Learning Objectives

Learners will be able to...

- **Explain deployment targets**
- **Automate code deployment**

info

Make Sure You Know

Learners should be familiar with `nodejs` and its basic use.

Limitations

Assignment will require an active GitHub repository for activities.

Continuous deployment

Continuous deployment is a step in automation in software development where code changes are released automatically into production or testing environments after all tests and quality checks have been successfully completed.

This is the final step in the **CI/CD pipeline** and should be executed after all tests and quality checks are successfully completed. Once all the tests pass, the system pushes the updates to end users. This can be beneficial for projects with large user bases that require quick bug fixes or improvements.

Benefits of using Continuous deployment:

- **Shorter mean time to resolution (MTTR)**
  - With automation, every small change/bug fix can go through the automated pipeline and be released quicker. This allows developers to quickly isolate and address any issues with their code and release features for customers much faster than with targeted releases.
- **Cost Savings**
  - By automating processes such as deployment and testing, manual labor is no longer required, reducing overhead costs.
- **Less Service Disruption**
  - Due to smaller changes and often releases there are fewer unintended consequences and less disruption for users.
- **Quick Fault Isolation**
  - Selecting smaller changes makes it easier to find faults sooner
- **Smaller backlogs**
  - The backlog of non-critical defects is lower because defects are often fixed before other feature pressures arise.

Continuous Deployment uses automation tools, like GitHub Actions, to streamline the process of moving code from development to production or test environments. This can help us save time and resources while assuring high-quality products.

Checkpoint

Automate deployment

In the same way we automated other actions in our CI/CD pipeline, like building and testing, deployment can be automated as well. As changes are merged to the `master` (`main`) branch, our app with our new changes can be automatically deployed directly from GitHub or any other CI/CD system.

Let's look at an example of deploying our app to GitHub pages using GitHub Actions.

Navigate to your workflow `.yml` file to add deployment automation

```
  deploy:
    needs: build
    # Grant GITHUB_TOKEN the permissions required to make a
Pages deployment
    permissions:
      pages: write      # to deploy to Pages
      id-token: write   # to verify the deployment originates
from an appropriate source

    # Deploy to the github-pages environment
    environment:
      name: github-pages
      url: ${{ steps.deployment.outputs.page_url }}

    # Specify runner + deployment step
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to GitHub Pages
        id: deployment
        uses: actions/deploy-pages@v1
```

This code snippet defines an action named `deploy`. This action `needs` the `build` stage to be successfully completed before beginning.

Kubernetes

If Kubernetes is your preference, it provides several APIs for automating software deployments, such as the `k8s-deploy` action which allows users to easily configure their applications for deployment into a Kubernetes namespace.

Here is an example code snippet of using the `k8s-deploy` action.

```
- uses: Azure/k8s-deploy@v4
  with:
     namespace: 'myapp'
     manifests: |
        dir/manifestsDirectory
     images: 'contoso.azurecr.io/myapp:${{ event.run_id }}'
     imagepullsecrets: |
        image-pull-secret1
        image-pull-secret2
```

This code snippet will trigger the `k8s-deploy` action, which will deploy the specified manifests and images into the given Kubernetes namespace.

GitHub Actions provide a simple and powerful way to set up automation for this task, while Kubernetes offers more advanced deployment options. Whichever tool you choose, automating your deployments will save time and effort for developers so that they can focus their efforts on developing quality code.

Checkpoint

Downtime

Downtime in software development is a period of time when the system or application being developed is not accessible to customers, users, or other stakeholders.

Downtime can be caused by various things such as:
- Server and hardware failure
- Software bugs
- Network issues
- Changes in the code causing unexpected problems

Potential downtime is largely dependent on your infrastructure and the released changes of your application.

CI/CD approaches try to **minimize downtime** as much as possible. We'll talk about potential deployment strategies for this a bit later.

Generally, regarding downtime, we want to keep the following things in mind:

1. **Support and design a system that is redundant and can handle failover.**
    - A good approach for this is a **rolling deployment**, which involves deploying new nodes and switching the load balancer or restarting services subsequently.
    - The goal is to minimize the impact of any service interruption in the production environment caused by the deployments.
2. **Deploy smaller changes that are comparable with the rest of the system.**
    - The **contract-first** approach can be used to ensure that all component contracts are in place before deploying changes.
    - This helps to avoid deploying dependencies between components and reduces the chances of any issues due to incompatibilities between different versions of software.
3. **Ensure that database structures are backwards compatible with previous versions.**
    - This ensures that in case of a rollback, it can be done quickly and without compromising existing or future data integrity.

It is important to have processes in place to help **reduce downtime** when developing software so we can detect any issues quickly and take corrective action.

Checkpoint

Blue/Green deployment

Blue/green deployment is a CI/CD (Continuous Integration and Continuous Deployment) strategy that enables developers to deploy new versions of software applications with minimal risk and downtime.

This strategy involves creating **two separate, but identical environments**, one running the current version of the application (the blue environment) and one running the new version of the application (the green environment).

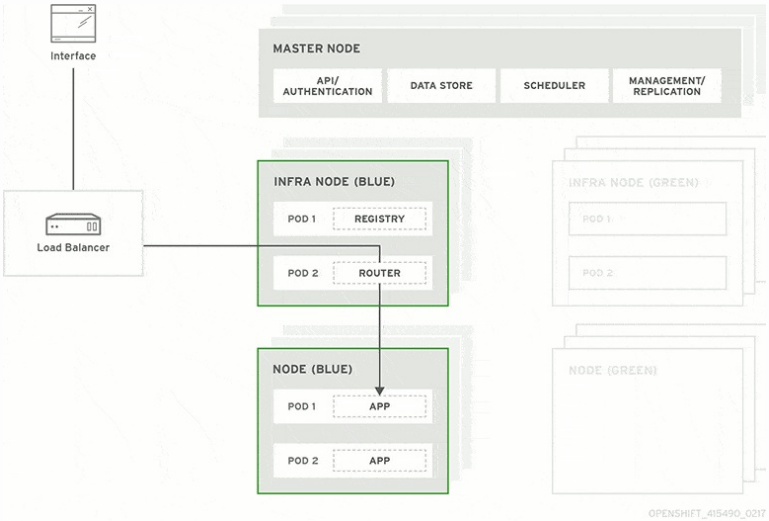

Diagram of load balancer switching between blue and green deployment environments.

By deploying new versions to a separate environment first, ops teams can thoroughly test and validate each release **before switching production traffic over** from the blue environment to the green one.

The blue/green method also **reduces deployment risk** by providing an easy rollback process if something goes wrong; once testing has been completed successfully on the green environment, live application traffic can be directed to it and then the deprecated blue environment can be removed or re-purposed.

Blue/green deployment helps to increase application availability, as there is no downtime when switching from one version to the other. It also allows developers to quickly and easily roll back to a previous version if necessary, allowing them to maintain control over their release cycle without sacrificing reliability or stability.

Checkpoint

A/B Testing

One useful outcome from automated deployment strategy and flexible infrastructure provision is the ability to do **A/B testing**.

A/B testing, also known as split testing, refers to a randomized experimentation process that involves **comparing two or more versions of a variable** (web page, page element, etc.) and showing them to different segments of website visitors at the same time to determine which version leaves the maximum impact and drives business metrics.

This can be used to test changes with little risk to business while giving valuable information about **what users like best**.

CI/CD environments provide a great setting for **A/B testing** for many businesses because changes can be deployed quickly and easily, allowing tests to be put up in minutes and monitored for weeks or months before making any substantial changes.

A/B testing is especially helpful when we want to add new features or improve ones you already have. It allows the developers to **use data and user feedback** instead of guesswork to make decisions.

For example, if a company wants to test two different versions of a website's homepage layout, they could show each version to different groups of visitors. Then, they could track things like **clicks**, **sign-ups**, **purchases**, etc. to see which version has the most effect on how users act.

When companies test ideas like this, they learn a lot about how their customers respond, which helps them improve their products.

Checkpoint

Rollback

Rollbacks are an important part of the software testing process because they allow developers to quickly and easily **reverse any changes they have made in the system**. Using the CI/CD approach, rollbacks are especially beneficial as they are fast, simple and invisible for users.

With the CI/CD method, all builds of the system are **stored in a repository** and then **deployed as needed**. This means that if a bug gets into production, a rollback can be done quickly and easily by:
* Navigating to your project's repository
* Locating the version of the build you'd like to revert to
* Triggering the desired version of the build to be deployed.

This reverts the system back to the same state as the previous version.

Alternatively, you can also perform a rollback by pushing a `rollback commit` to the commit corresponding to our desired version in order to trigger a new pipeline build. This becomes useful when you need to roll back changes other than the latest ones.

Checkpoint