# Implementing a Full CI/CD Pipeline

Introduction

# Implementing a Full CI/CD Pipeline

This course guides you through the process of building a CI/CD Pipeline using a variety of tools. It begins with basic CI-related concepts, like source control and build automation. Then it moves along to covering containers, orchestration, and monitoring. Finally, the course covers how to strengthen the stability of your pipeline using things like self-healing, autoscaling, and canary deployments.

While many courses focus on teaching you how to use a particular tool, this one focuses instead on using a variety of tools and how they fit together as part of the CI/CD Pipeline "big picture."

# Implementing a Full CI/CD Pipeline

What this course will do:

- Give you a hands-on introduction to some of the techniques and tools involved in doing CI/CD
- Give you an idea of what implementing CI/CD can look like in practice
- Provide you with enough knowledge to decide what tools you may want to learn more about
- Show you technologies and tools that you can use to implement CI and CD directly, like CI tools and automated deployment techniques
- Show you infrastructure tools that are designed to provide stability in the context of CD

# Implementing a Full CI/CD Pipeline

What this course will NOT do:

- Show you the only way to implement a CI/CD Pipeline:
  - There are many different options
- Show you the best way to implement a CI/CD Pipeline:
  - What is best for you depends on your situation
- Go in-depth on any of the tools:
  - But you will learn enough to implement what this course focuses on

# Implementing a Full CI/CD Pipeline

## SCM and the CI/CD Pipeline

# SCM and the CI/CD Pipeline

Source Control Management (SCM) is an important component of our CI/CD Pipeline.

- It is a huge part of the daily dev workflow:
  - All Code changes made by devs will need to be tracked in SCM
  - Devs will use SCM to track their changes separately and then merge them together
- All pieces of automation that need to interact with the source code will use SCM:
  - Continuous Integration (CI) will get the code from SCM
  - SCM will notify the CI server when code needs to be built

# Git SCM

We will be using Git, and specifically GitHub, to manage the source code for the Train Schedule app.

- Before moving forward, you need a basic understanding of a Git-based workflow
- You need to know what steps need to be done in order to make a change to the source code and ensure that change is managed in Git:
    - Cloning
    - Adding
    - Committing
    - Pushing
    - Branching
    - Merging
    - Pull Requests

# Implementing a Full CI/CD Pipeline

## Installing Git

# Installing Git

Git installation instructions for your system:

https://git-scm.com/downloads

On a Linux distro that uses rpm:

```
sudo yum -y install git
```

# Configuring Your Name and Email

After installing git, you need to set the name and email that will be associated with your commits. When using GitHub, it is best to use an email address that is associated with your GitHub account.

```
git config --global user.name "<your name>"
git config --global user.email <your email>
```

# Setting up Private Key Access

An easy way to authenticate with a remote git server like GitHub is to use an ssh key pair.

On a CentOS environment, you can generate an ssh key pair like this:

```
ssh-keygen –t rsa –b 4096
```

This command will prompt you for several things. These can be left as their defaults, though it is good practice to enter a passphrase. If you do use a passphrase, make sure you remember what it is.

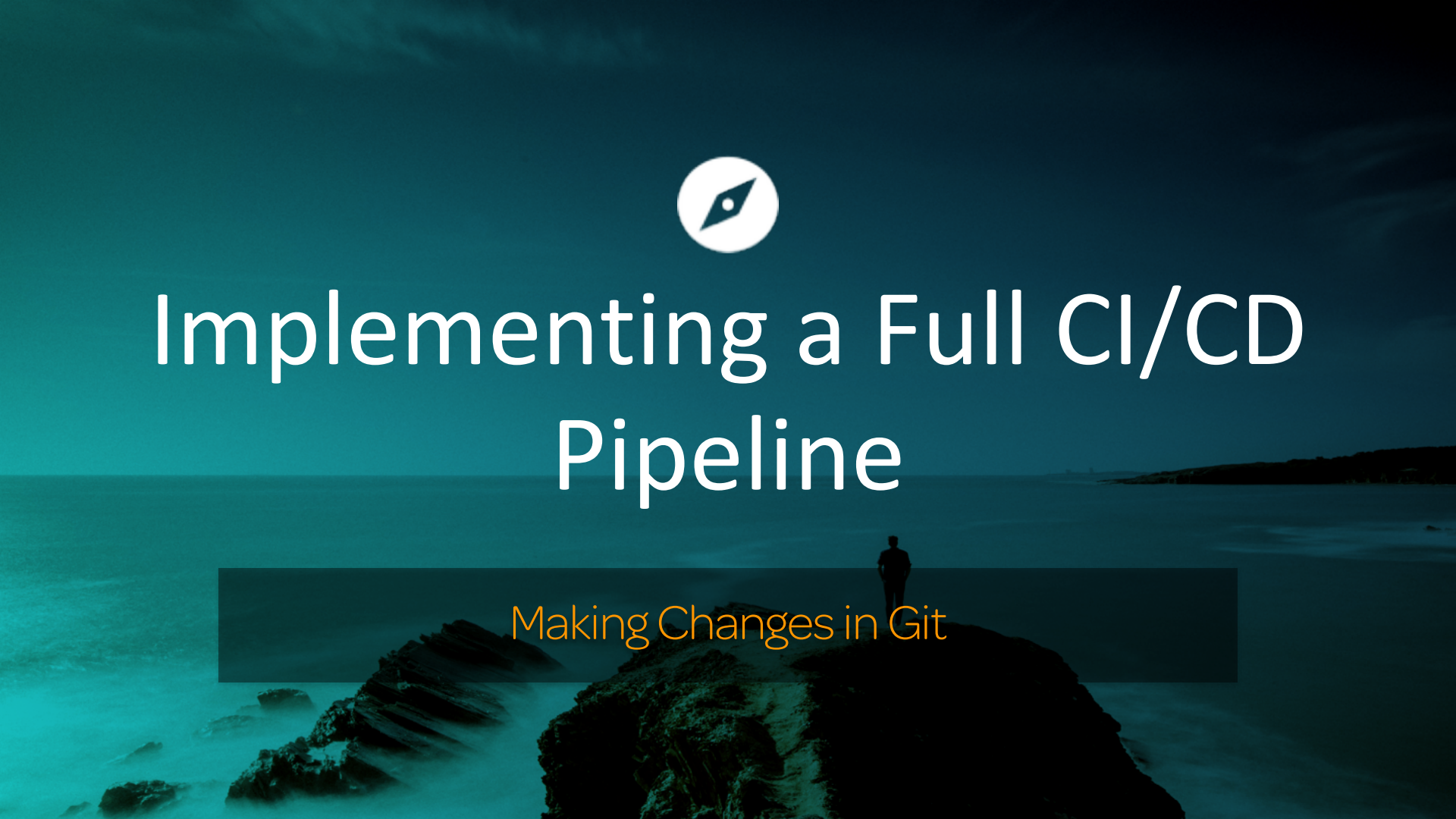
After generating the key pair, copy the contents of ~/.ssh/id_rsa.pub.

On GitHub.com, click your profile image at the top right, then click "settings," "SSH and GPG Keys."
Click "new ssh key," enter a name and paste the contents of id_rsa.pub into the key field, then submit the form.

Refer to the GitHub guide for more info, or if you want to set up a key on your own environment:

https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/

https://help.github.com/articles/adding-a-new-ssh-key-to-your-github-account/



# Implementing a Full CI/CD Pipeline

## Making Changes in Git

# git clone

If you are working with source code that already exists in a remote repository, the first thing you need to do is get a copy of the repository so that you can work with the files locally:

```
git clone <repository url>
```

This command does two things:

- Downloads a local copy of the repository, including the history of all changes
- Makes a local copy of the latest version of all the files known as the "working tree"

If the code doesn't already exist in a remote repository, you can initialize a new repository locally using:

```
git init
```

# git status

The status command is a very useful command whenever you need to know about the current status of your local repository:

```
git status
```

This command will tell you all sorts of useful information, like:

- Whether you have any changes that are not yet staged for commit
- Whether you have staged changes that are not committed
- What branch you are currently using

# git add

The add command stages changed files for the next commit. This allows you to control which files are committed and which are not:

```
git add <file>
```

Before you can commit a change to a file, you need to add the file to the staging area. Use git status to see which changed files have and have not been added. To add all files that have been changed use:

```
git add .
```

Or:

```
git add -A
```

# git commit

Commit adds your changes to your local repository and makes them part of the repository's change history. When you are satisfied with a change or set of changes that you have made to the files, you are ready to commit them:

```
git commit –m "<message describing the change>"
```

A few notes on commit:

- Commit only adds the changes to your local copy of the repository. It does not push them to any remote repository, such as GitHub
- Commit will only commit the changes that were staged using git add

# git push

Git push pushes the changes that have been made to your local repository to a remote repository, such as GitHub. Until you push your changes, they are only stored locally on your system:

```
git push
```

By default, push will push the changes to a remote repository associated with the current local branch. If you cloned the branch from an existing remote repository, this relationship is already set up for you, and you can simply use git push. If not, you may need to specify which remote repository and which remote branch you want push to use:

```
git push –u <remote name, usually origin> <branch name>
```

# Implementing a Full CI/CD Pipeline

## Branches and Tags

# git branches

Branches are used to maintain multiple versions of the code with different changes simultaneously.

Many teams use branches as part of the daily workflow.

Branches can also be used to interact with a CI/CD Pipeline. For example, some teams maintain a "production" branch, and merging changes into this branch initiates automated processes involved with deploying to production.

By default, a git repository starts with one branch called "master."

# git checkout

**git checkout** checks out an existing branch. This means that it puts the contents of the branch into your working tree and your working copy of the source code files.

When you commit, whichever branch you have checked out will be the branch that the commit is added to.

```
git checkout <branch>
```

You can create a new branch and check it out immediately with the –b flag:

```
git checkout –b <new branch>
```

# git tag

Tags in git are simply pointers to a particular commit. They can be used to provide a name that can be used to reference that commit in the future.

One use case for tags is to tag source code commits with the version of the software that they represent, so if a particular revision gets released as version 1.0.0, it could be tagged as “v1.0.0” in order to indicate that.

# Implementing a Full CI/CD Pipeline

## Pull Requests

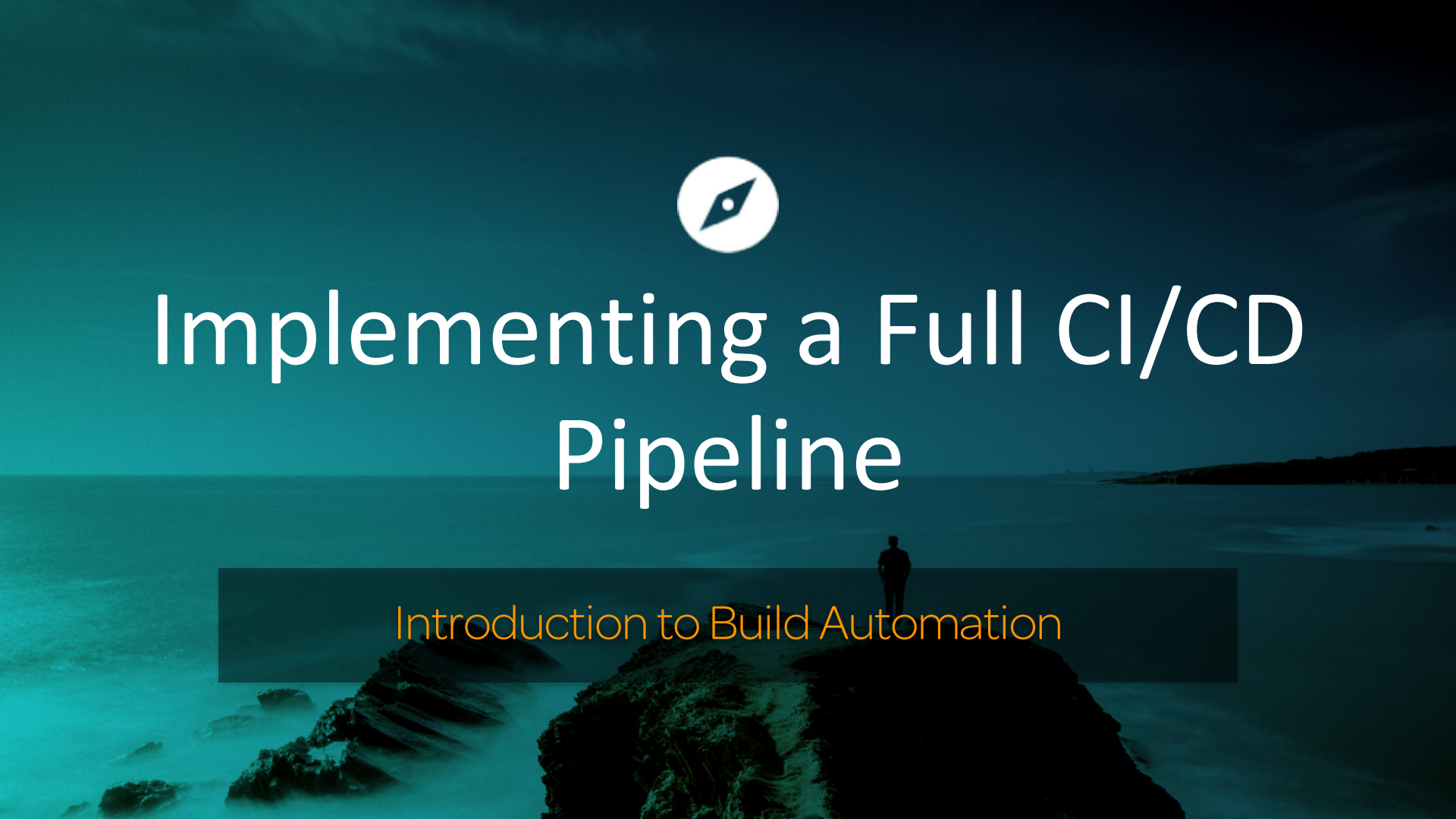
# Pull Requests

Pull requests are not actually a feature of the git software itself. However, they are a useful tool offered by many git remote server implementations, such as GitHub.

Usually, teams work using multiple branches to manage all of their changes. At some point, these branches need to be merged together (preferably as often as possible).

Merges can he handled locally using git, but another way to do this is through pull requests.

A pull request is a request made by a developer to merge their changes into another branch (usually a shared mainline). It gives other team members a chance to review the changes before performing the merge.



# Implementing a Full CI/CD Pipeline

## Introduction to Build Automation

# Build Automation

Build automation is the automation of tasks needed in order to process and prepare source code for deployment to production. It is an important component of continuous integration.

This includes things like:

- Compiling
- Dependency Management
- Executing Automated Tests
- Packaging the App for Deployment

# Gradle

In this course, we will use Gradle as our build automation tool. There are plenty of other tools, but Gradle is very powerful for doing the things we will need to do.

Use the following link to download Gradle:

https://gradle.org/

# Implementing a Full CI/CD Pipeline

## Installing Gradle

# Installing Gradle

Gradle can be easily installed with a package manager.

Dependencies: Java JDK 7 or higher.

On some systems, you will need to download and extract a zip file.

Installation instructions for a variety of environments can be found at:

https://gradle.org/install

You can verify that your installation is working with:

```
gradle --version
```

# The Gradle Wrapper

One of the features of Gradle that we will use is the Gradle Wrapper.

From the Gradle site: "The Wrapper is a script that invokes a declared version of Gradle, downloading it beforehand if necessary."

Gradle Wrapper allows Gradle to install itself using just the files from your project's source control.

The Wrapper is useful because:

- It removes the need to have Gradle installed beforehand in order to run the build
- Ensures the project is always built with a specific version of Gradle
- Lets you build multiple projects with different Gradle versions on one system
- Anyone (or any automated process) can run the build quickly and easily - they only need Java

# Installing The Gradle Wrapper

If you already have a normal Gradle install on your system, you can use it to easily install the wrapper:

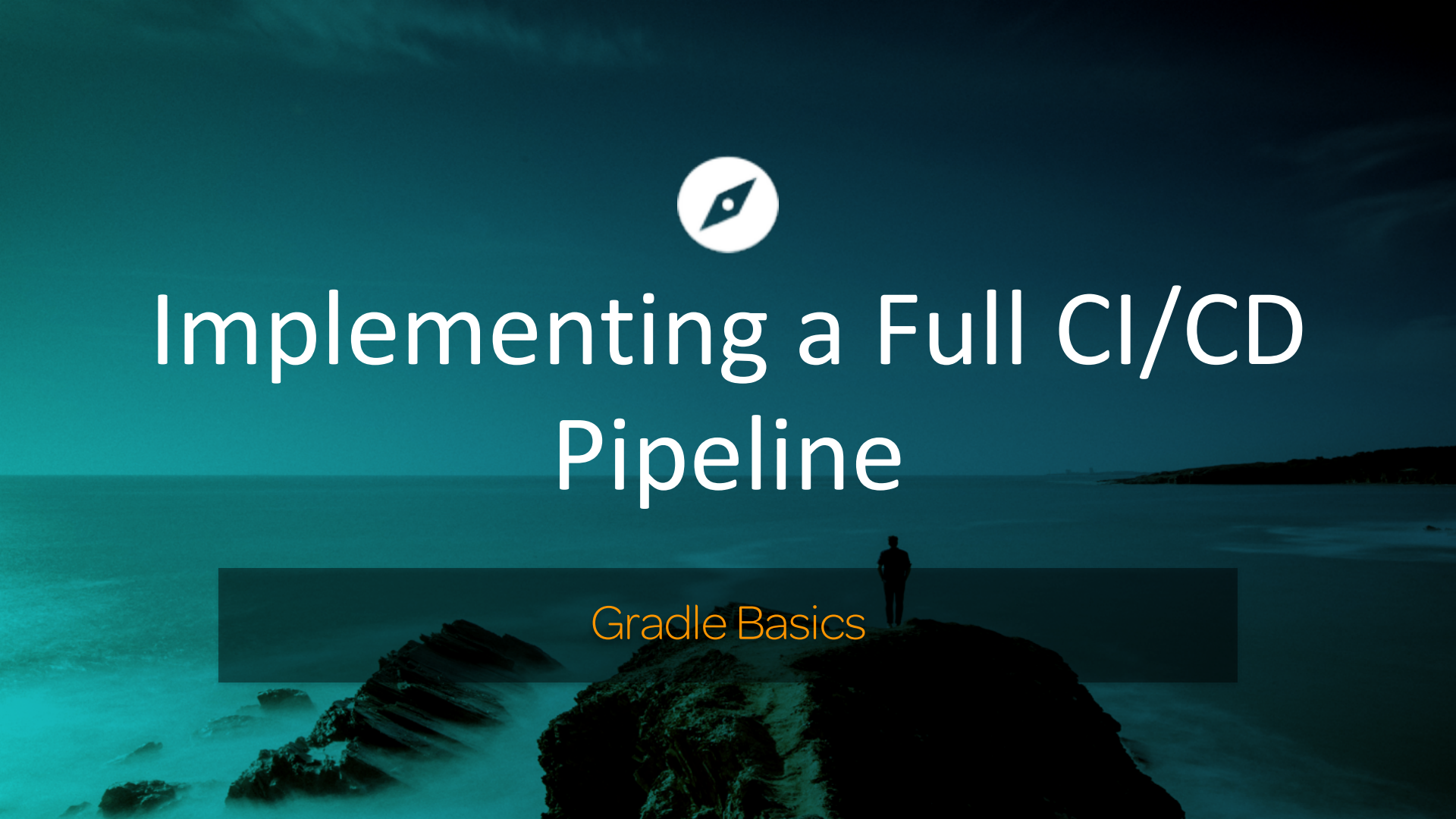
- cd /your/project/root/directory
- gradle wrapper

You should also add the line .gradle to your .gitignore file.

This will place a script file in your project's root directory called gradlew (there is also a gradlew.bat for Windows systems).

Run Gradle commands against your project like this:

- cd /your/project/root/directory
- ./gradlew build



# Implementing a Full CI/CD Pipeline

## Gradle Basics

# Gradle Basics

A gradle build is defined in a groovy script called build.gradle, located in the root directory of your project.

Use gradle init to initialize a new project. This also sets up the gradle wrapper for you.

# Running a Gradle Build

Gradle builds consist of a set of tasks that you call from the command line:

```
./gradlew someTask someOtherTask
```

This command will run a task named someTask, and a task named someOtherTask.

# Defining Tasks

The build.gradle file controls what tasks are available for your project.

You can define your own tasks with custom code in build.gradle:

```
task myTask {
  println 'Hello, World!'
}
```

Most of the tasks you use will come built into either Gradle itself or plugins.

# Task Dependencies

Gradle uses the concept of task dependencies to determine what tasks need to be run for a particular build command:

- If you run a task, any other tasks that depend on it will also be run

Task dependencies also determine the order in which tasks get run:

- A task's dependencies will always run before that task

You can define a dependency relationship between tasks in build.gradle like this:

- taskA.dependsOn taskB

# Plugins

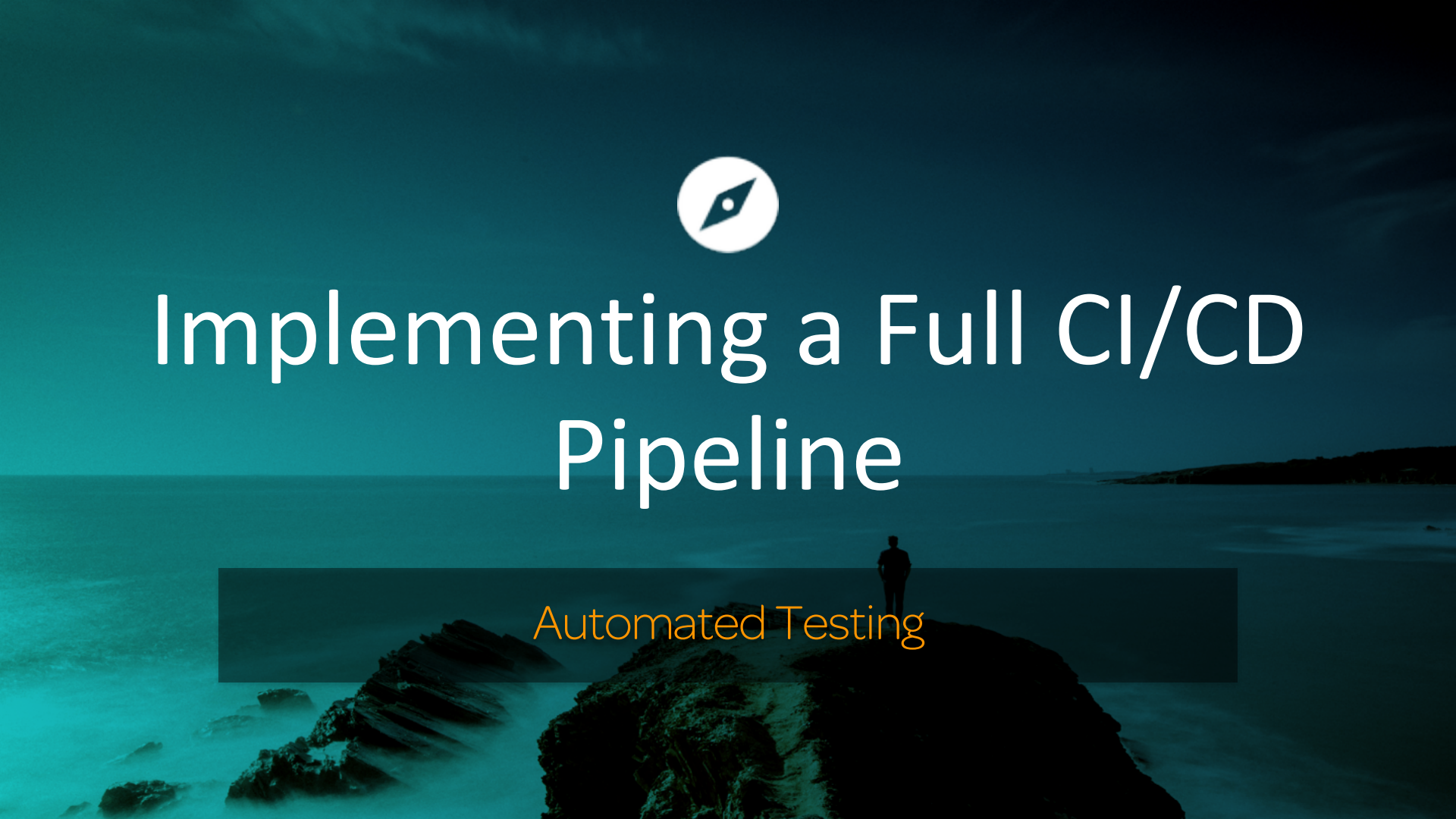
Gradle has a huge variety of plugins available, many of them contributed by the community.

Plugins add all kinds of functionality to Gradle. They usually include pre-built tasks that you can configure to do what you need.

You can include plugins in your build.gradle like this:

```
plugins {
  id "<plugin id>" version "<plugin version>"
}
```

You can browse public availably plugins at https://plugins.gradle.org/.



# Implementing a Full CI/CD Pipeline

## Automated Testing

# What is Automated Testing?

Automated testing is the automated execution of tests that verify the quality and stability of code.

Automated tests are usually code themselves, so they are code that is written to test other code.

Automated tests are often run as part of the build process and are executed using build tools like Gradle.

# Types of Automated Tests

There are multiple types of automated tests:

- Unit Tests - focus on testing small pieces of code in isolation. Usually a single method or function.
- Integration Tests - test larger portions of an application that are integrated with each other.
- Smoke Tests / Sanity Tests - these are high-level integration tests that verify basic, large-scale things like whether or not the application runs, whether application endpoints return http 500 errors, etc.

The degree of testing implemented at any of these levels is up to the team!

# Running Automated Tests in Gradle

How the tests are run depends on what type of tests you have and the technology used to build them. Generally, automated tests will be run in Gradle as a task that gets executed as part of the build process:

- task build
- task test
- build.dependsOn test

Whenever ./gradlew build gets run, the test task will also run!

# Implementing a Full CI/CD Pipeline

## Continuous Integration Overview

# Continuous Integration

Continuous Integration (CI): the practice of frequently merging code changes.

BUT frequent merges cause difficulties:

- What if the code doesn't compile after a merge?
- What if someone broke something?
- It would be a lot of work to check these things on every merge

The solution: automate it!

# How CI Works

A CI server executes a build that automatically prepares/compiles the code and runs automated tests.

Usually, the CI server automatically detects code changes in source control and runs the build whenever there is a change.

If something is wrong, the build fails. The team can get immediate feedback if their change broke something.

Merging frequently throughout the day and getting quick feedback means that if you broke something, you broke it with your changes within the last couple hours. This is much easier to fix than if you broke something with your changes from a week ago!

# Jenkins

We will be using Jenkins as our CI server in this course.

From https://jenkins.io :

**“The leading open source automation server, Jenkins provides hundreds of plugins to support building, deploying and automating any project.”**

# Implementing a Full CI/CD Pipeline

## Installing Jenkins

# Installing Jenkins

Installation docs: https://jenkins.io/doc/book/installing/

There are several installation options:

- Docker container
- WAR file
- Packages for various OSes

For now, we will install Jenkins on a CentOS server using yum:

https://wiki.jenkins.io/display/JENKINS/Installing+Jenkins+on+Red+Hat+distributions

# Install Java

Some versions of CentOS ship with a java version that is not compatible with Jenkins, so you may have to remove it:

- sudo yum remove java

Install java:

- sudo yum install java-1.8.0-openjdk

# Install Jenkins with yum

Configure the Jenkins yum repos so we can install Jenkins using yum:

- sudo yum install epel-release
- sudo wget -O /etc/yum.repos.d/jenkins.repo http://pkg.jenkins-ci.org/redhat/jenkins.repo
- sudo rpm --import https://jenkins-ci.org/redhat/jenkins-ci.org.key

Then install Jenkins using yum:

- sudo yum -y install jenkins-2.112-1.1.noarch

Enable and start the Jenkins service:

- sudo systemctl enable Jenkins
- sudo systemctl start jenkins

# Finish setting up Jenkins in the UI setup

Go to <your server address>:8080 in a browser.

Follow the on-screen prompts to finish setting up Jenkins!

# Implementing a Full CI/CD Pipeline

## Setting Up Jenkins Projects

# Jenkins Projects

The configuration which controls what a piece of Jenkins automation does and when it executes is called a project.

A simple way to implement a CI build in Jenkins is with a freestyle project.

Click on “New Item” to create a new project.

# Demo

Let's set up a new freestyle project in Jenkins!

We will set up a CI build for the train schedule app found at https://github.com/linuxacademy/cicd-pipeline-train-schedule-jenkins.

This build will:

- Check out the code from GitHub
- Run the gradle build automation that is packaged with the source code


Implementing a Full CI/CD Pipeline
Triggering Builds with Git Hooks

# Webhooks

Webhooks are event notifications made from one application to another over http.

In Jenkins, we can use webhooks to have GitHub notify Jenkins whenever the code in GitHub changes. Jenkins can respond by automatically running the build to implement any changes.

We can configure Jenkins to automatically create and manage webhooks in GitHub as necessary.

# Setting up GitHub Webhooks in Jenkins

**Configuring webhooks in Jenkins is relatively easy. We need to:**

- Create an access token in GitHub that has permission to read and create webhooks
- Add a GitHub server in Jenkins for GitHub.com
- Create a Jenkins credential with the token and configure the GitHub server configuration to use it
- Check "Manage Hooks" for the GitHub server configuration
- In the project configuration, under "Build Triggers," select "GitHub hook trigger for GITScm polling"

# Implementing a Full CI/CD Pipeline

Jenkins Pipelines

# What is Jenkins Pipelines?

Jenkins Pipelines is a set of Jenkins plugins that support doing continuous delivery in Jenkins.

A Jenkins Pipeline is an automated process built on these tools. It takes source code through a "pipeline" from the source code creation all the way to production deployment.

You can find the Jenkins Pipelines documentation at https://jenkins.io/doc/book/pipeline/.

# How do you Create a Jenkins Pipeline?

Pipelines adheres to the best practice of infrastructure as code.

Therefore, a pipeline is implemented in a file that is kept in source control along with the rest of the application code. This file is called a Jenkinsfile.

To create a Pipeline, simply create a file called Jenkinsfile and add it to your source control repo.

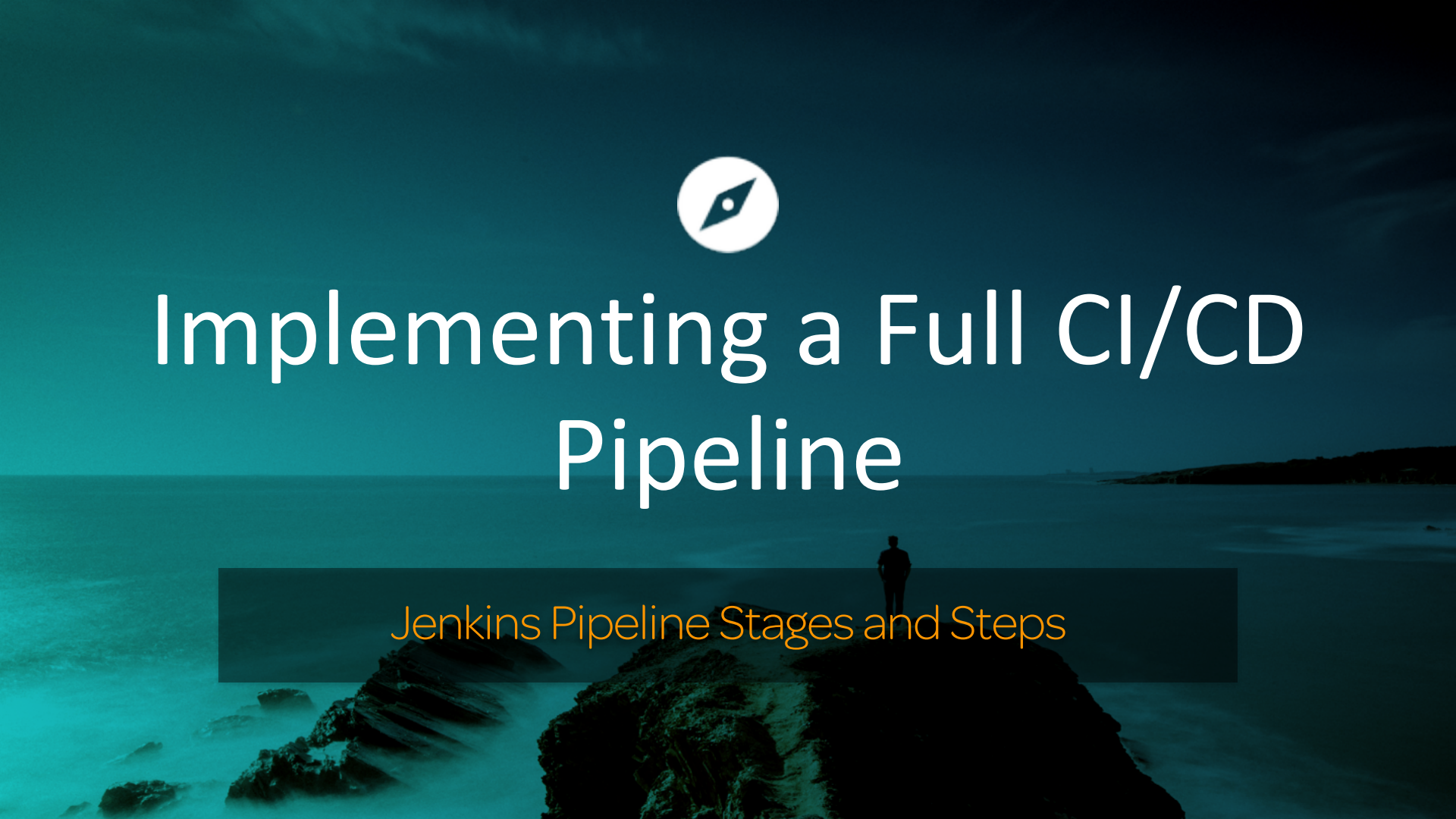
When creating the Jenkins project, choose the “Pipeline” or “Multibranch Pipeline” project type.

# What Goes in a Jenkinsfile?

Pipelines has a domain-specific-language (DSL) that is used to define the pipeline logic.

There are two styles of Pipeline syntax you can use (you must choose one or the other):

- **Scripted** – A bit more like procedural code
- **Declarative** – Syntax describes the Pipeline logic



# Implementing a Full CI/CD Pipeline

## Jenkins Pipeline Stages and Steps

# Pipeline Stages

Pipeline Stages are large pieces of the CD process.

For example:

- Build the code
- Test the code
- Deploy to Staging
- Deploy to Production

# Pipeline Steps

**Pipeline Steps** are the individual tasks that make up each stage.

For example:

- Execute a command
- Copy files to a server
- Restart a service
- Wait for input from a human

Steps are implemented through special declarative keywords in the Jenkinsfile DSL. Jenkins plugins can add new steps.

Check out the Steps reference for documentation on many of the steps that are available:
https://jenkins.io/doc/pipeline/steps/

# Implementing a Full CI/CD Pipeline

## Deployment with Jenkins Pipelines

# Continuous Delivery

Continuous Delivery means ensuring that you are always able to deploy any version of your code. It is necessary in Continuous Deployment, where you are actually deploying your code frequently.

In order to support Continuous Delivery / Deployment with a Jenkins Pipeline, we need use the Pipeline to automate the deployment process.

# Automated Deployment in a Pipeline

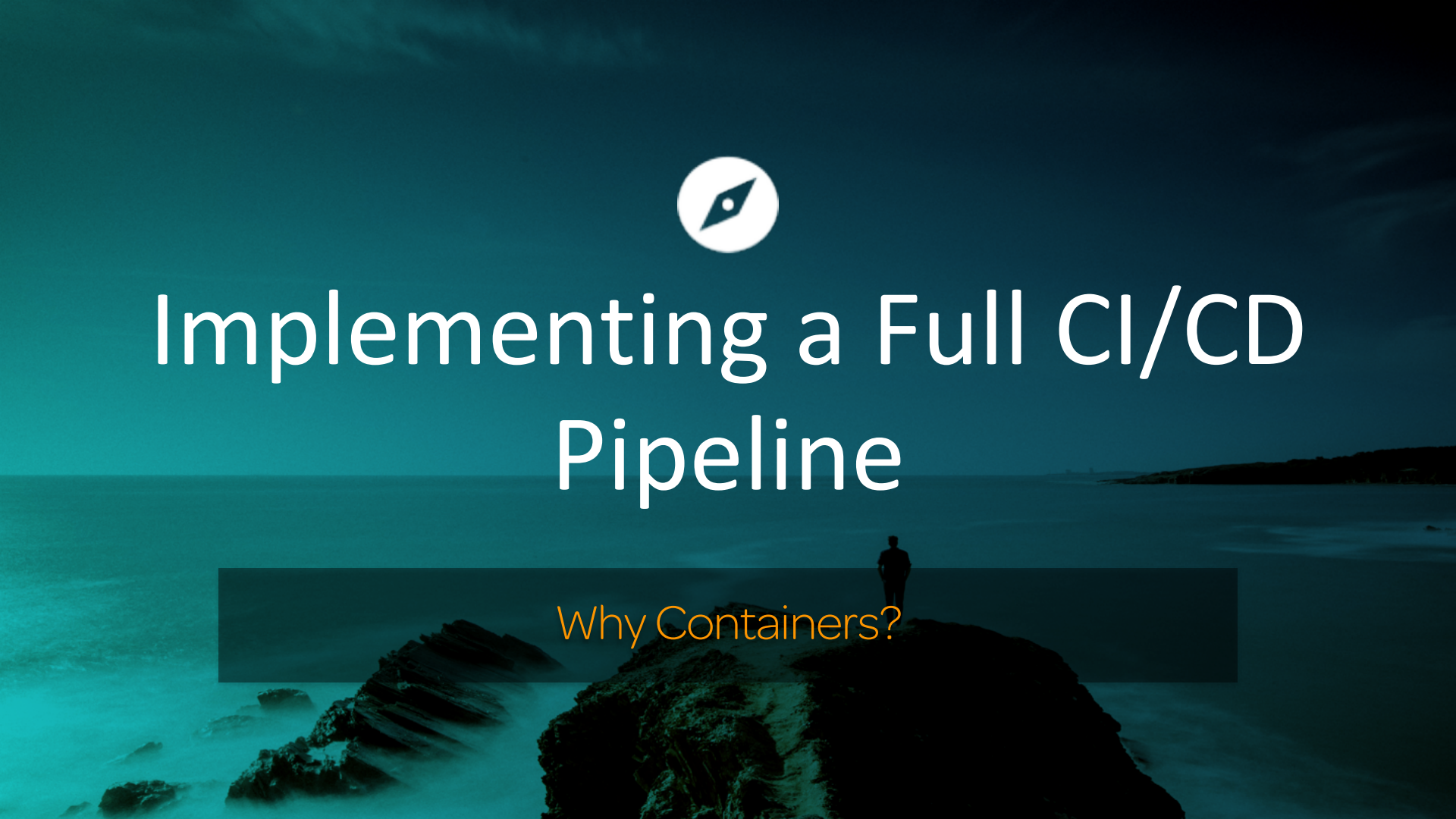
Here's how we can automate deployments in a Pipeline:

- Define **Stages** for stages of the CD process that involve deploying:
  - For example, if we have a staging server and a production server, we could implement stages called "Deploy to Staging" and "Deploy to Production"
- In each deployment stage, define **Steps** that perform the tasks necessary to carry out the deployment:
  - For example, copy files to a server, restart a service, etc
- We can also prompt a user for approval before performing the actual production deployment

# Demo Setup

Here's the current setup for the demo:

- Jenkins server configured to manage webhooks on GitHub
- Personal fork of sample code at https://github.com/linuxacademy/cicd-pipeline-train-schedule-cd
- Staging and production webservers with:
  - A user called deploy set up with permissions to perform deployment steps.
  - Systemd service configured for the train schedule app.



# Implementing a Full CI/CD Pipeline

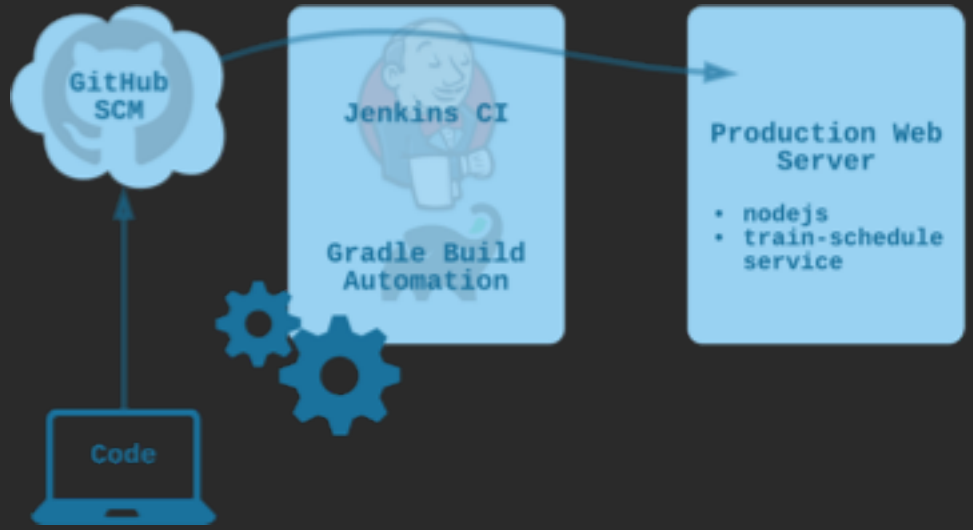
## Why Containers?

# Why Containers?

We can do continuous deployment with the current architecture!

BUT:

- What about scaling?
- What if some change breaks the server?
- Can we be confident our testing matches production?



# Why Containers?

One way to solve these problems would be utilize virtualization tools. We could build a VM image that would let us spin up new production servers when we need to.

But:

- VMs are slow
- VMs use a lot of resources
- VMs are not portable
- When we deploy a new version of the code, we have to make sure it gets deployed to a bunch of VMs

What if we could package the code AND the system-level configuration in a lightweight package that can stand up quickly and run almost anywhere?

# Why Containers?

VMs contain an entire copy of an OS, plus simulations of all the hardware. Containers have only what the app needs in order to run.

With containers we can:

- Stand up (and tear down) instances quickly
- Distribute the whole app including the "server" in a ready-to-run state
- Run the container on my developer machine as well as production
- Easily manage automation to provision and orchestrate container instances

In short, containers use fewer resources, and orchestration is a lot easier with containers!

# Docker


We will be using Docker as our containerization technology in this course.

https://www.docker.com/what-docker

"Docker is the world's leading software containerization platform"

# Implementing a Full CI/CD Pipeline

## Installing Docker

# Installing Docker

You can find installation instructions for a variety of systems at https://docs.docker.com/install/

On CentOS, you can install it like so:

```
yum install docker
```

Docker includes a background process that you need to start and enable:

```
systemctl start docker
systemctl enable docker
```

Verify the installation with:

```
docker info
```

# Implementing a Full CI/CD Pipeline

## Docker Basics

# Docker Basics

For more info, check out the docker documentation: https://docs.docker.com/get-started/

Image - Package of the software and everything it needs in order to run.

Container - an instance of an image.

Images exist in a hierarchy, meaning most images have a parent image. Each new layer of the hierarchy adds a little bit of new configuration or functionality.

When we build a docker image for the train-schedule app, we will use a parent image that already has 90% of what we need. We will just add the train-schedule app itself.

# Building and Running with Docker

An image is defined in a **Dockerfile** and then created using the **docker build** command.

When you build, you give your image a name (and possibly tags):

- docker build -t <docker username>/<image-name> .

Once you build the image, you can create and run a container instance of it with docker run:

- docker run –d <docker username>/<image-name>

Two other commands we'll need:

- **docker ps** - see running containers
- **docker stop <container id>** - stop a container

# Docker Registries

A Docker registry is a place to store images.

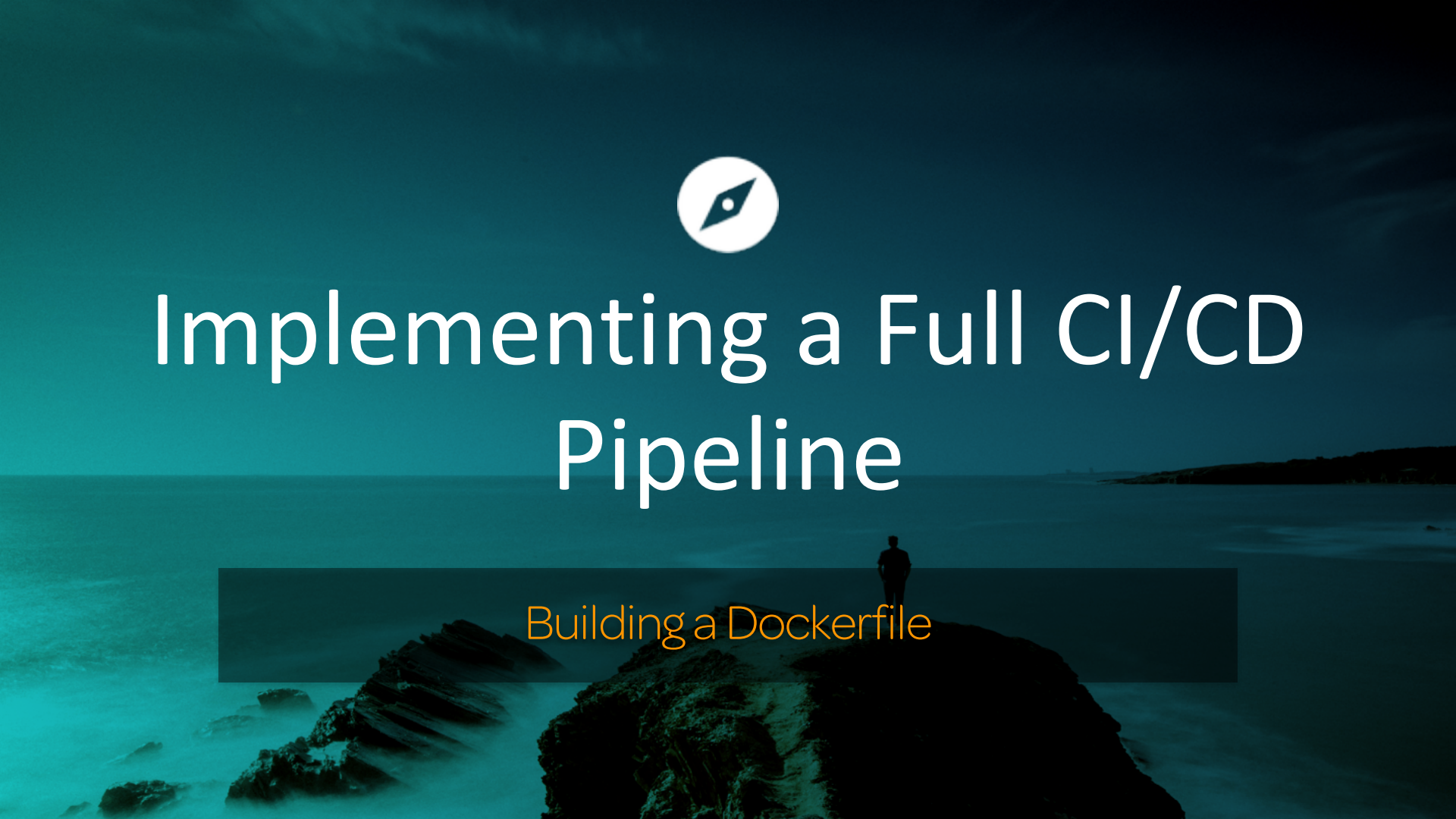
Once you docker build an image, you can docker push it to a registry.

Then you can docker run that image from anywhere that is set up to access that registry.

You can maintain your own private registries (as Docker containers, of course), or you can use the official cloud registry, Docker Hub (hub.docker.com).

You can authenticate with Docker Hub with:

```
docker login --username=<hub username> --email=<hub email>
```



# Implementing a Full CI/CD Pipeline

## Building a Dockerfile

# Dockerfile

The Dockerfile defines the docker image that will be built. It consists of a series of instructions for producing the image:

- FROM <image name> - sets the parent image
- WORKDIR <directory path> - sets the current working directory inside the container image for other commands
- COPY <source> <destination> - copies files from the host into the container image
- RUN <command> - executes a command within the container image
- EXPOSE <port> - tells docker that the software in the container listens on a particular port
- CMD <array of command arguments> - sets the command that is executed by the container when it is run

Check out the Dockerfile reference: https://docs.docker.com/engine/reference/builder/

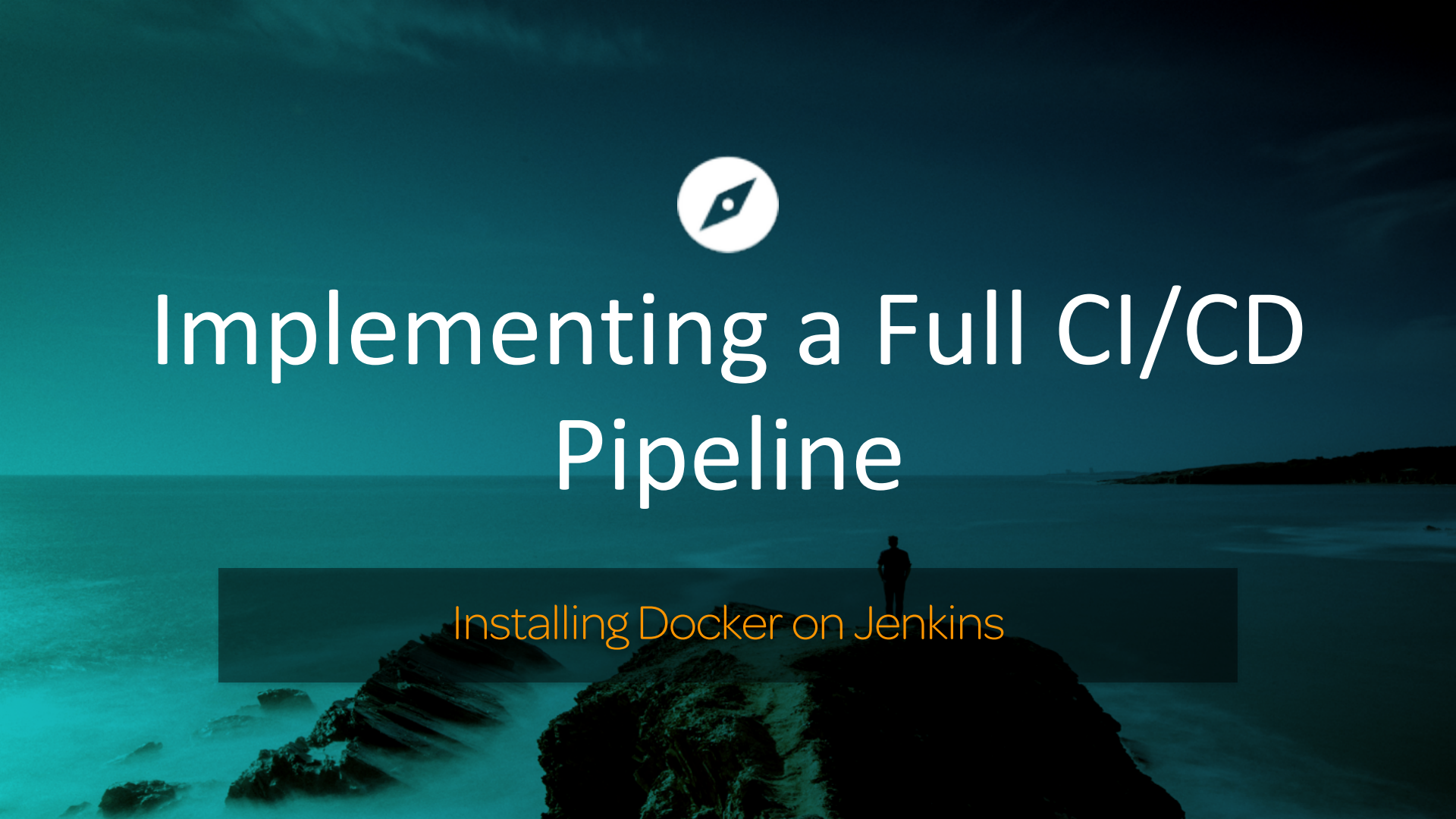
# Implementing a Full CI/CD Pipeline

## Running with Docker in Production

# Running with Docker in Production

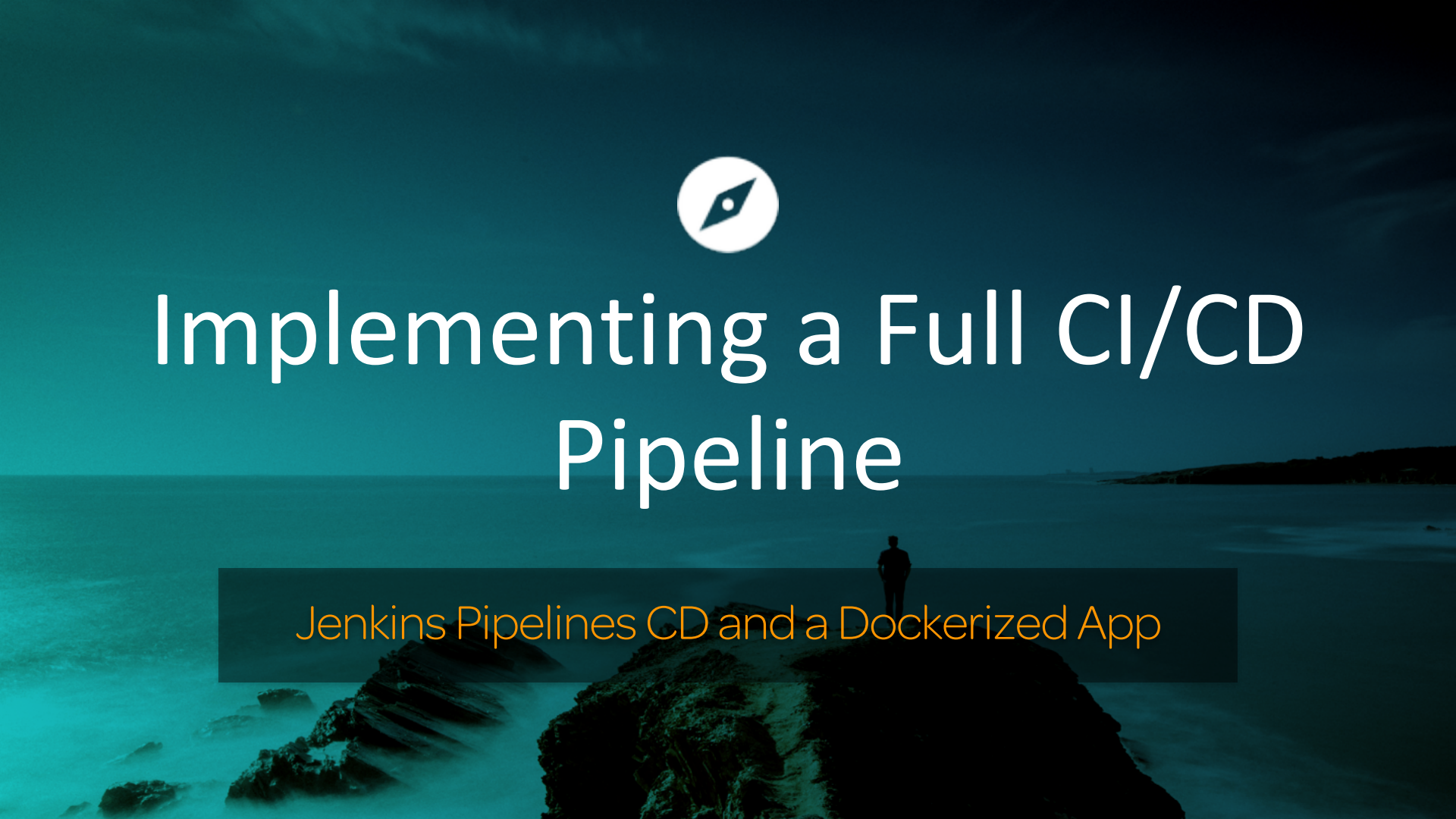
This is actually pretty easy:

- Install docker on the server
- Start and enable the docker service
- Use docker run with a restart policy:
    - docker run --restart always



# Implementing a Full CI/CD Pipeline

## Installing Docker on Jenkins



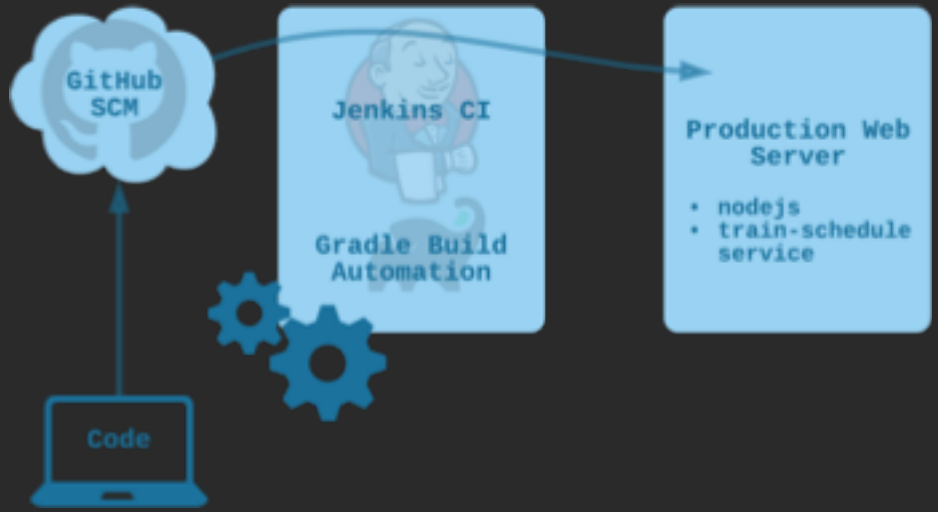
# Implementing a Full CI/CD Pipeline

Jenkins Pipelines CD and a Dockerized App

# The Current State

Here's where we now with Jenkins deployment:

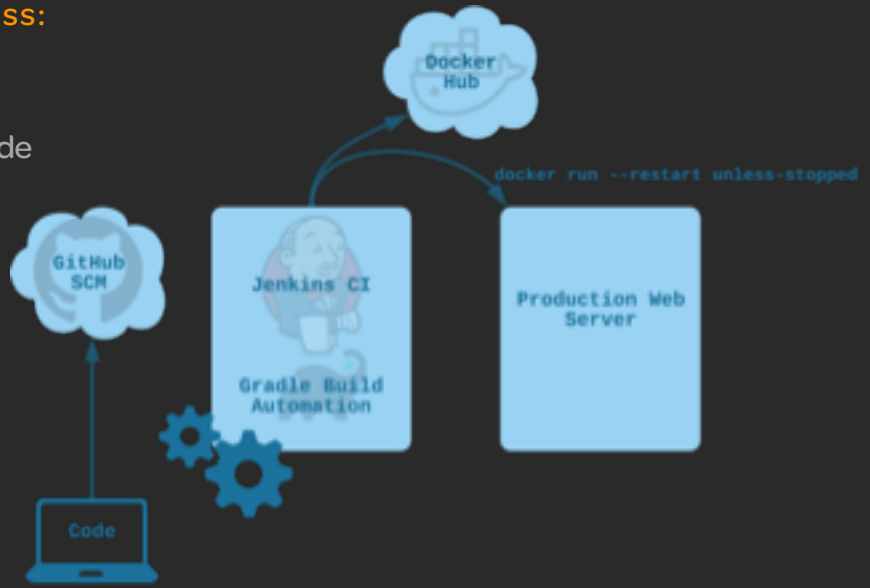
- Jenkins copies zip file to the server and extracts it
- Jenkins restarts the service



# Deploying a Docker Container

**Here's how we can dockerize the deployment process:**

- Jenkins produces a Docker image with the new code
- Jenkins pushes the new image to docker hub
- Jenkins pulls the new image on the production server
- Jenkins stops the container running the old code
- Jenkins runs **docker run** on the server to run the new image



# Implementing a Full CI/CD Pipeline

## Orchestration

# Orchestration

Orchestration uses automation to perform workflows and processes to manage our applications and infrastructure.

So far, we have built some great automation. With orchestration, these pieces of automation are like players in an orchestra. Orchestration tools make us like a conductor, guiding the orchestra without being too concerned with the details.

Some examples of orchestration:

- I scale up by asking my orchestration tools for 10 new instances
- My orchestration tools automatically scale up in response to load
- My orchestration tools automatically destroy and replace broken application nodes

# Kubernetes

Kubernetes is the orchestration tool that will be used for this course.

From kubernetes.io –

"Automated container deployment, scaling, and management."

We've already containerized our app,

so that sounds like exactly what we need!

# Implementing a Full CI/CD Pipeline

## Creating a Kubernetes Cluster

# Installing Kubernetes

There are many different ways to install Kubernetes. Check out their documentation for more info:
https://kubernetes.io/docs/setup/

In this course, we will be using Kubeadm. Kubeadm will help us get a functional Kubernetes setup up and running quickly and easily!

For more information on setting up kubeadm, check out:

https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/

# Installing Kubernetes with kubeadm

To set this up, you need at least TWO servers: a master and a node.

We can use cloud servers that already have the necessary packages installed: docker.io, kubelet, kubeadm, and kubectl

- Set up the master with:

  sudo kubeadm init --config=/path/to/config/file.yml

- Configure kubectl for use by your user with the three commands given to you by kubeadm init
- Apply flannel pod networking config on the master:

  kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/v0.9.1/Documentation/kube-flannel.yml

- Join nodes to the cluster

# Implementing a Full CI/CD Pipeline

Kubernetes Basics

# Kubernetes Objects

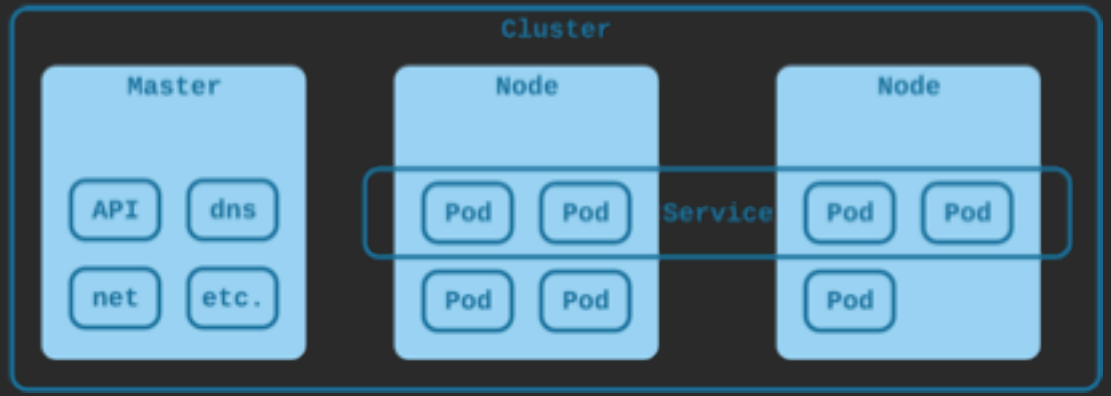
Cluster – A master and one or more nodes.

Master – Runs control processes and Kubernetes API.

Node – A worker machine.

Pod – A running process on a node. For example, a running container instance or group of containers.

Service – An abstraction defining a set of pods and a means to access them.



# Kubernetes Deployments

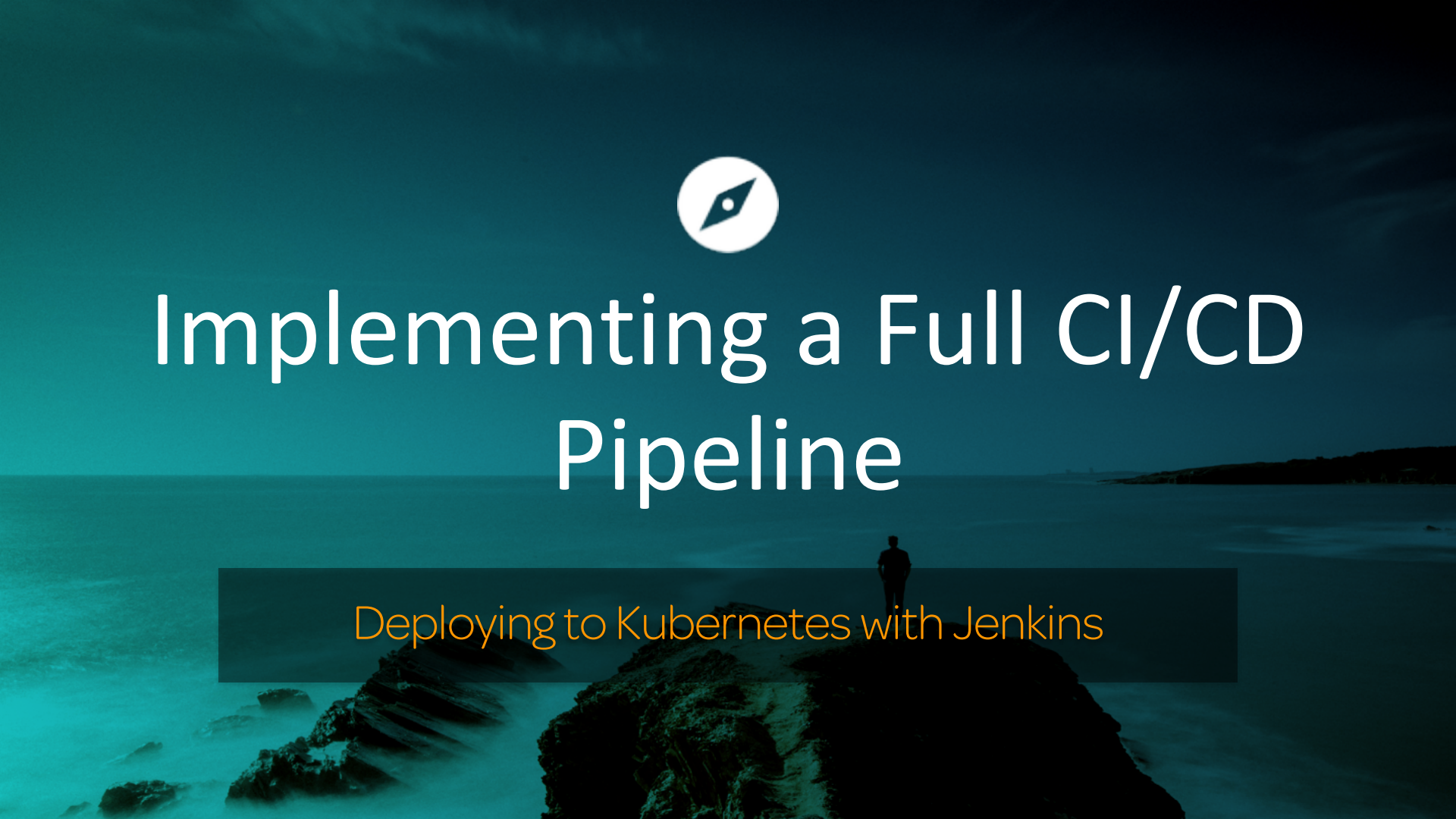
A deployment is a declarative definition for Pods and ReplicaSets.

Deployments let you do things like:

- "I want X replicas of Y Docker image" – Kubernetes will orchestrate the process of standing up the desired number of pods across multiple nodes
- "I want to change the number of replicas" – If you change the number of replicas in a deployment, Kubernetes will create or destroy pods to meet the new desired number
- "I want to deploy a newer Docker image with code changes" – Kubernetes will orchestrate the rollout of the new version with zero downtime, gradually replacing the old with the new

In this course, we will be standing up a deployment and a service using YAML configuration. You can apply YAML configuration with:

```
sudo kubectl apply -f /path/to/yaml/file.yml
```



# Implementing a Full CI/CD Pipeline

## Deploying to Kubernetes with Jenkins

# Deploying to Kubernetes with Jenkins

Where we are now:

- We have a dockerized app
- We have a Jenkins pipeline that builds a Docker image and pushes it to Docker Hub
- We have a Kubernetes cluster where we can deploy the Docker image manually with a Kubernetes deployment

The next step is to run the Kubernetes deployment as part of our Jenkins pipeline!

# The Kubernetes CD Jenkins Plugin

To accomplish this, we can use the Kubernetes Continuous Deploy plugin in Jenkins:
https://jenkins.io/doc/pipeline/steps/kubernetes-cd

# Implementing a Full CI/CD Pipeline

## Monitoring

# Monitoring

Monitoring means collecting and presenting data about the performance and stability of applications and infrastructure.

This is an important part of a CI/CD Pipeline. In order to deploy frequently, you need to be able to identify and fix problems quickly!

Two major types of data you can collect:

- Infrastructure monitoring – collecting data about the health, performance, and capacity needs of infrastructure: CPU load, Memory usage, disk usage, etc.
- Application performance monitoring (APM) – collecting data about the health and performance of the apps themselves: Requests per minute, error rate, average response time, etc.

# Monitoring

Two major ways of presenting and using the data:

- Aggregation – Usually means displaying the data in the form of graphs and dashboards designed to provide actionable information
- Alerting – Pushing notifications to people about important events in real time

# Prometheus and Grafana

To implement monitoring in this course, we will be using Prometheus and Grafana.

Prometheus gathers and stores monitoring data. It also provides alerting.

Grafana will let us display and build dashboards for the data.

# Implementing a Full CI/CD Pipeline

## Installing Prometheus and Grafana

# Helm Installation

Helm is a tool for installing software using Kubernetes charts.

Charts make it easy to install software on a Kubernetes cluster in a standardized configuration.

The Kubernetes charts repo includes great charts for both Prometheus and Grafana!

Install steps:

- Install and initialize Helm
- Clone the Kubernetes charts repo from GitHub
- Create values.yml files for any special settings we want to use
- Install Prometheus and Grafana with helm install
- Set up a Prometheus datasource in Grafana and verify that it can connect

# Implementing a Full CI/CD Pipeline

## Cluster Monitoring

# The Kubernetes All Nodes Dashboard

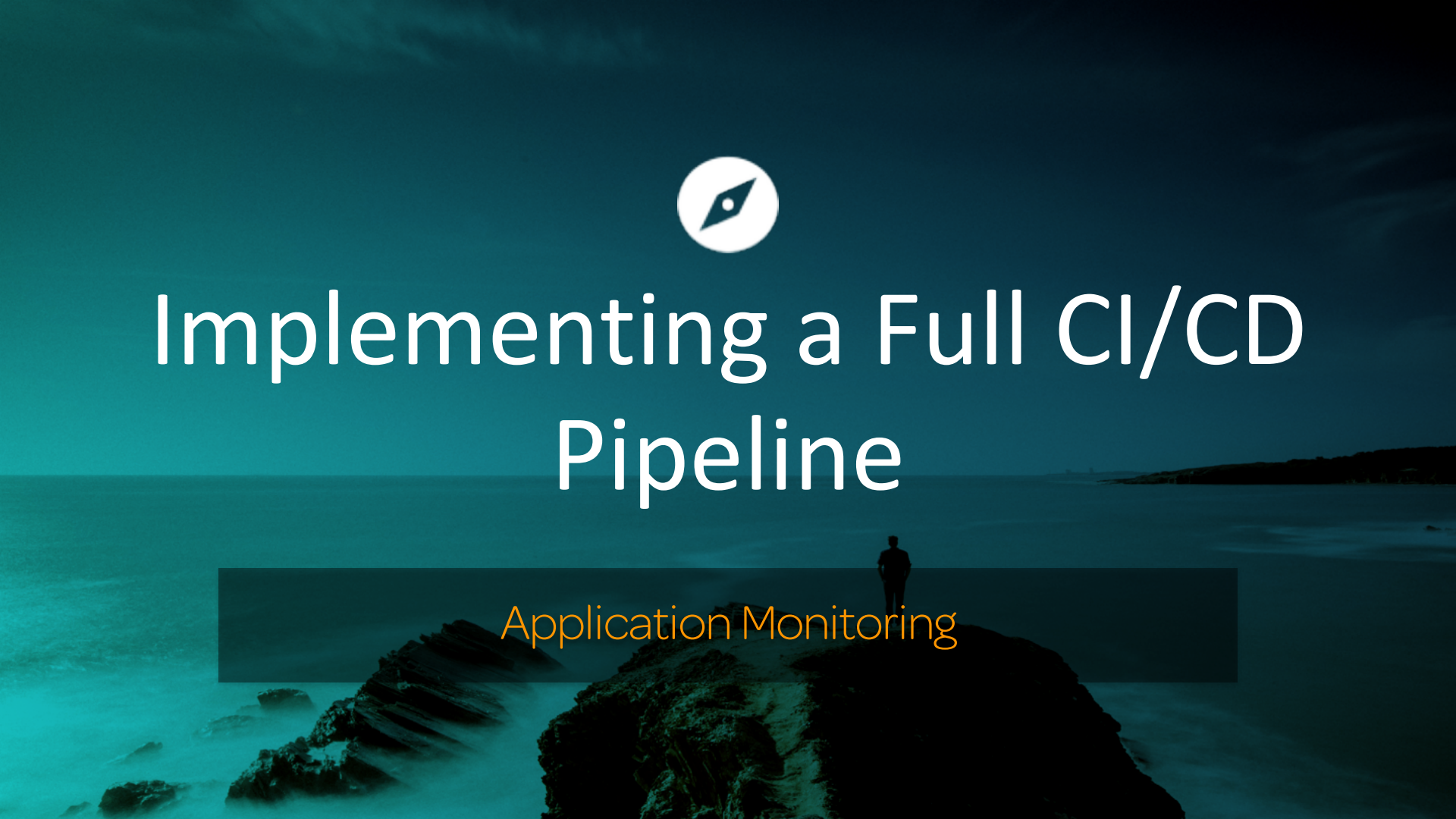
A quick and easy way to get started with Grafana is to use community dashboards. These are pre-made dashboards created by the Grafana community.

Find them at https://grafana.com/dashboards.

A useful one that can help us monitor our cluster is the Kubernetes All Nodes dashboard. It is great for gaining insight into the health of a cluster, and can be a big help with capacity planning.

https://grafana.com/dashboards/3131

With the standard helm configuration for Prometheus and Grafana, all we have to do is import the dashboard and it works!



# Implementing a Full CI/CD Pipeline

## Application Monitoring

# Getting Data from Applications

One way to get monitoring data from applications is to instrument the applications themselves to provide data.

The languages and frameworks that get used determine how an app gets instrumented. The Train Schedule app is a nodejs app, and there is a nodejs Prometheus client library called prom-client that is being used to instrument the app.

The app serves monitoring data on the /metrics endpoint, which Prometheus can automatically detect.

We can add a service annotation in Kubernetes which will tell Prometheus to scrape data from the /metrics endpoint of all instances represented by that service: prometheus.io/scrape: 'true'

# Implementing a Full CI/CD Pipeline

## Alerting

# Alerting

Alerting allows us to set up notifications when certain things happen, according to monitoring data.

For example, we could set up an alert to send an email or post a slack message when average response times go above a certain threshold. This would enable us to act quickly to fix the problem.

We can set up alerts in both Prometheus and Grafana. Since we are already using Grafana for visualization, we will demonstrate setting them up in Grafana.

# Implementing a Full CI/CD Pipeline

## Kubernetes and Self-Healing

# Self-Healing

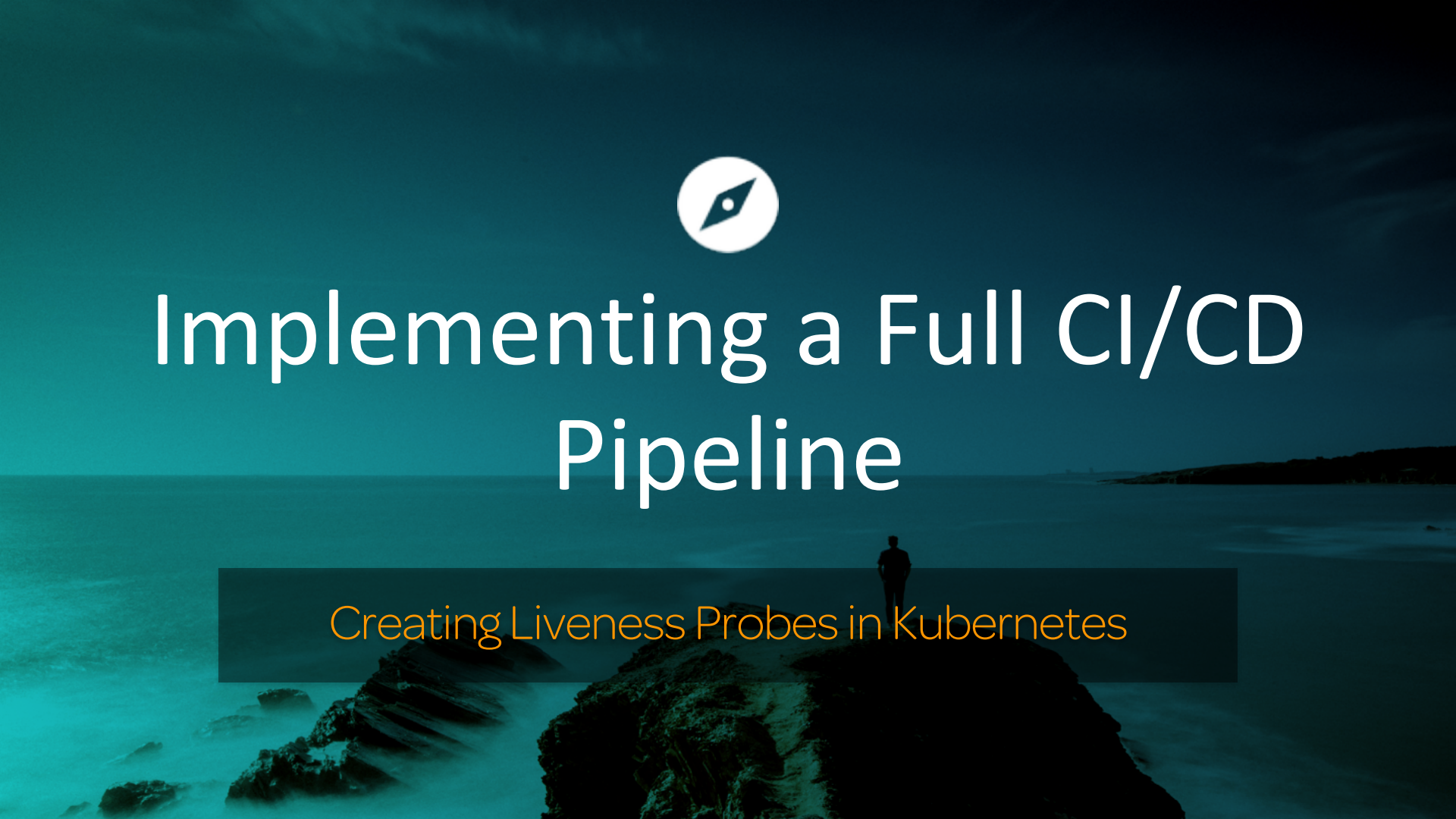
A Self-Healing system is able to detect when something is wrong within itself, and automatically take corrective action to fix it without any kind of human intervention.

# Kubernetes and Container State

Docker containers stop when their main process exits. If a container exits, Kubernetes will automatically restart it by default.

For information on other restart strategies, check out Kubernetes restart policies:
https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy

So, if the nodejs app crashes on a container, that container will be automatically restarted.



# Implementing a Full CI/CD Pipeline

## Creating Liveness Probes in Kubernetes

# Liveness Probes

An application can become unhealthy without the main process exiting. In these cases, we need something more sophisticated in order to properly self-heal.

Kubernetes allows us to create liveness probes, which are custom checks run periodically against containers to detect whether or not they are healthy. If a liveness probe determines that a container is unhealthy, that container will be restarted.

# Implementing a Full CI/CD Pipeline

## Kubernetes and Autoscaling

# Autoscaling

Autoscaling means automatically allocating resources based upon resource need over time.

In Kubernetes, this usually means creating additional pod replicas when resource usage is high, and removing unneeded replicas when resource usage is low.

# Autoscaling in Kubernetes

Autoscaling is driven by metrics such as:

- CPU usage
- Memory usage
- Requests per second

In order to autoscale, Kubernetes needs to first collect the necessary metrics. Then it needs to be configured to automatically manage the number of replicas in response to changes in those metrics.

# Implementing a Full CI/CD Pipeline

## Horizontal Pod Autoscalers in Kubernetes

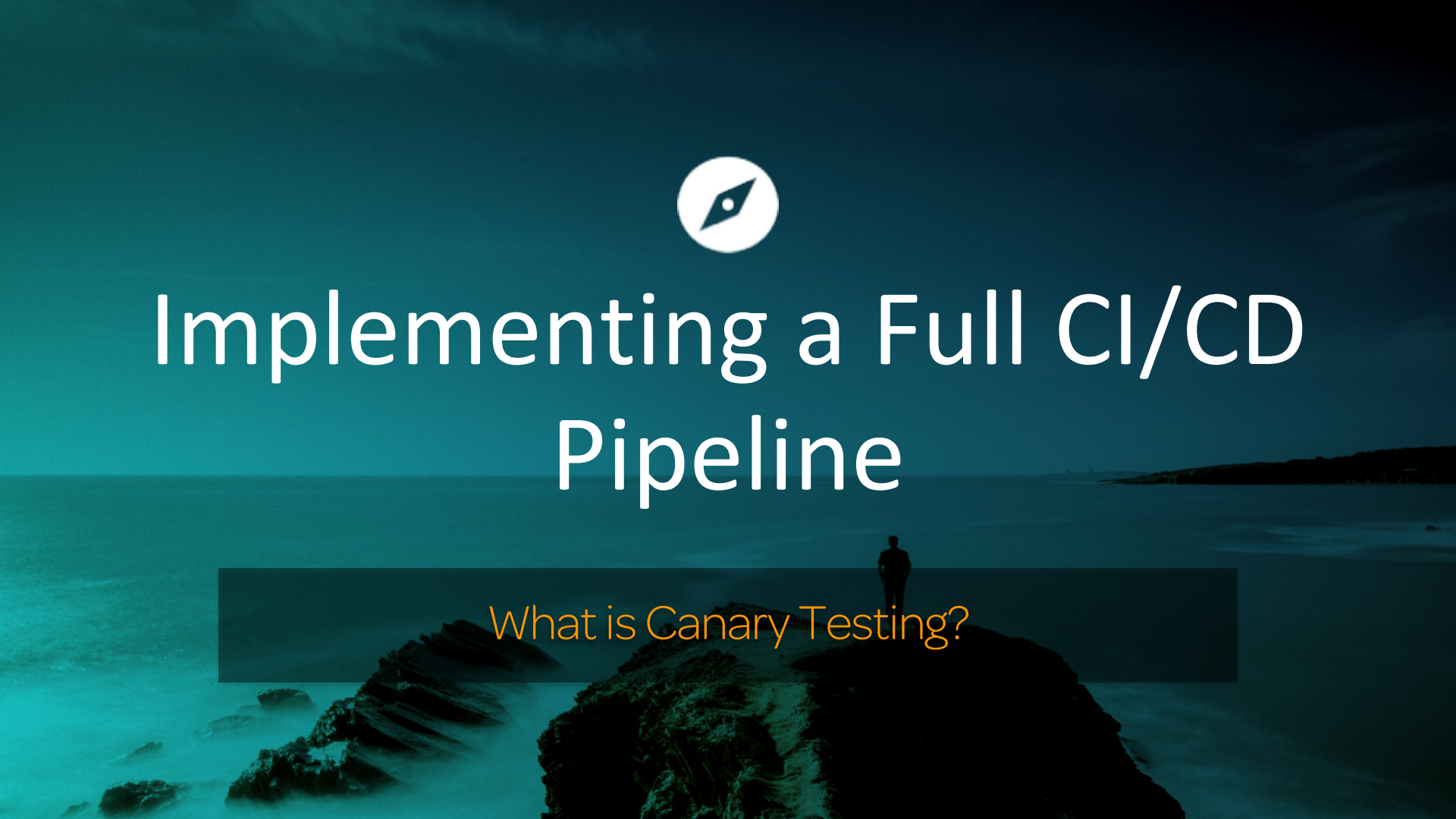
# Horizontal Pod Autoscalers in Kubernetes

Autoscaling can be implemented in Kubernetes using a Horizontal Pod Autoscaler (HPA):
https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

HPAs work with the Metrics API to periodically check metrics and perform autoscaling according to the HPA configuration.

Steps to implement autoscaling based on CPU usage with an HPA:

- Install metrics API
- Set a resource request for the pods that will be autoscaled
- Create an HPA



# Implementing a Full CI/CD Pipeline

## What is Canary Testing?

# The Need for Canary Testing

Unit tests, integration tests, and even manual tests run in a staging environment can work together to provide a very robust testing solution.

But all of these methods suffer from one problem: The tests never perfectly replicate real-world usage of the code no matter how hard we try to simulate it.

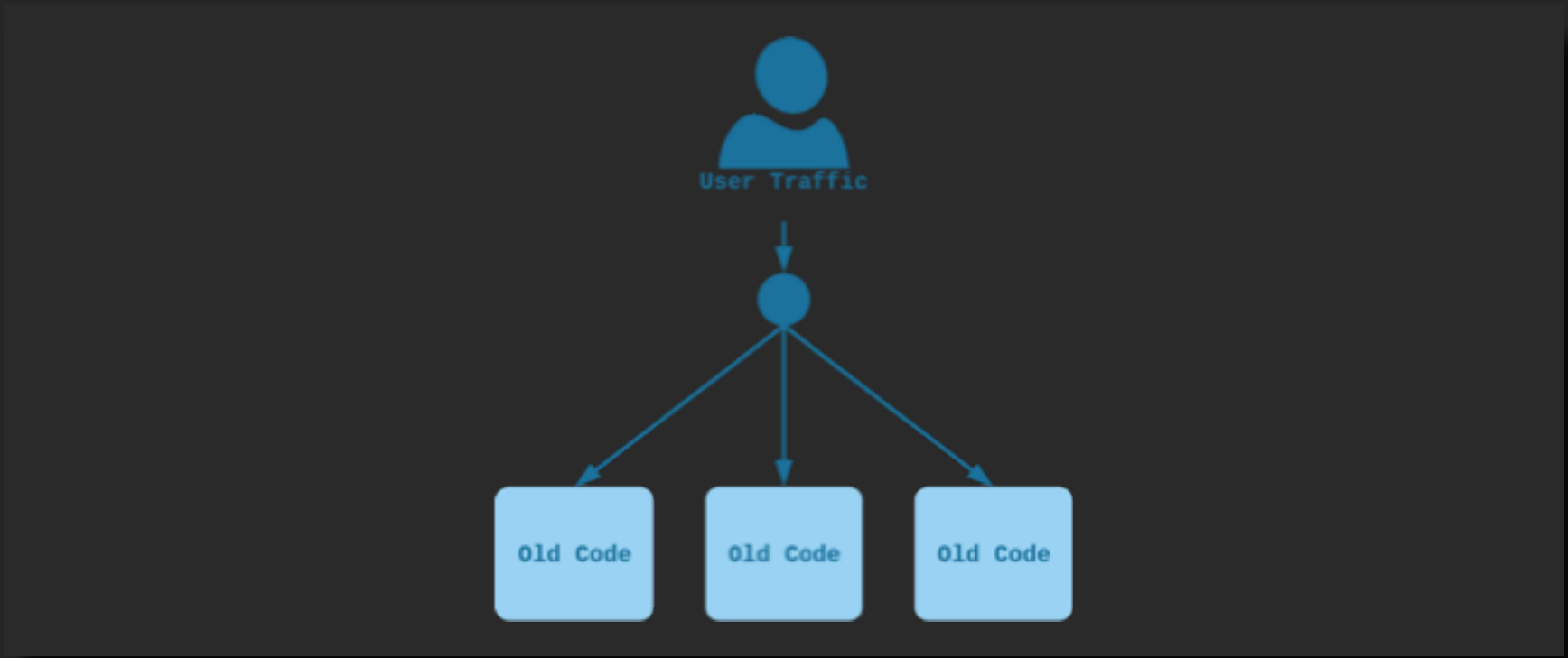
This is where canary testing comes in!

# Canary Testing

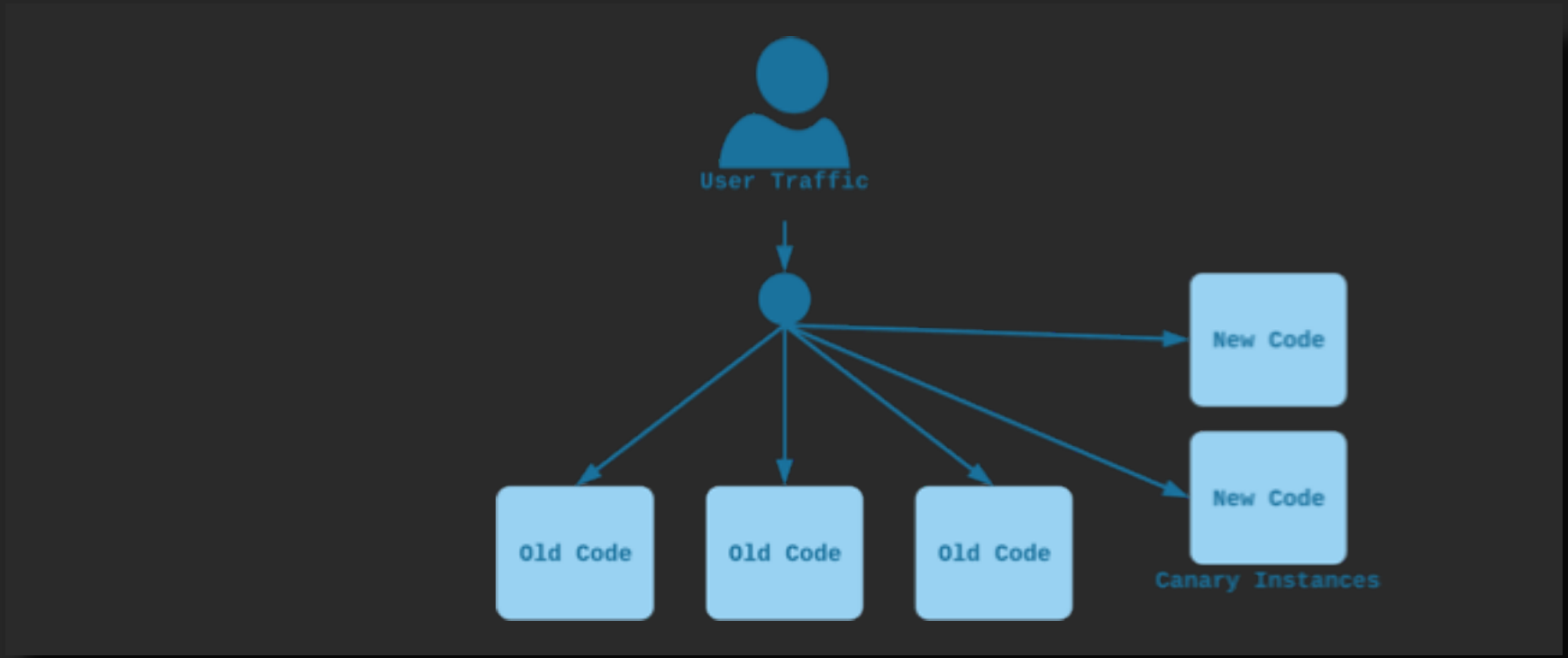
Canary Testing means pushing new code to a small group of real users, usually with those users being unaware that they are using new code.

This allows us to see how our code performs under real-world usage, while limiting the impact of any bugs to a small number of users.

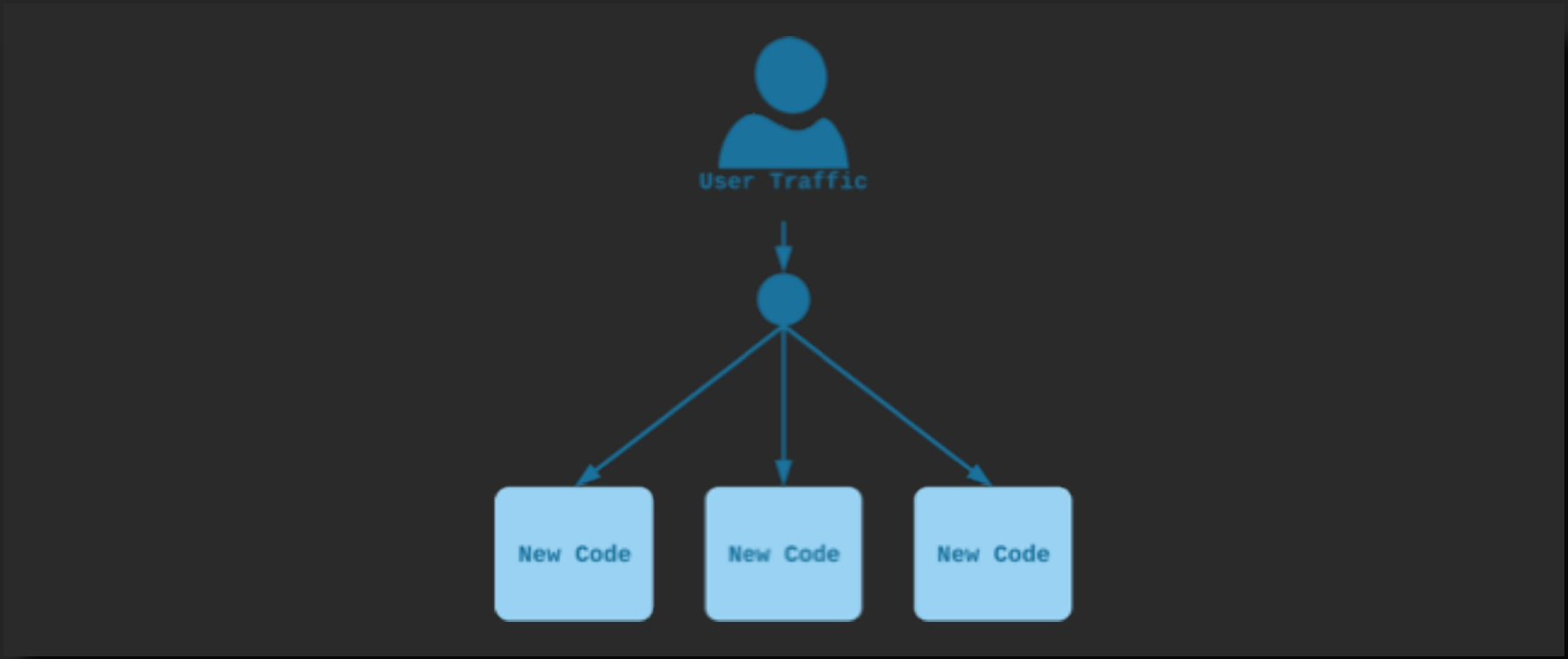
# Canary Testing

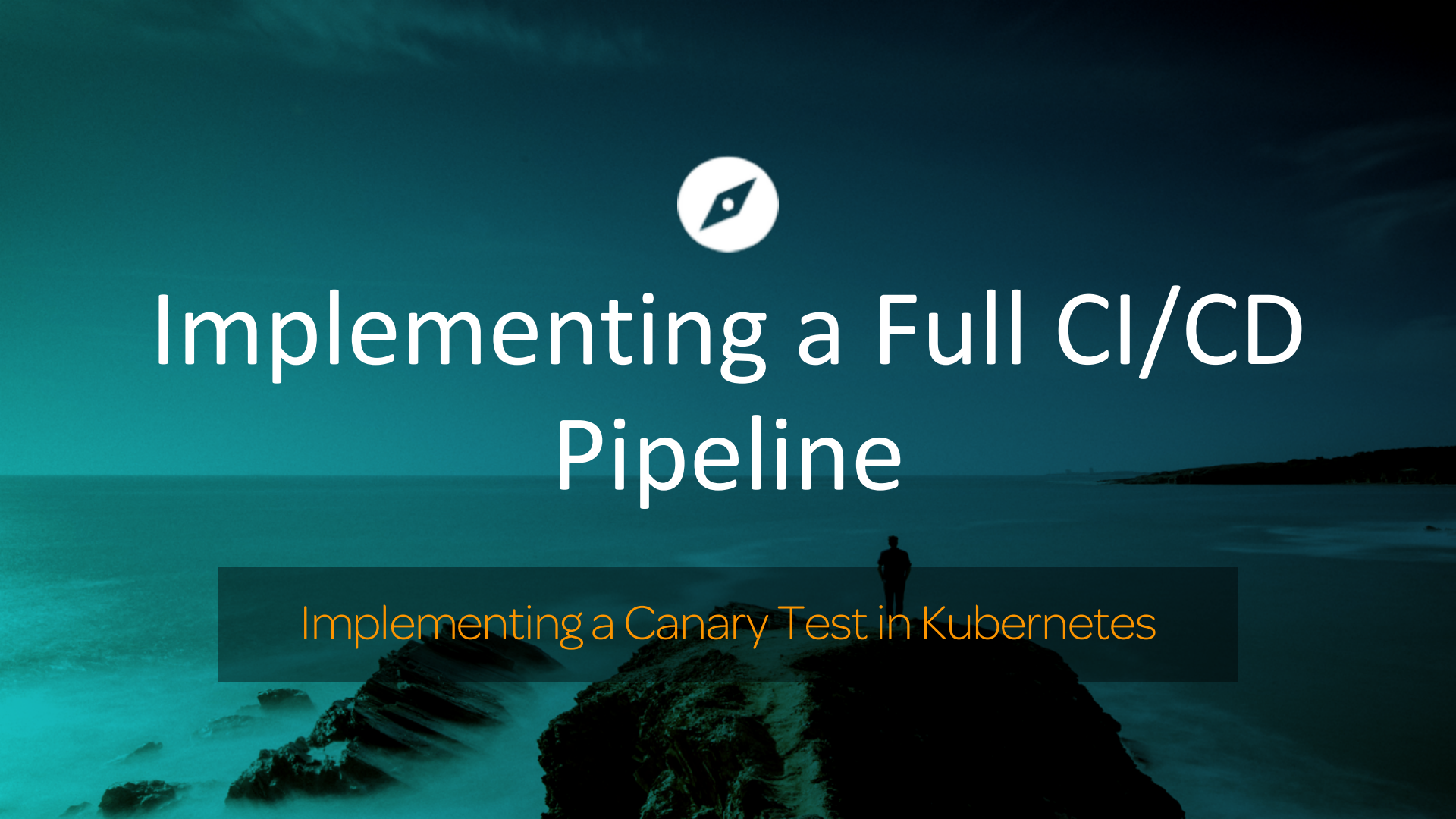


# Canary Testing



# Canary Testing





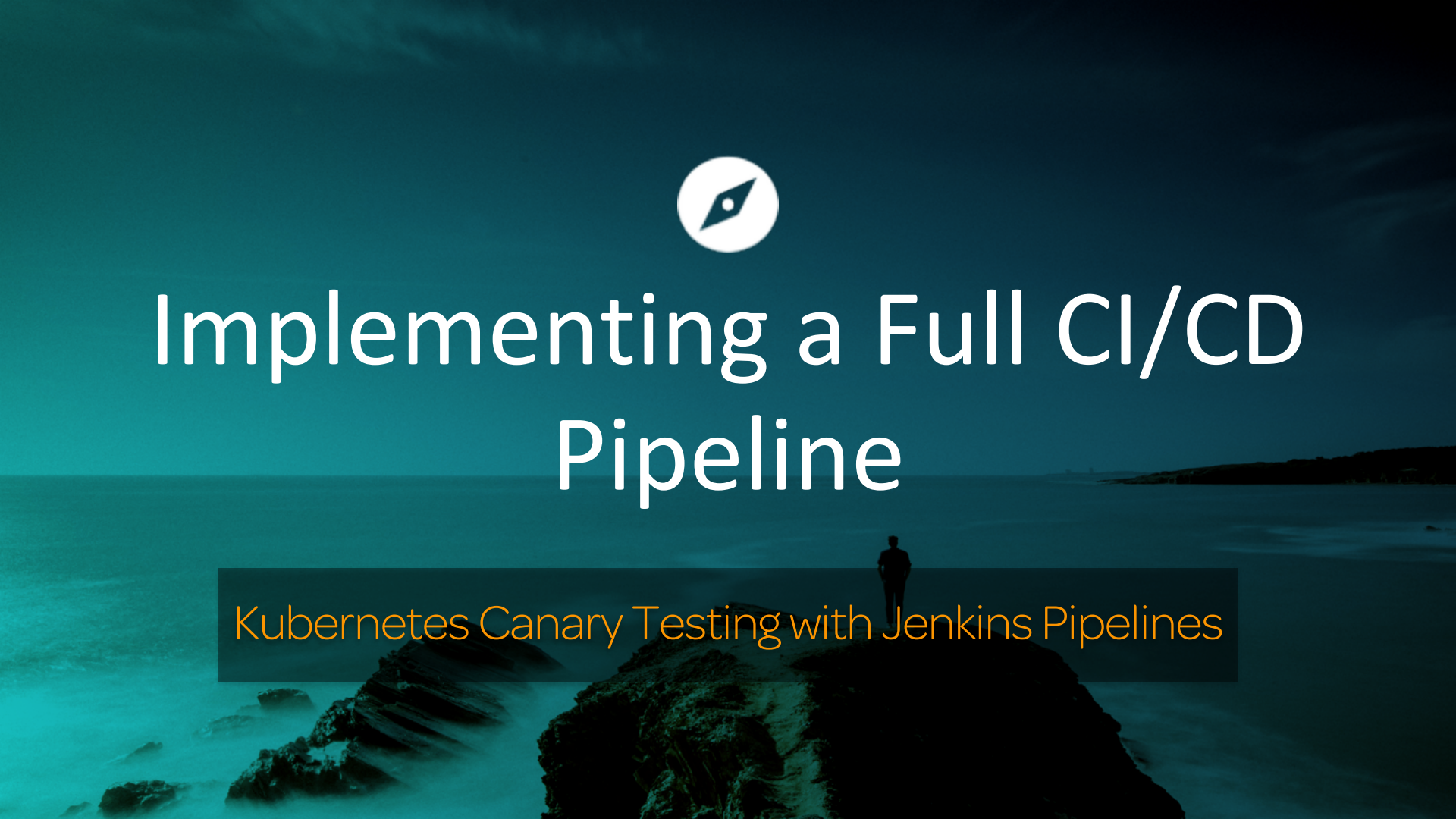
# Implementing a Full CI/CD Pipeline

## Implementing a Canary Test in Kubernetes

# Implementing a Canary Test in Kubernetes

We can implement a canary test in Kubernetes by:

- Adding labels to pods that will differentiate between canary and normal pods
- Creating a new deployment for the canary pods
- Rolling out the new code to the normal pods by updating the regular deployment after canary testing is complete


Implementing a Full CI/CD Pipeline
Kubernetes Canary Testing with Jenkins Pipelines

# Canary Testing in Kubernetes with Jenkins Pipelines

To implement the canary test in a Jenkins pipeline, we need to do the following:

- Add a stage to execute the canary deployment
- Add steps to the production deployment stage that will clean up canary pods when they are no longer needed


Implementing a Full CI/CD Pipeline
Fully Automated Deployment

# Fully Automated Deployment

Right now, our Deployment Pipeline includes a human approval step.

But what if we removed that step?

We would need confidence in our:

- Automation
- Testing
- Monitoring

Fully Automated deployment is not necessary or desirable for everyone, but in some situations it can be great!

# Implementing a Full CI/CD Pipeline

## Next Steps

# Next Steps

Learn more about any of the tools covered in this course!

Linux Academy has courses specifically covering many of these tools:

- Git – Git Quick Start, Source Control with Git
- Jenkins – Jenkins Quick Start, Jenkins and Build Automation, Certified Jenkins Engineer
- Docker – Docker Quick Start, Docker Deep Dive, Docker Certified Associate Prep Course
- Kubernetes - Certified Kubernetes Administrator

Also check out the LPI DevOps Tools Engineer Certification!

# Next Steps

Learn about a particular cloud platform!

Cloud platforms support CI/CD Pipelines in a variety of ways.

Linux Academy offers relevant content for:

- Amazon Web Services
- Azure
- Google Cloud Platform