which helped with the early detection and remediation of vulnerabilities, reinforcing the platform's security posture.

Scaling the testing infrastructure to keep up with the rapid pace of development proved challenging. The company invested in cloud-based testing services that offered scalability and flexibility. They also streamlined debugging by implementing centralized logging and monitoring, which simplified the identification of issues. The following flow shows the issues and measurements taken by a team encountering these problems:

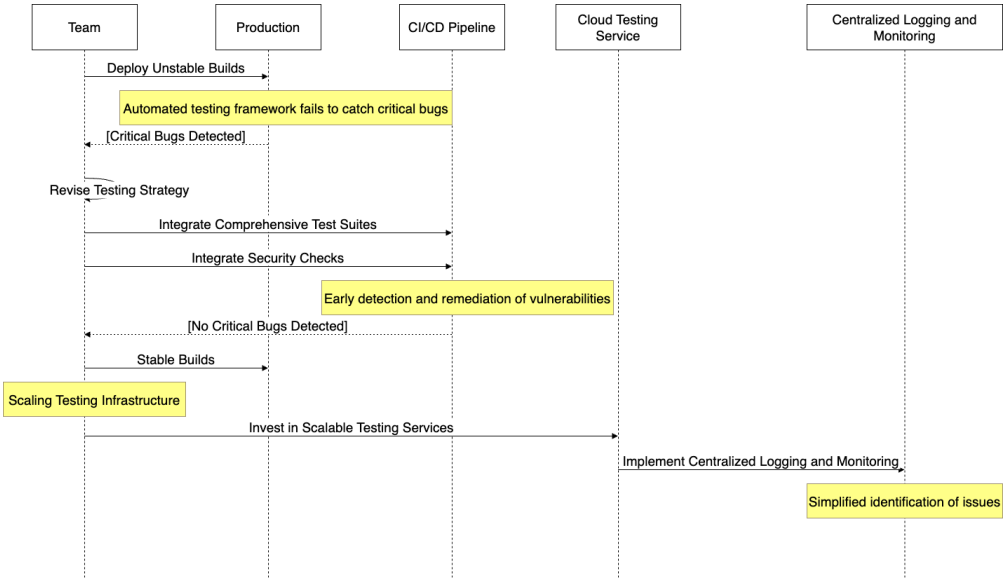

Figure 12.4 – Problems, issues, and measurements taken

The following is happening here:

- The team deploys unstable builds to the production environment. There is an indication that the automated testing framework fails to catch critical bugs.
- When these critical bugs are detected, the following occurs:
  - Production notifies the team about the unstable builds.
  - The team revises its testing strategy.
  - The team integrates comprehensive test suites and security checks into the CI/CD pipeline.

- If no critical bugs are detected, the following occurs:
  - Production receives stable builds from the team.
  - The team invests in scalable testing services provided by the cloud testing service.
  - The team also implements centralized logging and monitoring for simplified issue identification.

Figure 12.4 illustrates the interactions between different components during the software development and deployment process. It shows the importance of testing, bug detection, and monitoring for maintaining a reliable production environment.

Integrating third-party services, such as payment gateways and credit score APIs, required custom configurations that introduced dependencies. The team created a shared library of custom connectors to standardize these integrations, reducing complexity.

Ensuring consistent environments across the development, testing, and production stages was crucial for preventing deployment issues. The company utilized containerization and **Infrastructure as Code** (**IaC**) to maintain consistency and enable smooth transitions between environments.

The adoption of a microservices architecture introduced additional complexity, with more components and interactions to manage. The team implemented a service mesh to manage service-to-service communication, providing enhanced visibility and control over the microservices.

Through strategic planning, investment in robust infrastructure, and the adoption of a culture of continuous improvement and collaboration, the financial services company successfully overcame the challenges of implementing a real-time CI/CD system. This resulted in a more resilient, scalable, and efficient online banking platform that met the evolving needs of its customers. The real-life application of CI/CD in this scenario demonstrates the tangible benefits and the critical importance of addressing the multifaceted challenges associated with its implementation.

Now that we have looked at some real-life scenarios of implementations, along with the associated challenges, it's time to look ahead to further evolution in CI/CD.

Further evolution and roadmap considerations

Software development, along with CI/CD, has established itself as a cornerstone of modern DevOps practices, streamlining the process from code commit to production deployment. Advanced CI/CD design patterns and use cases are continuously evolving to accommodate the growing complexity of software systems and the need for faster delivery cycles. As organizations strive for efficiency and agility, it's crucial to consider the integration of machine learning and generative AI into CI/CD pipelines, which can predict potential issues and automate solutions, enhancing the robustness of the deployment process. Furthermore, near-real-time CI/CD design patterns are gaining traction, enabling organizations to react swiftly to changes in the world or in user behavior. We have to adopt new ways, which we will discuss in the next section.

Adoption of new architectures and technologies

When contemplating the future roadmap of CI/CD, several considerations come to the forefront. The adoption of microservices architecture, containerization, and serverless computing has necessitated the evolution of CI/CD patterns to support these paradigms effectively. This includes the development of dynamic environment provisioning and IaC to manage complex deployments. Additionally, security has become a paramount concern, leading to the integration of DevSecOps practices within CI/CD pipelines to ensure that security checks and balances are automated and embedded throughout the software delivery process.

Another important topic is scalability, which we will have a look at next.

Scalability

The roadmap for CI/CD must also take into account the scalability challenges posed by large-scale distributed systems. This involves implementing scaling patterns that can handle the increased load and complexity without compromising performance or reliability. Moreover, the CI/CD pipeline itself must be resilient, capable of self-healing in the event of failures, and designed to provide feedback loops that facilitate continuous improvement.

Let's have a look at how this fits into practical implementations.

Practical implementation

In terms of practical implementation, organizations are encouraged to automate as much of the build, testing, and deployment processes as possible. This includes automating the integration of code changes, managing feature development through feature flags, and ensuring that unit tests and additional tests at each stage of the pipeline are performed to catch issues early. The goal is to establish a CI/CD pipeline that acts as a model for the **software development lifecycle** (**SDLC**), providing automated governance and risk management.

To successfully navigate the evolution of CI/CD, it's essential to maintain a focus on the goals of CI/CD. This means fostering a culture of collaboration between development, assurance, operations, security, and finance teams, and leveraging automation to reduce lead times and improve the reliability of software releases. As CI/CD continues to evolve, staying up to date with best practices and emerging trends will be key to ensuring that the roadmap for CI/CD aligns with the strategic objectives of the organization and the ever-changing landscape of software development.

Looking into the near future, generative AI may become more and more important in the CI/CD world, so we will take a closer look at what this will look like.

CI/CD in the near future – generative AI

If we look for an example of what CI/CD will look like in the near future, the best thing to do is to show it in a flow in which some AI and tooling are integrated, as shown in this figure:

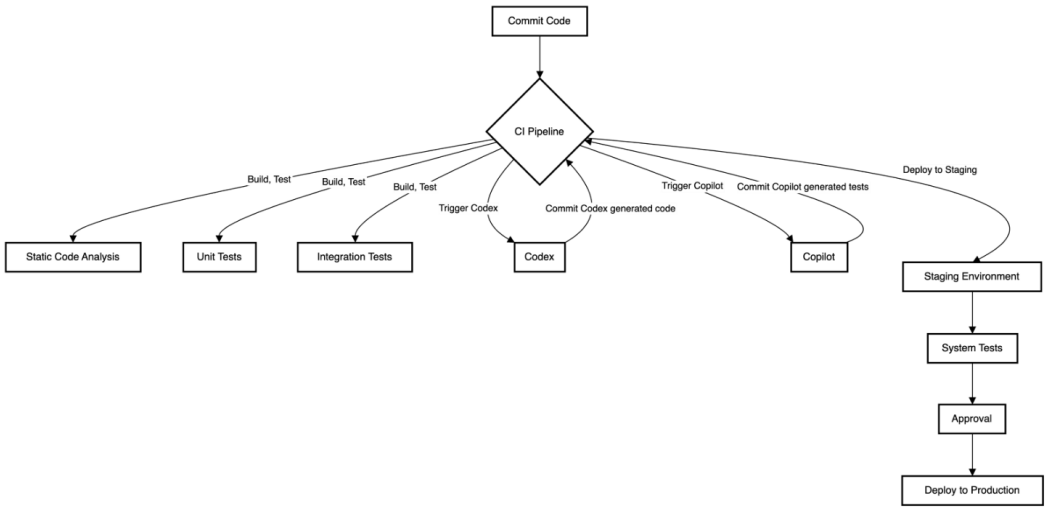

Figure 12.5 – Built-in generative AI in a CI/CD pipeline

In this diagram, the following process is shown, with an example of using Codex and Microsoft Copilot:

1. The developer commits code, which triggers the CI pipeline.
2. The CI pipeline performs various checks and tests such as static code analysis, unit tests, and integration tests.
3. The pipeline then triggers Codex to generate new code.
4. Codex commits the generated code back to the pipeline, which then re-runs with the new code.
5. The pipeline then triggers Copilot to generate new tests.
6. Copilot commits the generated tests back to the pipeline, which then re-runs with the new tests.
7. If all tests pass, the code is deployed to a staging environment where system tests are run.

After successful system tests and manual approval, the code is finally deployed to production.

Integrating generative AI into CI/CD pipelines requires a transformative shift in software development, offering a roadmap for innovation and efficiency. Advanced CI/CD design patterns now increasingly incorporate generative AI to automate and optimize various stages of the development lifecycle. For instance, layered caching strategies are being employed to reduce latency and cost while improving the precision of AI models through fine-tuning based on accumulated data. Multiplexing AI agents,

each specialized in different tasks, can work in tandem to address complex queries, thereby enhancing the problem-solving capabilities of CI/CD systems.

Moreover, the use of generative AI in CI/CD is not just limited to operational efficiency but also extends to strategic planning and roadmap considerations. It necessitates a forward-thinking approach, where the potential of AI to innovate development practices is balanced against the need for solid governance and ethical considerations. As generative AI evolves, it is crucial to anticipate the trajectory of this technology and its implications for software development. This includes preparing for the integration of AI at various stages of the CI/CD pipeline, from code commits to static scanning, ensuring that AI tools are leveraged to enhance developer workflows without compromising on code quality or security.

To be ready for the near future, the journey and roadmap to embed generative AI into CI/CD can be useful, so we will look a bit closer to it.

Generative AI – journey and roadmap

The roadmap for generative AI in CI/CD must also consider the scalability of AI solutions and their adaptability to different development environments. As highlighted in recent industry insights, AI-engineered CI can streamline code branching, commits, and merges, reducing merge conflicts and suggesting optimal strategies for code management. Static scanning, powered by AI, can improve accuracy and reduce false positives, offering potential fixes for identified issues. These advancements underscore the need for a strategic guide that helps enterprises use generative AI effectively, aligning with their long-term goals and industry standards.

Here is an example of a possible roadmap that could apply to an organization that started using AI and wants to incorporate it into its CI/CD processes. The time of 5 years is just picked as an example.

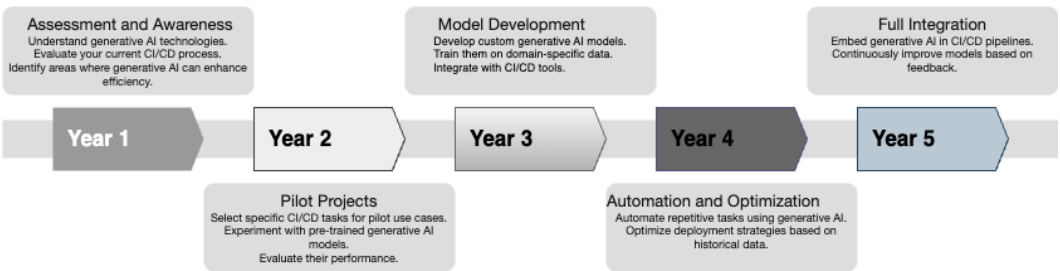

Figure 12.6 – Five-year roadmap of incorporating AI into CI/CD

This is a high-level example of a possible roadmap but, of course, it will vary per organization.

In conclusion, the further evolution of CI/CD design patterns with generative AI at the helm is poised to redefine the landscape of software development. It offers a lot of use cases that promise increased efficiency, creativity, and problem-solving processes. However, it also demands careful consideration of the roadmap for AI integration, ensuring that as we harness the power of generative AI, we do so with foresight and responsibility, paving the way for a future where AI and human ingenuity coalesce

to push the boundaries of what's possible in software engineering. The journey towards this future is marked by continuous learning, adaptation, and innovation, guided by the principles of ethical AI use and sustainable development practices.

Summary

In this chapter, we looked into advanced design patterns that integrate machine learning, generative AI, and real-time utility-based CI/CD design patterns, offering cutting-edge solutions for modern software delivery pipelines. With a focus on advanced techniques and real-time provisioning, this chapter explored the forefront of CI/CD evolution.

Emerging technologies, trends from the industry, and evolving best practices will drive the future evolution of CI/CD design patterns. Considerations for future roadmap include advancements in AI-driven automation, the integration of emerging technologies such as blockchain and edge computing, and the adoption of DevSecOps principles for enhanced security and compliance. By staying up to date on these developments and proactively adapting to changing landscapes, organizations can continue to innovate and optimize their CI/CD pipelines for maximum efficiency and effectiveness.

In the next chapter, we will look not at patterns, but antipatterns.

13

Exploring Anti-Patterns for CI/CD Design Pattern Deployments

In this chapter, we will highlight the anti-patterns of CI/CD design patterns. Anti-patterns seem to be the right choice at first but have long-term consequences to consider. We will discuss some of the key anti-patterns with long-term implications. We will also discuss how to find the best-fit design pattern and its considerations. Lastly, we will connect the dots with platform engineering and address why CI/CD design patterns are vital, followed by how the design pattern can be considered foundational.

This chapter offers you valuable insights by identifying common pitfalls in CI/CD design patterns, known as **anti-patterns**. By understanding these anti-patterns, you can avoid the long-term negative consequences they entail. The chapter will also guide you in selecting the most suitable design patterns for their needs, taking into account various considerations. Furthermore, it emphasizes the importance of CI/CD design patterns within platform engineering, illustrating how these patterns form the foundation for successful development and operations strategies.

In this chapter, we will cover the following topics:

- Anti-patterns for CI/CD design patterns
- Best fit design patterns – considerations
- CI/CD design pattern and platform engineering
- Exploring a case study of CI/CD integration in platform engineering

Anti-patterns for CI/CD design patterns

In the previous chapters, we talked about all the good things that happen when using CI/CD design patterns. But in this chapter, we will go through some anti-patterns. But why would we do such a thing?

Discussing anti-patterns can be beneficial for several reasons, such as the following:

- **Awareness of ineffective solutions**: Anti-patterns are common solutions to problems that are ineffective and can cause more problems than they solve. By discussing them, we can raise awareness about these ineffective solutions and prevent their use.
- **Understanding consequences**: Anti-patterns often seem to work in the short term, but they can have negative long-term consequences. Discussing them helps us understand these consequences and consider the larger context.
- **Improving code quality**: Many anti-patterns relate to poor coding practices that lead to problems such as increased complexity and reduced maintainability. Discussing these can help improve code quality.
- **Enhancing communication**: Anti-patterns provide a common language for discussing problems. This can improve communication within a team or organization.
- **Preventing recurrence**: Once an anti-pattern is recognized and discussed, it's easier to avoid in the future.
- **Promoting best practices**: Discussing anti-patterns naturally leads to a discussion of best practices, as the opposite of an anti-pattern is a pattern or best practice.

In fact, the goal is not just to avoid anti-patterns but to promote patterns and best practices that lead to effective and efficient solutions. But let us first explore what anti-patterns are.

What is an anti-pattern?

The term *anti-pattern* may sound a little bit debatable; one could say, either you have a pattern, or it's not a pattern. But that's not entirely true. They are still commonly used practices; however, they are not effective and might end up bad in the end.

Anti-patterns are essentially common but ineffective solutions to recurring problems within software development and other fields. They are practices that appear to be beneficial on the surface but lead to more issues than they resolve. The term already heads back to 1995, introduced by Andrew Koenig, inspired by the concept of design patterns in software engineering, which are considered to be reliable and effective solutions. Unlike design patterns, anti-patterns are not best practices and are generally to be avoided as they often result in poor outcomes such as increased complexity, decreased maintainability, and overall inefficiency.

Examples of anti-patterns in programming include *spaghetti code*, where code is tangled and chaotic, and *god objects*, where a single class or object is overloaded with responsibilities. Recognizing and avoiding anti-patterns is crucial for developing robust, scalable, and maintainable software.

So, let's have a look at what these anti-patterns are in relation to CI/CD, and what they would look like. The most common anti-patterns are as follows:

- Lack of proper pipeline modeling
- Crappy or incomplete automation in testing
- Ignoring pipeline failures
- Poor monitoring/observability
- **Single point of failure** (**SPOF**) in CI/CD infrastructure
- Bad security pipeline integration
- Static pipeline configuration

Let's go through them one by one!

Lack of proper pipeline modeling

An important fact about the lack of proper pipeline modeling is that it is a significant anti-pattern that can lead to a host of inefficiencies and challenges. This anti-pattern manifests when teams fail to create a clear and comprehensive model of their deployment pipeline, resulting in a lack of visibility into the process and an inability to identify bottlenecks or areas for improvement. Without a proper model, it becomes difficult to understand the flow of work through the pipeline, which results in bad or misallocation of resources and efforts that do not address the most critical issues.

The following figure represents a badly modeled pipeline:

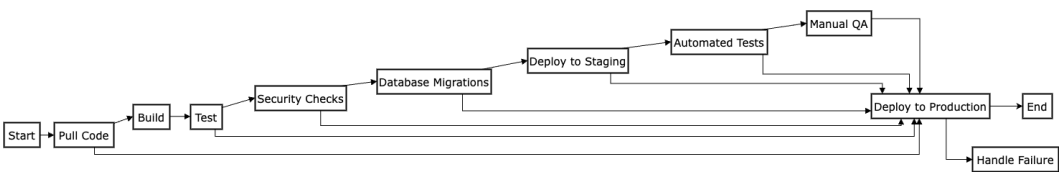

Figure 13.1 – A badly modeled pipeline

You can see the following stages in this pipeline:

1. The pipeline starts.
2. *Pull Code* is the first step where the latest code is pulled from the repository.
3. *Build* is where the application is built.
4. *Test* is where unit tests are run.

5. *Security Checks* is where security checks are performed to ensure the application is secure.
6. *Database Migrations* is where database migrations are performed to update the database schema.
7. *Deploy to Staging* is where the application is deployed to a staging environment.
8. *Automated Tests* is where automated tests are run against the staging environment.
9. *Manual QA* is where manual quality assurance checks are performed.
10. *Deploy to Production* is where the application is deployed to the production environment.
11. The pipeline ends.
12. Finally, *Handle Failure* is where failures in the pipeline are handled.

After each step, the pipeline deploys directly to production (e.g., **Pull Code | Deploy to Production**, **Test | Deploy to Production**, etc.). This skips important steps such as building, testing, security checks, database migrations, deploying to staging, automated tests, and manual QA.

Failures are only handled after deploying to production (**Deploy to Production | Handle Failure**). Failures should be handled immediately after they occur to prevent broken code from being deployed to production.

These shortcuts can lead to untested or broken code being deployed to production, which is a major anti-pattern in CI/CD pipelines. It's important to ensure that all steps are properly sequenced, that no steps are skipped, and that failures are properly handled.

To overcome this anti-pattern, it is essential to implement *systematic pipeline modeling*. This involves mapping out each stage of the pipeline and measuring key performance indicators such as lead times, process times, and the percentage of completed and accurate tasks. By doing so, teams can gain an end-to-end view of their pipeline, allowing them to identify and prioritize the removal of bottlenecks. Furthermore, it is crucial to balance investments between improving the pipeline and developing new features. Often, the focus on feature development overshadows the need to optimize the pipeline, which can lead to a decline in both velocity and quality over time.

Poor or incomplete automation in testing

Test automation is a cornerstone for ensuring the rapid and reliable delivery of software. However, when automation is poorly implemented or incomplete, it can lead to significant types of anti-patterns that undermine the efficiency and effectiveness of the development pipeline. The anti-patterns we can identify are the following:

- One such anti-pattern is the reliance on manual processes that should be automated, leading to bottlenecks and errors.
- Another is the creation of automated tests that are flaky or non-deterministic, which can cause false positives or negatives in the testing results.

- Additionally, failing to maintain and update test suites to reflect changes in the application can result in outdated tests that do not catch new issues.
- It's also crucial to avoid an over-reliance on end-to-end tests, which can be time-consuming and fragile, instead of a balanced approach that includes unit and integration tests.
- Furthermore, neglecting to incorporate adequate logging and monitoring within the automated tests can leave teams blind to the root causes of failures.

To overcome these anti-patterns, here are some of the best practices to consider:

- Suggest a comprehensive strategy that includes writing reliable and maintainable automated tests, ensuring tests are up to date with the application's evolution, and maintaining a balanced test suite with a mix of test types.
- Regularly performing load and performance testing, as well as early integration testing, can also help in identifying issues before they escalate.

By adhering to these practices, teams can leverage the full potential of CI/CD to deliver high-quality software at speed.

Improving your testing automation strategy involves a multifaceted approach that begins with establishing clear and achievable goals that align with your organization's objectives, such as enhancing test coverage or accelerating release cycles. Selecting the right tools is crucial; consider factors such as application architecture, team expertise, and scalability requirements. For instance, tools such as Selenium and TestComplete offer capabilities for web automation, while Appium caters to mobile testing needs.

Let's have a look at an example of a bad test flow:

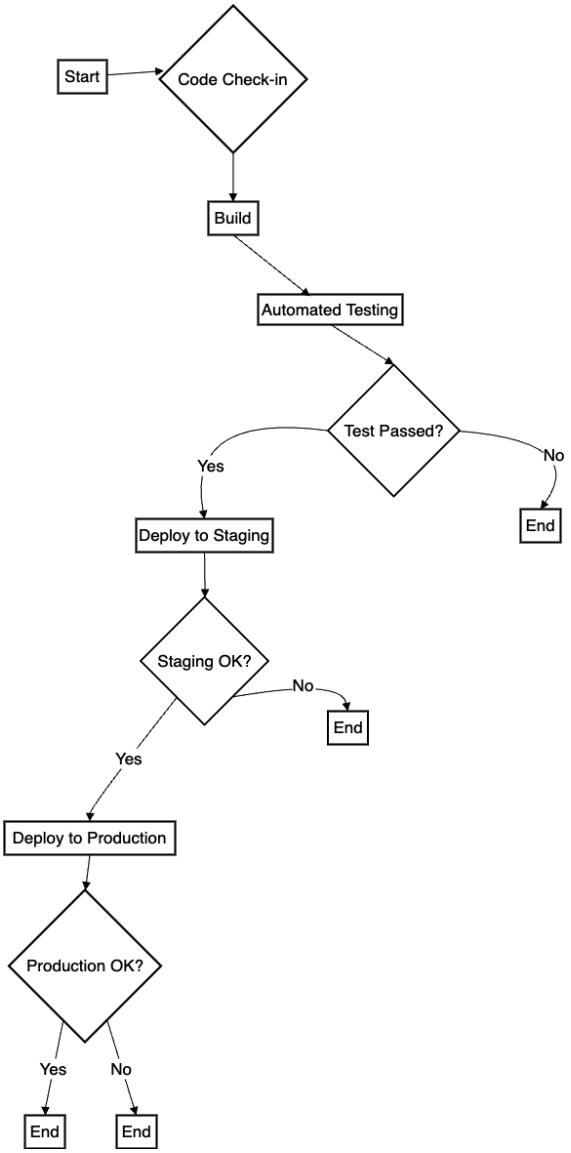

Figure 13.2 – Bad and incomplete automated test flow

This picture tells us the following:

1. The process starts with a code check-in.
2. The code is then built and goes through automated testing.

3. If the tests pass, the code is deployed to a staging environment.
4. If the staging environment is OK, the code is deployed to production.
5. If the production environment is not OK, the process ends without any rollback mechanism.

This is bad practice because it lacks proper error handling and rollback mechanisms. If something goes wrong in the production environment, there's no automated way to reverse the changes, which could potentially bring down the entire production environment. It's always recommended to include proper error handling and rollback mechanisms in your CI/CD pipeline to ensure the stability and reliability of your software. A thorough test suitability assessment will help you determine which tests are ideal for automation. Typically, repetitive and time-consuming tests, or those prone to human error, are prime candidates. Conversely, tests that require human intuition or have frequently changing requirements might be better suited for manual testing.

Setting up a stable and reliable test environment is essential for consistent automation execution. This includes configuring browsers, devices, operating systems, and network settings to replicate end-user conditions. Platforms such as TestGrid can assist in managing these environments across various combinations of browsers and operating systems.

Your test data requirements should be clearly defined, ensuring the data supports both current and future automated testing scenarios. Techniques such as mocking or property-based testing can help you test realistic production data while maintaining privacy and security.

Integrating your automated tests with a CI/CD pipeline ensures that testing becomes an inherent part of the development workflow, with tests triggered automatically at various stages. This integration provides detailed feedback on the quality and functionality of the code, helping to identify issues early in the development cycle.

Prioritizing the automation of high-priority scenarios based on feature criticality and impact on the application is also important. A balanced test suite that includes unit, integration, and end-to-end tests can prevent over-reliance on any single type of test.

Continuously evaluating and updating your test automation strategy is vital to keep pace with the evolving landscape. This includes monitoring test results to gain insights into the effectiveness of your testing efforts and making necessary adjustments.

Investing in test maintenance is equally important; this involves regularly updating test cases to reflect changes in the application and ensuring that the automation suite remains relevant and effective.

Following the test automation pyramid (see also the *Implementing testing in CI/CD* section in *Chapter 1*) as a model can guide you in optimizing your test strategy, ensuring that you have a solid foundation of unit tests, a middle layer of integration tests, and a smaller top layer of UI tests.

Ignoring pipeline failures

Handling pipeline failures is crucial to maintain the workflow's efficiency. Ignoring pipeline failures can be a strategic decision, especially when certain steps are not critical for the overall success of the build. Let's investigate two use cases, one with the popular tool Jenkins, and one from the Azure cloud. For example, in Jenkins, you can use a try-catch block within a script step to catch errors and prevent the pipeline from failing. Alternatively, you can employ the `catchError` step to allow the pipeline to continue even if a particular stage fails.

Let's have a look at a code snippet for this step:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                script {
                    try {
                        sh '''
                            echo "Building..."
                            exit 1
                        '''
                    } catch(Exception e) {
                        echo "Build failed with error: ${e.
getMessage()}"
                    }
                }
            }
        }
    }
}
```

In this example, if an error occurs during the execution of the build steps, the error will be caught, and the error message will be printed to the console. However, the pipeline will not fail.

Alternatively, you can use the `catchError` step to allow the pipeline to continue even if a particular stage fails, which you can see in this example:

```
pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                catchError(buildResult: 'SUCCESS', stageResult:
'FAILURE') {
```

```
                    // Your build steps go here
                }
            }
        }
    }
}
```

In this example, if an error occurs during the execution of the build steps, the stage will be marked as failed. However, the pipeline will continue to execute, and the build result will be marked as successful. This approach can be useful if you want to ensure that a specific failure does not prevent subsequent stages from running.

Similarly, in Azure Pipelines, you can set the `continueOnError` flag to `true` for specific tasks, allowing the rest of the job to proceed even if that task fails. This approach ensures that non-critical failures do not impede the progress of the pipeline while still allowing for visibility into which parts of the process may need attention.

Here's an example of how you can set the `continueOnError` flag to true for a specific task in an Azure Pipelines YAML file:

```
trigger:
- master

pool:
  vmImage: 'ubuntu-latest'

steps:
- script: echo "Building..."
  displayName: 'Build'

- script: exit 1
  displayName: 'Test'
  continueOnError: true

- script: echo "Deploying..."
  displayName: 'Deploy'
```

The pipeline comprises three steps: `Build`, `Test`, and `Deploy`. The `continueOnError` flag is set to `true` for the `Test` step. This means that even if the `Test` step fails (which it will, because of `exit 1`), the rest of the job will proceed. The `Deploy` step will run regardless of the outcome of the `Test` step.

This approach ensures that non-critical failures do not impede the progress of the pipeline while still providing visibility into which parts of the process may need attention. It strikes a balance between resilience and accountability, ensuring that the pipeline remains robust yet informative.

It's a balance between resilience and accountability, ensuring that the pipeline remains robust yet informative. Implementing such strategies requires careful consideration of the pipeline's design and the criticality of each task to the overall project goals.

Poor monitoring/observability

These are essential for maintaining the efficiency of the software delivery process. To achieve effective observability, it's important to integrate comprehensive monitoring tools that offer insights into the performance of the CI/CD pipeline. This involves tracking key metrics such as build time, test results, deployment frequency, and resource utilization. Tools such as InfluxDB and Grafana can be set up to create dashboards that display these metrics, making it easier for teams to interpret and respond to the data.

Additionally, implementing logging and tracing can aid in quickly pinpointing issues, thereby reducing downtime and improving the reliability of the pipeline. For a more advanced setup, the OpenTelemetry Collector can be deployed to enable low-latency communication between CI/CD tools and the monitoring system, allowing for more detailed and real-time observability.

Inadequate monitoring and observability in CI/CD pipelines can cause various issues, affecting software delivery and quality assurance. Consider the following examples:

- Slow deployments resulting from inefficient CI/CD operations can delay the release of performance bug fixes, leading to critical issues for end users.
- Incomplete or inadequate testing, caused by slow builds or disorganized handoffs between developers and testers, can lead to deploying untested releases or delaying updates, both of which can harm the user experience.
- Limited testing coverage is another consequence, as releases may not be tested against a comprehensive set of configurations, risking bad application performance.
- Poor credential management, including weak secret management or insecure storage, can lead to security breaches within the pipeline.
- Lack of visibility into the CI/CD process can lead to technical debt, making it challenging to identify and address inefficient processes.

These examples emphasize the importance of robust monitoring and observability practices to uphold a healthy and efficient CI/CD pipeline.

SPOF in CI/CD infrastructure

Avoiding SPOFs in CI/CD infrastructure is crucial as they can cause a complete system shutdown if a single component fails. It's important to identify and address SPOFs to ensure the reliability and stability of CI/CD pipelines. During the system design phase, conducting a comprehensive business impact analysis and risk assessment is essential to identify potential SPOFs, which can often be found in non-redundant hardware components.

However, it's not only hardware that presents risks; services, personnel, and specific expertise can also become SPOFs if not properly managed. For example, relying on a single server or a single network switch for all servers can result in significant downtime and operational losses. Similarly, depending solely on one internet service provider or a single expert for a critical application can put business continuity at risk.

To mitigate these risks, it's important to develop a comprehensive strategy that includes redundancy, regular reviews, and a transparent culture where team members feel comfortable identifying and reporting potential SPOFs without fear of reprisal. This approach helps organizations build more resilient CI/CD infrastructures that can withstand individual component failures without compromising overall system integrity.

Bad security pipeline integration

Remember to integrate security into CI/CD pipelines, as it's a critical aspect of modern software development. Integrating security into CI/CD pipelines is a critical aspect of modern software development, ensuring that the code is secure at every stage from development to deployment.

A robust security pipeline integration should include automated security checks, such as static code analysis and dynamic application security testing, to detect and mitigate vulnerabilities early.

Additionally, it's essential to manage access controls rigorously to prevent unauthorized code changes or access to sensitive data. Implementing a *shift-left* approach, where security is integrated from the beginning of the development process, can reduce the risks associated with bad security pipeline integration. This ensures that the code remains secure throughout the development and deployment stages. A robust security pipeline integration should include automated security checks, such as static code analysis and dynamic application security testing, to identify and mitigate vulnerabilities early. Managing access controls rigorously is also essential to prevent unauthorized code changes or access to sensitive data. By implementing a shift-left approach, where security is integrated from the beginning of the development process, you can significantly reduce the risks associated with poor security pipeline integration.

Let's have a look at how this is integrated in the following figure:

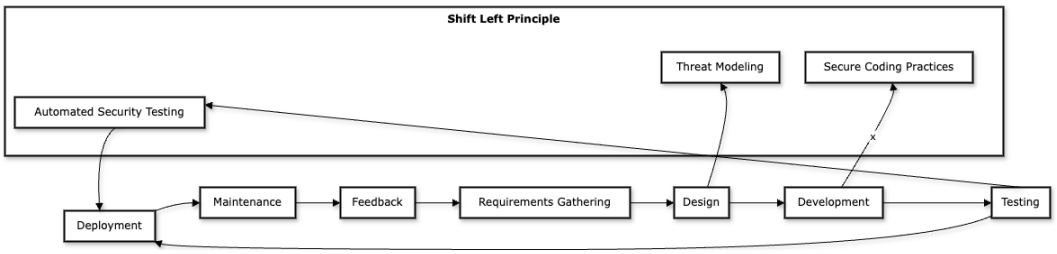

Figure 13.3 – The shift-left security principle

In this diagram, the principle is shown in the development life cycle.

The traditional software development life cycle starts with *Requirements Gathering*, followed by *Design*, *Development*, *Testing*, *Deployment*, and *Maintenance*. Feedback from the *Maintenance* phase is then used to inform the next round of requirements gathering.

The **shift-left security principle** is represented by the *Threat Modeling*, *Secure Coding Practices*, and *Automated Security Testing* stages. These stages are inserted between the *Design* and *Deployment* stages of the traditional life cycle. Let's have a look at those being inserted:

- Threat modeling involves identifying potential threats and vulnerabilities during the *Design* stage
- Secure coding practices are implemented during the *Development* stage to prevent security vulnerabilities in the code
- Automated security testing is performed during the *Testing* stage to catch any security issues before deployment

When we look at software development and security issues, we can identify vulnerabilities found in developed applications, such as the following:

- **Broken access control**: Users can perform actions without proper restrictions. For example, an employee accesses admin privileges.
- **Cryptographic failures**: Inadequate protection of sensitive data, whether stored or transmitted. For example, unencrypted user passwords.
- **Injection flaws**: Untrusted data causes command or query execution. For example, SQL injection leads to unauthorized database access.
- **Insecure design**: Lack of effective security controls. For example, a web application without input validation.
- **Security misconfiguration**: Default security settings lead to vulnerabilities. For example, unchanged default application passwords.
- **Outdated components**: Using old software exposes systems to risks. For example, unpatched servers are susceptible to exploitation.
- **Identification and authentication failures**: Systems fail to verify user identities. For example, brute force attacks on weak authentication.
- **Software and data integrity failures**: Compromised code and data integrity. For example, malware inserted in a software update.
- **Security logging and monitoring failures**: Inadequate breach detection. For example, undetected intrusion due to lack of monitoring.

Static pipeline configuration

Creating a static pipeline configuration is crucial in DevOps, especially for static web applications. This process involves setting up a pipeline to automate building, testing, and deploying applications consistently and reliably.

For example, the Azure Static Web Apps service uses a YAML configuration file to handle the build and deployment process via GitHub Actions or Azure Pipelines. This file specifies parameters such as the source code location for the Frontend app and API, build commands, and output locations. The static nature of the pipeline means that the configuration is predefined and doesn't change during pipeline execution. This helps ensure that the deployment process is repeatable and predictable, essential for maintaining application stability and reliability in production. Additionally, static pipelines are often easier to secure and manage since their behavior is well defined and less prone to variations that could introduce errors or vulnerabilities. Here is a graphical representation of a static pipeline configuration:

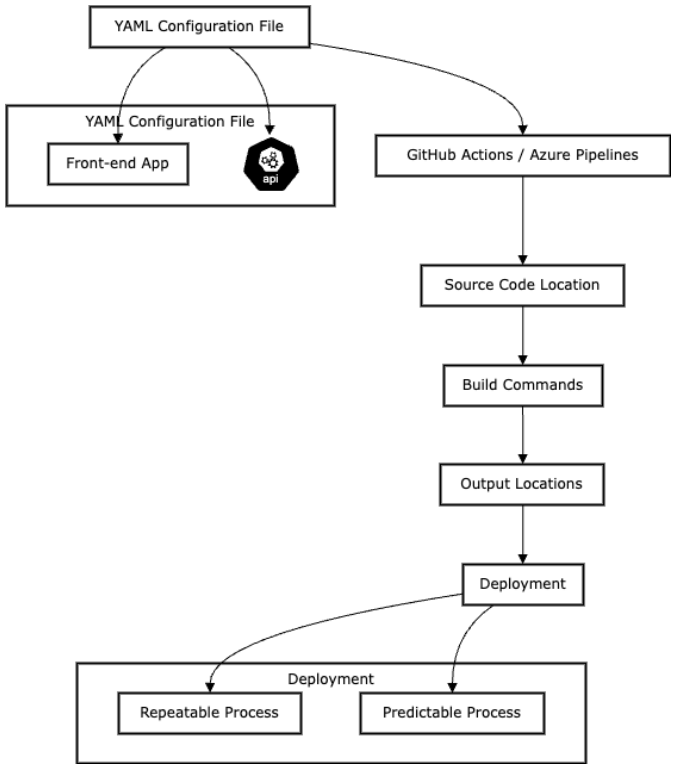

Figure 13.4 – Static pipeline configuration

This pipeline is considered to be static because the YAML configuration file specifies parameters such as the source code location for the Frontend app and API, build commands, and output locations. These parameters are set before the pipeline runs and do not change during the execution of the pipeline.

However, the use of static pipeline configuration is considered an anti-pattern in modern software development. This is because it can lead to several issues that hinder the agility and scalability of the development process. Static configuration is inflexible and does not adapt well to changes, which is common in dynamic development environments. This rigidity can result in a pipeline that is difficult to update or modify without extensive manual intervention, leading to bottlenecks and delays in the delivery process.

Furthermore, it can cause problems with collaboration and innovation, as it may not easily accommodate new tools or practices that teams wish to integrate into their workflow. To maintain a competitive edge and ensure continuous integration and delivery, it's crucial to adopt configurations that are dynamic and can evolve with the project's needs. This approach promotes a more efficient, scalable, and collaborative environment, which is essential for successful DevOps practices.

Now that we have looked at a variety of anti-patterns, we are going to think about how to consider what would be the best fit in several use cases, to result in not ending up having an anti-pattern.

Best fit design patterns – considerations

We have learned that anti-patterns are common pitfalls that can lead to inefficient and error-prone practices within CI/CD processes. Anti-patterns often arise from a need for a better understanding of CI/CD principles or from short-term solutions that become detrimental in the long run.

When considering the best fit for CI/CD, it is essential to evaluate the project's specific needs and context. There are several factors involved, such as team size, project complexity, and the technology stack influencing the choice of design patterns. For example, a microservices architecture might benefit from a pipeline that emphasizes independent deployments, while a monolithic application might require a different approach.

Moreover, the continuous evolution of software development practices necessitates regular reviews and updates of CI/CD patterns. What may be considered a best practice today could evolve into an anti-pattern tomorrow as new tools and methodologies emerge. Therefore, it is important for teams to stay informed and adaptable, regularly revisiting their CI/CD strategies to align with current best practices.

Let's put some considerations together to give you a clear overview. Implementing the right design patterns in your CI/CD pipeline can significantly improve your software development life cycle by enhancing efficiency, reliability, and scalability. The following are some of the best-fit design patterns for CI/CD, along with their key considerations:

- **Pipeline as code**: Define your CI/CD pipeline using code, which is version-controlled and can be reviewed like any other piece of code:
    - **Benefits**:
        - This enhances transparency and collaboration.

    - This facilitates version control of pipeline configurations.
    - This ensures reproducibility and consistency.
  - **Considerations**:
    - This requires familiarity with the pipeline scripting language (Jenkinsfile, GitLab CI YAML).
    - This involves an initial learning curve and setup time.
- **Immutable infrastructure**: Infrastructure components are entirely replaced rather than being modified in place:
  - **Benefits**:
    - This ensures consistency across environments.
    - This reduces configuration drift and simplifies rollback processes.
  - **Considerations**:
    - This requires comprehensive automation of infrastructure provisioning.
    - This can lead to higher initial setup costs and complexity.
- **Microservices architecture**: Applications are divided into small, autonomous services that can be developed, deployed, and scaled independently:
  - **Benefits**:
    - This enhances agility and scalability.
    - This improves fault isolation and allows for independent deployment cycles.
  - **Considerations**:
    - This increases complexity in service management and inter-service communication.
    - This requires robust service discovery, orchestration, and monitoring tools.
- **Infrastructure as Code (IaC)**: This treats infrastructure provisioning and management as code:
  - **Benefits**:
    - This accelerates deployment and improves traceability.
    - This ensures consistent and repeatable environments.

  - **Considerations**:
    - This requires investment in IaC tools (e.g., Terraform, AWS CloudFormation).
    - This needs ongoing maintenance and version control of the infrastructure code.
- **Blue-green deployment**: This maintains two identical production environments, with only one serving live traffic at any time:
  - **Benefits**:
    - This minimizes downtime and reduces deployment risk.
    - This simplifies rollback in case of deployment failures.
  - **Considerations**:
    - This requires duplicate infrastructure, increasing costs.
    - This needs robust traffic management and routing mechanisms.
- **Canary releases**: These gradually roll out new code to a small subset of users before wider deployment:
  - **Benefits**:
    - These reduce risk by monitoring the impact on a small segment.
    - These allow for early detection of issues before full-scale rollout.
  - **Considerations**:
    - These require detailed monitoring and metrics to evaluate canary performance.
    - These need quick rollback mechanisms in case of issues.
- **Automated testing**: This integrates automated tests (unit, integration, functional) into the CI/CD pipeline:
  - **Benefits**:
    - This ensures code quality and reduces manual testing effort.
    - This provides quick feedback to developers.
  - **Considerations**:
    - This requires investment in test automation frameworks and tools.
    - This needs ongoing maintenance of test scripts and infrastructure.

- **Continuous feedback**: This collects feedback from all stages of the development life cycle:
  - **Benefits**:
    - This drives continuous improvement in the CI/CD process.
    - This ensures high-quality, reliable deployments
  - **Considerations**:
    - This requires comprehensive monitoring and feedback mechanisms.
    - This needs integration with various tools and platforms to collect and analyze feedback.
- **GitOps**: This uses Git as the source of truth for infrastructure and application configurations:
  - **Benefits**:
    - It provides a single source for configuration and deployments.
    - It facilitates auditability and compliance.
  - **Considerations**:
    - It requires knowledge of Git and CI/CD tools integration.
    - It needs proper access controls and security measures for the Git repository.
- **Monorepo versus polyrepo**:
  - **Monorepo**: A single repository contains the code for multiple projects:
    - **Benefits**:
    - It simplifies dependency management and version control.
    - It facilitates consistent development workflows.
  - **Considerations**:
    - It can become complex with very large code bases.
    - It requires robust build and CI/CD tools to manage scale.
  - **Polyrepo**: Separate repositories for different projects:
    - **Benefits**:
    - It isolates changes and reduces the impact of breaking changes.
    - It simplifies repository management for small teams.

- **Considerations**:
  - It can lead to dependency management challenges.
  - It requires coordination for cross-repo changes.

In conclusion, the successful implementation of CI/CD relies on a careful balance between following proven design patterns and avoiding anti-patterns. By staying flexible and adaptable, teams can ensure that their CI/CD pipelines remain effective and contribute positively to the software development life cycle.

Another aspect, certainly not to be forgotten, is platform engineering. Let's have a look at it in the next section.

CI/CD design pattern and platform engineering

Platform engineering could very much complement CI/CD by establishing a robust foundation that supports the entire software development life cycle. It focuses on building and managing platforms that offer standardized tools, automated workflows, and consistent environments to enhance developer productivity. This discipline is crucial for cloud-native development where the complexity of services and the need for rapid delivery are ever-increasing.

The synergy between CI/CD design patterns and platform engineering lies in their shared goal of improving the developer experience and operational efficiency. While CI/CD design patterns streamline the process of integrating and delivering code changes, platform engineering ensures that the underlying infrastructure and tools are optimized for these activities. Together, they enable organizations to accelerate the pace of innovation and respond quickly to market demands.

In practice, CI/CD design patterns might include practices such as feature toggles, blue-green deployments, and canary releases, which help manage risk and reduce downtime during deployments.

Platform engineering might involve creating **internal developer platforms** (**IDPs**) that offer self-service capabilities, reducing the cognitive load on developers and allowing them to focus on creating value-added features.

Let us look at this with the help of a real-life example. A software solutions provider can have a large development team working on various microservices. To improve efficiency, they create an IDP. This platform provides automated, self-service capabilities for setting up and managing development environments, deploying applications, monitoring performance, and more. With this platform, developers don't have to spend time on these tasks and can focus on writing code and creating value-added features.

As the field evolves, we see a trend toward more intelligent and adaptive CI/CD systems that can learn from past deployments and adjust their behavior to improve future releases. Techniques such as advanced analytics and machine learning can be used to analyze patterns in code changes, build successes, and deploy outcomes to fine-tune the CI/CD process.

Moreover, platform engineering is expected to grow in importance as more organizations recognize the benefits of providing a seamless developer experience. It will become increasingly common that a significant majority of large software engineering organizations will have dedicated platform engineering teams.

To conclude, CI/CD design patterns and platform engineering are two sides of the same coin, both aiming to deliver software more efficiently and reliably. As they continuously evolve, we can expect them to become even more integrated, with platform engineering providing the infrastructure and tooling that CI/CD design patterns leverage to automate and improve the software delivery pipeline. The future of software development looks promising, with these practices leading the way toward more agile, resilient, and user-centric delivery models.

In the next section, we will have a look at a case study to show you what the next steps can be for an organization to avoid anti-patterns and combine platform engineering with CI/CD design patterns.

Exploring a case study of CI/CD integration in platform engineering

To explain this section, the best thing to do is to look at a possible case study. The following case study is fictional but could very well be applied to a real-life case.

The integration of CI/CD in platform engineering

The subject of this case study is a mid-sized software development company that sought to improve its deployment frequency and quality of software releases. The company faced challenges with manual processes that were error-prone and time-consuming. The goal was to implement a CI/CD pipeline that could automate these processes, enhance collaboration among developers, and streamline the path from development to production. Next, we are going to look at the design patterns we are going to use.

Design patterns employed

The following design patterns are being used:

- **Microservices architecture**: The company adopted a microservices architecture to decouple services and enable independent deployments. This pattern allowed for smaller, more manageable pieces of the application to be developed and deployed, which is a cornerstone of effective CI/CD practices.
- **Immutable infrastructure**: IaC was used to create reproducible and consistent environments. By treating infrastructure as immutable, any changes required a rebuild of the environment, thus reducing inconsistencies and drift between development and production.
- **Blue-green deployments**: To minimize downtime and risk, the company implemented blue-green deployment strategies. This involved maintaining two identical production environments

(blue and green) where one would serve live traffic while the other was updated. After testing, traffic would be switched to the updated environment.

- **Canary releases**: Canary releases were used to deploy changes to a small subset of users before a full-scale rollout. This pattern helped in identifying potential issues early and mitigating risk by not affecting the entire user base.
- **Automated testing**: A comprehensive suite of automated tests was integrated into the CI/CD pipeline. This included unit tests, integration tests, and end-to-end tests to ensure code quality and functionality at every stage of the pipeline.

In this diagram, all the components and patterns are involved:

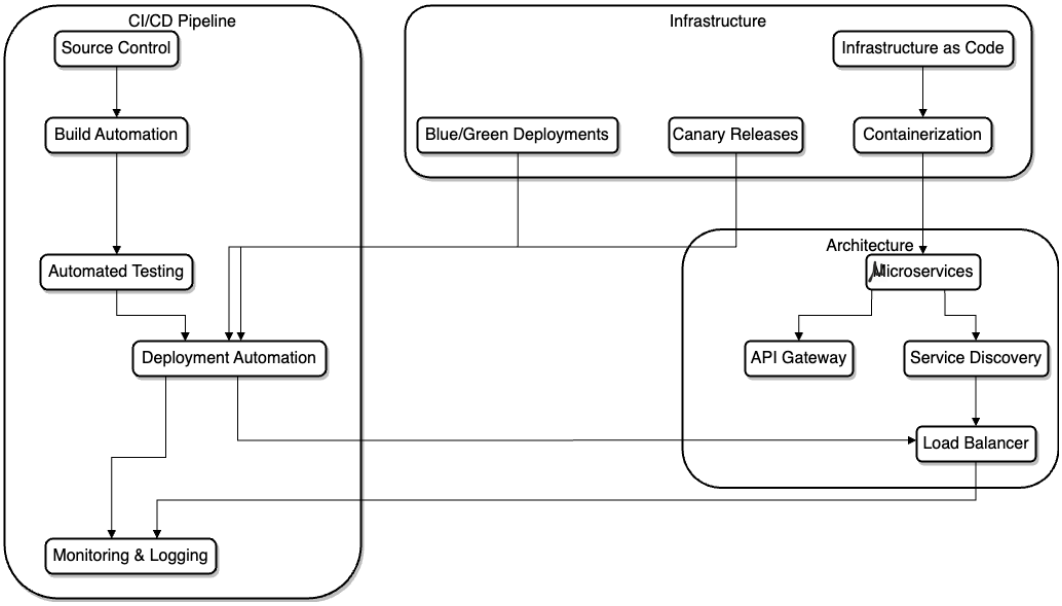

Figure 13.5 – Architecture diagram of components used in this case study

Here's a brief explanation of the components in the architecture diagram:

- **CI/CD Pipeline**:
  - **Source Control**: The repository where the code is stored and versioned (e.g., Git). It tracks changes and manages different branches of code.
  - **Build Automation**: Automated processes that compile and build the application. This stage ensures that the code integrates correctly and produces an executable package.
  - **Automated Testing**: A suite of tests (unit, integration, end-to-end) that automatically run to verify the code's functionality, quality, and performance before deployment.

  - **Deployment Automation**: This automates the process of deploying code to different environments (staging, production). This minimizes manual intervention and speeds up deployments.
  - **Monitoring and Logging**: Tools and services that track the performance and health of the deployed applications. Logs are collected and analyzed to detect and resolve issues.

- **Architecture**:
  - **Microservices**: Building modular applications where each service is independent and handles a specific functionality. This enables easier scaling and deployment.
  - **API Gateway**: A point of entry, centralized, for managing and routing requests to the appropriate microservices. It handles tasks such as load balancing, security, and monitoring.
  - **Load Balancer**: Traffic will be distributed across multiple servers or instances of microservices to ensure high availability and reliability.
  - **Service Discovery**: A system that helps microservices locate each other dynamically at runtime, enabling them to interact without hard-coded addresses.

- **Infrastructure**:
  - **IaC**: Treat computing infrastructure as a software artifact, through machine-readable configuration files, ensuring consistency and reducing manual errors.
  - **Containerization**: Packaging applications and their dependencies into containers (e.g., Docker), which run consistently across different environments.
  - **Blue-green Deployments**: A deployment strategy where two identical environments (blue and green) are used. One serves production traffic while the other is updated and tested before switching traffic to it.
  - **Canary Releases**: Gradual rollout of new code to a small subset of users before deploying it to the entire user base. This helps in catching issues early with minimal impact.

These components work together to form a robust, automated CI/CD pipeline within a microservices-based architecture, supported by modern infrastructure practices.

Anti-patterns avoided

What has been avoided are the following anti-patterns:

- **Big bang deployments**: The company moved away from large, infrequent releases, which often led to significant integration issues and deployment challenges. Instead, they embraced smaller, more frequent deployments that were easier to manage and troubleshoot.
- **Manual gates**: Manual approval processes were minimized to avoid bottlenecks. Automation was introduced wherever possible to maintain a steady flow of code from development to production.

- **Configuration drift**: By using IaC and containerization, the company avoided configuration drift between environments. This ensured that the code would run consistently across all stages of the pipeline.
- **Overdependence on end-to-end testing**: While end-to-end tests are valuable, overreliance on them can slow down the pipeline. The company balanced its testing strategy by investing in unit and integration tests that could run quickly and provide fast feedback.

Results

The implementation of these CI/CD design patterns resulted in a 50% reduction in time to market new features. Deployment frequency increased from bi-weekly to multiple times per day, with a significant decrease in deployment failures. The automated pipeline also freed up developer time, allowing them to focus on feature development rather than deployment issues.

Conclusion

This case study demonstrates the impact of CI/CD practices in platform engineering. By embracing modern design patterns and avoiding common anti-patterns, the company achieved a more efficient, reliable, and agile deployment process. As the field of platform engineering continues to evolve, the lessons learned from such implementations will be invaluable for organizations looking to optimize their software delivery life cycle.

Summary

In this chapter, we learned about what anti-patterns are, and how they seem common and legitimate to use. However, they can become a real issue because they can help in the short term, but in the long term, they will pose all kinds of problems. That is why you should avoid them as much as possible; although it takes more effort in the beginning, the benefits will pay you back in the long term.

Knowing this will make you able to choose the right path from the beginning, a skill that will bring high value to you, your projects, and, of course, your customers. Looking at the case study, but also looking at real life around you, you might find enough examples where the usage of anti-patterns proves why you should not use them.

In the next chapter, we are going to look at some interesting case studies; plus, you can test the knowledge you have gained so far!

Part 6: Case Studies

This part is specifically targeted toward providing some case studies to visualize the knowledge and patterns learned throughout the book. In this part, you will see some hypothetical scenarios inspired by real-world problems where applying design patterns on the CI/CD pipeline can have multiple impacts in terms of quality and delivery of software.

This part has the following chapters:

- *Chapter 14*, *Appendix – Knowledge Test and Case Studies*

14
Appendix – Knowledge Test and Case Studies

Well, you've reached the last chapter of this book. Congratulations! We hope you found interesting patterns that you plan to implement, or have maybe even implemented some of them already.

Our goal with this book was to arm you with the knowledge of how to improve not just your pipelines but the whole delivery process, to teach you patterns and practices that can help you to properly architect the delivery process and, in the case of any problems, be able to find the solution for it without sacrificing any of the expected outcomes.

In this chapter, we give you the chance to check what you have learned from this book. The book describes a complex approach to CI/CD, so by completing the quiz and reviewing the answers, you will know whether you should come back to some chapters.

The case studies show the impact of the implementation of specific design patterns. Learning from the outcomes leads to a better understanding of the impact of implementing a pattern properly.

Knowledge test

In this section, you can test your knowledge of what you have learned in this book. We combined some topics and questions; your job is to identify the correct answer. After the quiz, you will find the answers with signposts to the specific chapter where the topic is discussed.

Give it a go!

The quiz

Each of the following questions has only one correct answer, and you can find them after the quiz.

1. What is a design pattern?

    I. A specific, reusable solution to common problems that represents best practices for solving this type of issue

    II. A general, reusable solution to common problems that represents best practices for solving this type of issue

    III. A general, reusable solution to common problems that represents best practices for solving all types of issues

2. The correct build cycle is:

    I. Initializing, compiling, testing, packaging, reviewing

    II. Initializing, compiling, reviewing, packaging, testing

    III. Initializing, reviewing, compiling, testing, packaging

3. Why might a test diamond be a better choice for applications based on microservices?

    I. A test diamond framework doesn't exist

    II. A test diamond is the better choice for microservices architecture, as it focuses heavily on integration tests, especially between microservices

    III. A test diamond is the better choice for microservices architecture, as it removes the testing obstacles by mocking integrations and focuses on E2E tests

4. What are the stages of a CI/CD pipeline?

    I. Source control, develop, build, deploy, revert, release

    II. Develop, build, deploy, test, release

    III. Source control, build, test, deploy, release

5. What is the difference between continuous delivery and continuous deployment?

    I. Continuous deployment means deployment to a production environment, while continuous delivery means carrying out the development process and deployment to development environments

    II. Continuous delivery is related to the delivery of the code to a VCS, and continuous deployment means deployments to environments

    III. Continuous deployment is a fully automated process from commit to deployment to production. Continuous delivery contains some manual steps, such as tests or approvals.

6. What is GitOps?

    I. CI/CD build using GitHub
    II. CI/CD for infrastructure
    III. An approach to delivery where the model is changed from pushing code to pulling code

7. How many metrics can we find in DORA?

    I. Three
    II. Five
    III. Four

8. What does "a pipeline is monolithic" mean?

    I. Different triggers are kept in one pipeline and are logically separated
    II. All triggers are executed in the same way
    III. Only one trigger can be configured for this repo and pipeline

9. What is a trunk in the CI?

    I. A space in a car for your luggage
    II. The main branch in the repository
    III. A package deployed from the main branch

10. The Adapter pattern is a:

    I. Structural pattern
    II. Behavioral pattern
    III. Test pattern

11. Is Domain-Driven Design (DDD) relevant to designing CI/CD?

    I. No
    II. Yes

12. What are the three key components of cloud-native ecosystems?

    I. Operating systems, applications, databases
    II. Containerization, microservices, serverless
    III. Code, deployment, monitoring

13. Which of the following examples can be treated as a singleton in IaC?

    I. One Terraform file with resource definitions

    II. Resource definitions in Terraform

    III. Module definitions in Terraform

14. What is the biggest disadvantage of blue-green deployment?

    I. A long time is needed for deployment

    II. Doubled infrastructure

    III. No control over rollbacks

15. Why should organizations use tools such as SonarQube in their CI/CD chains? Select the most complete answer.

    I. For a security and shift-left approach using quality gates

    II. For code vulnerability control

    III. For access management control

Now, let's look at the answers.

Answers

The following are the answers to the questions in the previous subsection:

1. *Option II*. You can find more information in the *An overview of design patterns* section of *Chapter 1*.
2. *Option I*. A description of this process can be found in the *The build cycle* section of *Chapter 1*.
3. *Option II*. The integration differences between monolithic and microservices are significant and require more focus and more testing of these integrations. More details can be found in the *Implementing testing in CI/CD* section of *Chapter 1*.
4. *Option III*. The stages of a CI/CD are discussed in the *What are the components of a CI/CD pipeline?* section of *Chapter 3*.
5. *Option III*. More about the differences can be found in the *A decade of action using CI/CD patterns* section of *Chapter 4*.
6. *Option III*. We mention GitOps in the *A decade of action using CI/CD patterns* section of *Chapter 4*.
7. *Option II*. Originally, DORA had four metrics, but a fifth one was added recently. More information about DORA can be found in the *Introduction to CI/CD measurements* section of *Chapter 4*.

8. *Option I*. A detailed description of this type of pipeline can be found in the *Design examples for the monorepo and polyrepo approaches* section of *Chapter 5*.
9. *Option II*. If your answer was *Option I*, well, in general, you'd be right; however, we are talking about CI/CD, not cars. The term trunk refers to the main branch in trunk-based development. A description of this approach can be found in the *Key components – tools, code, and modularity* section of *Chapter 5*.
10. *Option I*. More details can be found in the *Structural patterns* section of *Chapter 1*.
11. *Option II*. Well, we have a whole chapter about it – *Chapter 8*!
12. *Option II*. This is covered in the *Key components – containerization, microservices, and the serverless ecosystem* section of *Chapter 9*.
13. *Option III*. Although *Option II* might seem like the correct answer, it is not. A singleton in this case must be reusable. More about this can be found in the *Singletons in clouds* section of *Chapter 10*.
14. *Option II*. More details can be found in the *Blue-green deployment strategy* section of *Chapter 10*.
15. *Option I*. Yes, the answers are quite vague, but we did this on purpose. SonarQube allows teams to control the quality of the code and shift the responsibility for this quality left, to the development teams. More about this can be found in the *Implementing audit points with quality gates* section of *Chapter 11*.

Case studies

In this section, we present a few quick case studies from the market. These examples show different approaches, built on specific criteria. However, the goals are always the same – to improve the delivery process.

Case 1 – Improving unnecessary "wiring"

In this case, we explore a situation where an organization has a very strong opinion about the tools that should be in the CI/CD chain. This, however, heavily impacts the build process for CI/CD and later for maintenance.

Without going deeper into the tooling itself, let's look at the following diagram, which helps us to understand the initial state of the system:

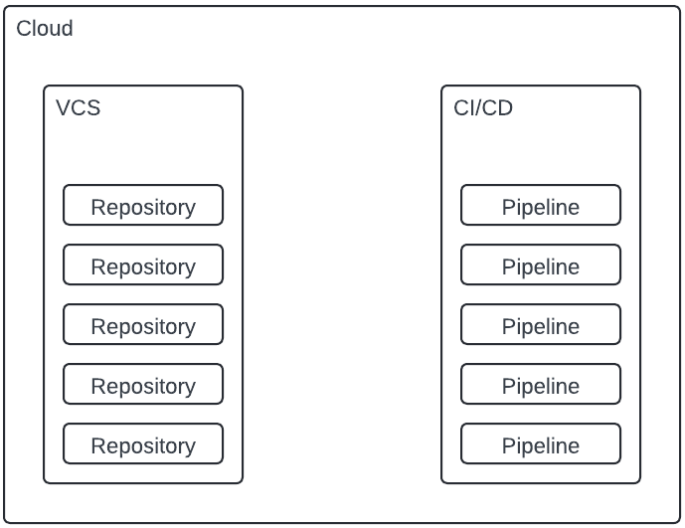

Figure 14.1 – Initial setup of a VCS and CI/CD tools

Very quickly, the team realizes that this setup is not complete. In fact, they need more pipelines related to tests, infrastructure, and so on. The team adds these pipelines and the state of the system looks as in the following diagram:

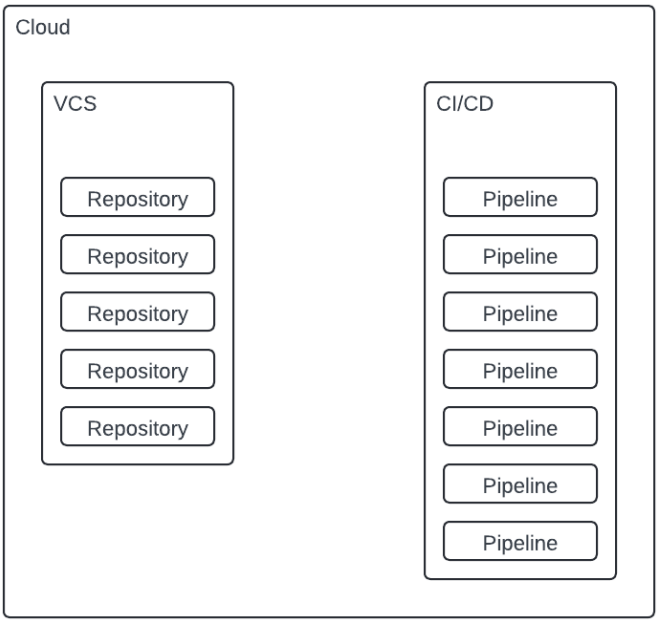

Figure 14.2 – Additional pipelines added to the system to fulfill missing functionalities

In order to make everything work, the team implements something called *wiring* to connect these two separate elements and allow them to communicate.

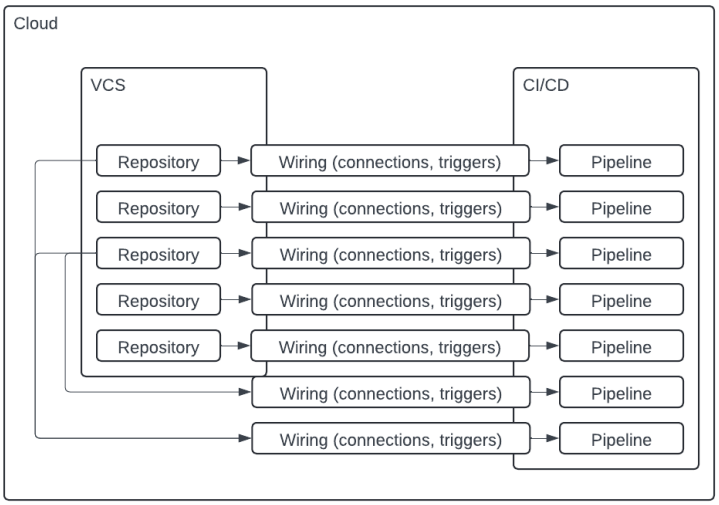

Figure 14.3 – Wiring implemented in the process

Right now, we can easily see the additional burden added to the process. Each repository is connected with its pipeline using a dedicated construct (in this case, it is an AWS Lambda function, but it doesn't matter what it is in other cases). This approach, although quite common, creates challenges for those who will maintain and scale the solution in the future.

So, the team implements a solution, as presented in the following diagram:

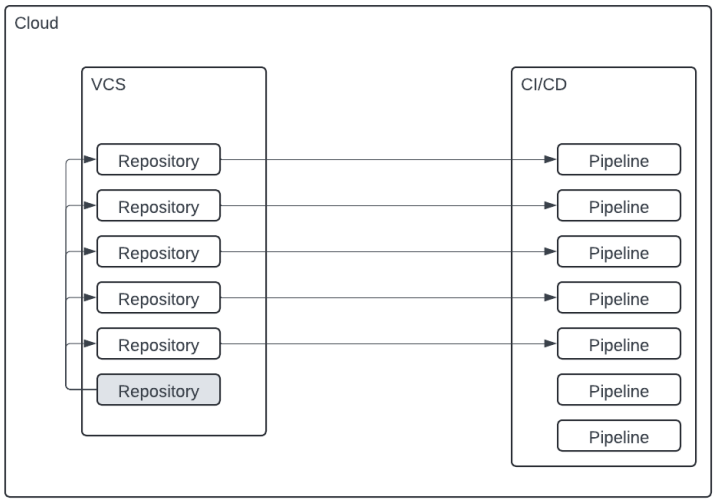

Figure 14.4 – Solution implemented

As we can see, a new repository was added (darker box in the diagram), where all elements needed for wiring are created and stored. This doesn't change the technical way of connecting the components, but modularization, automation, and better control of wiring are achieved.

Case 2 – Fixing bad decision

We will stick with the same project. We saw in the preceding section how bad architectural decisions can impact the solution. In this case, the team decides to analyze the approach again and pinpoint the main issues. What do they find?

- **Disconnection**: The VCS and CI/CD are disconnected and the information flow between them is impossible to implement simply. The wiring has helped but hasn't solved all the problems.
- **Missing data**: With this disconnection, the team can't really measure the CI/CD process. We mean *measure*, not *monitor*. Although the monitoring itself isn't very easy, the team isn't able to collect valid DORA data.
- **Additional steps and additional code needed to be maintained**: This is the toughest task for the team in the long run from a technical standpoint. They realize that even with an improved approach, they still need a lot of time to collect information about the failure and spend more time than expected to fix it. And believe us, they will face failures very often, as the wiring isn't very stable.

With these findings in mind, the team decides to change their approach to the tooling. As they have identified the problematic elements, the final decision is to move to one tool, where repositories can connect to pipelines in a native way. This removes the burden of wiring and allows the team to, first, get proper metrics, and second, limit the maintenance time for the solution to a minimum.

Case 3 – Embracing the platform team

The organization created a platform team, which is also responsible for CI/CD pipelines. In the beginning, the team decides that when their users (other teams) are onboarded to the project, they will receive an empty code repository in the process. The repository will have all the needed connections to their resources in the cloud, and so on. However, the users will be responsible for creating the code in the repo, including technical code, such as pipelines.

This quickly escalates to a situation where the platform team has no time for work related to platform management and platform development, as all their time is spent placating angry users. Users have myriad problems, from complicated ones such as some specific errors in the pipelines to unexpected ones such as users who can't write pipelines.

The solution is to modularize and standardize the approach. The platform team constructs the skeletons of pipelines for different types of workloads (such as infrastructure), compiles code for the backend, and interprets code for the frontend, tests, and so on. When users select the type of repository, they also select the type of workload. After this, the relevant skeletons are copied to the newly created

repositories. Users also receive a notification if something is updated in the skeletons they use, so they can update their pipelines if they want to.

Standardization also includes quality gates, such as IaC checks, SonarQube, SBOM generation, and, most importantly, centralized artifact storage. With this solution, the organization achieves not only a standardized process but a fully controlled and auditable process.

Summary

Congratulations! You (hopefully) passed the quiz, and you saw the struggles teams can experience through the case studies we presented.

We hope that this book has helped you to better understand the bigger picture of CI/CD. In modern organizations, we should stop thinking about CI/CD as a simple pipeline, but as a vital and crucial element of the delivery system.

If we treat the organization as a body, CI/CD should be treated as the bloodstream, as its function is to deliver blood (in our case, code) to all parts of the body (environments). This means that we need to keep the CI/CD in a healthy state and do frequent check-ups.

This book gave you all diagnostic tools and proper treatments to apply (keeping with the biological metaphor) in order to make sure your approach to CI/CD is not only valid for today but also does not create risks for the future. You learned how to apply the different design patterns, how to scale the system, and also how to measure adoption and trends.

Now it is time to try it in practice. Good luck!

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

Kubernetes Anti-Patterns

Govardhana Miriyala Kannaiah

ISBN: 978-1-83546-068-9

- Get to grips with the nature and characteristics of Kubernetes anti-patterns
- Find out how to achieve stable Kubernetes deployments
- Extract insights from real-world use cases for informed decision-making
- Cultivate a proactive mindset for anticipating and preventing potential problems
- Optimize Kubernetes deployments for improved efficiency and performance
- Discover actionable strategies for continuous improvement in your Kubernetes workflow

Platform Engineering for Architects

Max Körbächer, Andreas Grabner, Hilliary Lipsig

ISBN: 978-1-83620-359-9

- Make informed decisions based on your organization's platform needs
- Identify missing platform capabilities and technical debt
- Develop a critical user journey through your platform capabilities
- Define the purpose, principles, and key performance indicators (KPIs) for your platform
- Utilize relevant data points for making data-driven product decisions
- Implement your own platform reference and target architectures

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *CI/CD Design Patterns*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/978-1-83588-964-0`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly