Continuous Integration and Delivery (CI/CD) with GitLab

By Josh Samuelson, Senior Engineer at Dropbox, Delivery Engineering

Overview

Software development is not just about writing code; it involves **careful planning, testing, and deployment**. Without proper execution of these processes, users may face **broken features** and **frustrating downtimes**.

Key Concepts:

1. **Continuous Integration (CI)**: Automates the process of merging and testing code, ensuring that changes are validated consistently.
2. **Continuous Delivery (CD)**: Automates the deployment process, ensuring reliable releases to production with minimal manual intervention.

Benefits of CI/CD:

- **Thorough Automated Testing**: Every change is tested automatically, reducing bugs and issues in production.
- **Reliable Deployment**: Ensures that deployments happen in a predictable and stable manner.
- **Improved Developer Productivity**: Developers focus more on writing code and less on manual testing and deployment processes.
- **Increased Developer Satisfaction**: CI/CD reduces the stress of deployment failures and bug handling post-release.

GitLab for CI/CD

GitLab provides an integrated solution for implementing CI/CD pipelines. Compared to other platforms, GitLab offers a **robust and fully-featured** CI/CD solution that simplifies setup while maintaining flexibility for more complex workflows.

Advantages of GitLab:

- **Integrated CI/CD Features**: GitLab provides everything in one place, from version control to pipeline execution.

- **User-friendly Setup**: Easier to configure CI/CD pipelines compared to other platforms, without compromising on functionality.

End-to-End CI/CD Pipeline in GitLab

In this course, we'll focus on setting up an **end-to-end CI/CD pipeline** in GitLab, covering all the major stages from integration to deployment. The pipeline will ensure **automated testing** and **reliable delivery** of the application.

Example GitLab CI/CD Pipeline Configuration

Here is a basic `.gitlab-ci.yml` file to set up a CI/CD pipeline:

```yaml
stages:
  - build
  - test
  - deploy

build:
  stage: build
  script:
    - echo "Building the application..."

test:
  stage: test
  script:
    - echo "Running tests..."
    - ./run_tests.sh

deploy:
  stage: deploy
  script:
    - echo "Deploying the application..."
    - ./deploy.sh
  when: manual  # Deployment can be triggered manually
```

Key Stages of the Pipeline:

1. **Build Stage**:
    - Prepares the application by building necessary components.
    - Example: Compiling code or bundling assets.
2. **Test Stage**:
    - Runs automated tests to ensure the code functions correctly.
    - Example: Unit tests, integration tests, or functional tests.
3. **Deploy Stage**:
    - Deploys the application to the target environment.
    - Example: Deploying to a server or cloud infrastructure.

Conclusion

Implementing CI/CD practices through GitLab can significantly **improve software quality** and **streamline deployment processes**. If you're looking to boost your team's productivity and ensure consistent, high-quality software delivery, CI/CD is the solution.

Understanding GitLab for Code Management and CI/CD

What is GitLab?

GitLab is a **web-based code management** system that also supports **continuous integration (CI)** and **continuous delivery (CD)**. It offers both **free and open-source** versions, as well as **commercial solutions** with additional features or hosted services.

Key Points:

- **Initial Focus**: Originally focused on **code management**.
- **Expanded Features**: Now includes **CI/CD systems**.
- **GitHub Comparison**: GitLab is a primary competitor to GitHub but distinguishes itself by offering features not found in GitHub, aiming to be an **end-to-end solution**.

Why Choose GitLab?

- **Versatile**: You can use it for source control only, or utilize its full CI/CD capabilities.
- **Feature-Rich**: Offers capabilities that can replace other tools, such as Jenkins for CI/CD.

GitLab's CI/CD Overview

Continuous Integration (CI):

CI involves **automating the process of integrating code changes** into the main codebase. It is designed to be a **continuous and automated process**. The main goal is to ensure that updates or new features don't introduce bugs or break the application.

What is Integration?

- **Code Integration**: Merging code changes from different developers into the main codebase.
- **Feature Integration**: Ensuring new features work seamlessly with the existing ones.

Automation in CI:

CI needs automated testing and validation, which can vary from:

- **Syntax checks**: Simple validation.
- **User interaction simulation**: Full end-to-end testing.

Example CI Pipeline in GitLab:

In GitLab, CI revolves around **pipelines** where code changes move through a series of **automated steps** before merging.

```
stages:

  - lint

  - test

  - build



lint:

  stage: lint

  script:
    - echo "Linting the code..."

    - ./run_linter.sh


test:

  stage: test

  script:

    - echo "Running tests..."

    - ./run_tests.sh


build:

  stage: build

  script:

    - echo "Building the application..."

    - ./build.sh
```

Continuous Delivery (CD):

CD is closely related to CI but focuses on **automating the deployment** process. Like CI, it is **continuous and automated**, but the steps depend on what your software does.

What is Delivery?

- In packaged software, delivery might involve **uploading files** for users to download.
- In web-based applications, delivery involves **updating pre-production environments**, testing the application, and then pushing the changes to production.

Deployment Process:

For web applications, CD includes:

1. **Pre-production testing**: Validating changes in a non-customer-facing environment.
2. **Production deployment**: Moving validated changes to the live environment.

Example CD Pipeline in GitLab:

The **deployment** stage often comes after CI in GitLab pipelines.

```
stages:

  - deploy



deploy:

  stage: deploy

  script:

    - echo "Deploying the application..."

    - ./deploy_to_server.sh

  when: manual  # Deployment can be triggered manually
```

Fully Automated vs. Manual Release:

- **Automated CD**: Some companies release changes immediately once they are ready, but this is usually seen in **startups** or **highly automated environments**.
- **Manual Releases**: Most organisations opt for controlled releases to minimise downtime and ensure developers are available to handle any issues.

Source Control with Git and GitLab

What is Source Control?

Source control manages the **tracking of changes to files** over time. In GitLab's case, it uses **Git**, a version control system, to track these changes. Git records **file changes** rather than creating entire copies of files, making it **faster** and more efficient.

Advantages of Git:

- **Speed**: Git can switch between different code versions almost instantly.
- **Efficiency**: Tracks changes rather than full file copies.
- **Easy Integration**: Git makes it easy for multiple developers to work on the same file, often automatically merging changes.

Example Git Commands:

Basic Git commands for version control:

```
# Initialise a Git repository

git init


# Add changes to the staging area

git add .


# Commit changes

git commit -m "Initial commit"


# Push changes to remote repository

git push origin main
```

Importance of Source Control:

Source control is essential for **collaboration** and **tracking** in modern software development. Git has become the **industry standard** for version control due to its speed and flexibility.

Conclusion

GitLab offers a **complete solution** for **source control**, **CI**, and **CD**, making it a strong alternative to platforms like GitHub. Its powerful integration features and ease of use help developers **automate testing**, **validate code**, and **deploy applications** with confidence.

Whether you're managing source code or setting up a fully automated pipeline, GitLab provides the tools you need to streamline your development and deployment processes.

Understanding CICD in the Software Development Life Cycle (SDLC)

1. Problem Overview

The key question being addressed is: **What problem does CICD solve?** To answer this, it's essential to understand the broader process of software development, known as the **Software Development Life Cycle (SDLC)**. By analysing the SDLC, we can better appreciate the role of Continuous Integration and Continuous Delivery (CICD) and how it enhances software delivery.

---

2. Key Components of the SDLC

The SDLC consists of several stages, each playing a vital role in software creation:

- **Analysis**: This phase involves researching a problem or market and identifying the need for a solution.
- **Design**: After identifying the need, a conceptual design for a solution is created.
- **Development**: Here, the actual code is written, and necessary assets such as graphics are developed.
- **Deployment (Delivery)**: This stage focuses on getting the final product into the hands of users, whether through online updates or physical distribution.
- **Maintenance**: Post-deployment, the product requires ongoing support, including bug fixes and updates.

---

3. The Role of CICD in SDLC

CICD aims to streamline and enhance the **Development** and **Deployment** stages of the SDLC, but its impact extends across the entire cycle. The traditional development process involved rigid integration and delivery phases, which did not align well with the flexible and dynamic nature of the rest of the SDLC. CICD changes this by introducing adaptability.

Key Innovations of CICD:

- Allows integration and delivery to happen more frequently and fluidly, aligning with the continuous and iterative nature of software development.

- Supports dynamic back-and-forth adjustments during different stages, including **Design** and **Development**, which can blur into each other as teams iterate and improve based on feedback.

CICD ensures that each of these phases can adapt to changing requirements or constraints, promoting faster feedback loops and reducing the time to address issues or deliver new features.

---

4. SDLC Stage Flexibility

There is no fixed time for each SDLC stage—timing depends on the team and project. For example:

- **Large Companies**: May spend considerable time in the **Analysis** and **Design** phases to avoid reputational damage.
- **Solo Developers**: Might move quickly in these phases, focusing more on testing their ideas in the market.

In the **Development** stage, large teams might progress faster due to more resources, while a solo developer would handle all aspects themselves. Additionally, stages like **Analysis** and **Design** often overlap, and **Design** may evolve during **Development** to adjust to technical challenges or new opportunities.

---

5. Key Takeaways

- CICD adapts traditional, rigid integration and delivery methods into flexible, continuous processes that fit the natural variability of software development.
- It allows developers to better integrate the **Development** and **Delivery** phases within the context of the SDLC, improving overall efficiency and responsiveness to change.

Code Snippet Example for CICD Pipeline (CI/CD Pipeline)

A simple example of a **CICD Pipeline** in YAML format for a continuous integration tool like GitLab CI:

```yaml
stages:
  - build

  - test

  - deploy


build_job:

  stage: build

  script:

    - echo "Building the project..."

    - npm install


test_job:

  stage: test

  script:

    - echo "Running tests..."

    - npm test


deploy_job:

  stage: deploy

  script:

    - echo "Deploying to production..."

    - npm run deploy
```

This code demonstrates how to set up stages for **Build**, **Test**, and **Deploy** in a typical CICD pipeline.

By leveraging CICD, the overall software development process becomes more agile, and teams are better equipped to handle evolving requirements or issues that arise during the SDLC.

Setting up a GitLab Project for CICD

1. Introduction to GitLab Setup

This section explains how to set up a GitLab project for continuous integration and continuous delivery (CICD). The instructor doesn't walk through account creation on GitLab, but recommends checking out an introductory course for that if needed.

Keynote: There is some confusion when signing up for a free account on GitLab.com because the main page emphasises a free trial. To sign up for the free version, navigate to the **Pricing** section and select the free plan.

---

2. Creating a GitLab Project

Once the GitLab account is set up, follow these steps to create a project:

1. **Create a Blank Project**:
    - On your GitLab dashboard, select **Create Blank Project**.
    - Enter the project name, e.g., `gitlab-cicd`. You can also add an optional description.

```
Project Name: gitlab-cicd

Description: Example GitLab project for CICD
```

2. **Optional Configurations**:
    - Deployment Target: If you're deploying to environments like Kubernetes, you can select deployment templates. However, this is not needed for the course example.
    - Static Application Security Testing (SAST): For real-world projects, you can enable SAST to automatically scan for security vulnerabilities in the code, but it's left off for simplicity in this course.
3. **Defaults**: Accept the rest of the default settings and click **Create Project**.

---

3. Exploring the Project

After creating the project, you'll land on the project page with several key elements to note:

Main Branch: GitLab creates a default branch called `main`. If you prefer to use `master` or `trunk`, this can be changed in the project settings.

```
# To rename the branch from 'main' to 'master':

git branch -m main master

git push origin master
```

- **README File**: A README file is generated by default. It includes inline documentation on how to use Git and GitLab effectively.

---

4. Web IDE and Git Pod

GitLab provides two key tools for development directly within the platform:

- **Gitpod**: Gitpod is an environment that allows you to create a full deployment test environment where you can run your code.
- **Web IDE**: The Web IDE is a browser-based integrated development environment that enables you to edit code files directly within GitLab.

Although both tools are useful, this course does not focus on using the Web IDE.

---

5. Exploring GitLab Features

Before moving on, the instructor recommends exploring the various features available in the GitLab interface, especially those found in the left-hand menu. These include several options and functionalities beyond the scope of this course. However, the primary focus of the course will be on features under the **CICD** tab.

Example of GitLab Project Setup Command

```
# Step 1: Clone the repository

git clone https://gitlab.com/your-username/gitlab-cicd.git


# Step 2: Create a new branch

git checkout -b my-feature-branch


# Step 3: Add files and commit changes

touch example-file.txt

git add example-file.txt

git commit -m "Add example file"


# Step 4: Push the changes

git push origin my-feature-branch
```

This command block helps with cloning your project, creating branches, adding files, and pushing changes to your GitLab repository.

Key Takeaways

- **Creating Projects**: Learn how to set up a basic project with optional configurations for deployment and security testing.
- **Branch Management**: Understand the default branch creation and how to change it if needed.
- **Web IDE and Gitpod**: Know the tools available for easy code editing and deployment testing within GitLab.

Lean Manufacturing Models in Software Development

1. Introduction to Lean Manufacturing

Lean manufacturing, a methodology developed by **Toyota in the 1930s**, revolutionised production by focusing on efficiency and reducing waste. The core idea, **Just-in-Time** (JIT), involves producing goods in small batches, purchasing materials as needed, and minimising finished product inventory.

In software development, these concepts translate well. For example:

- **Requirements gathering and design** are like the "raw materials" of software.
- **Batch sizes** represent the set of changes released together.
- The **finished product inventory** is similar to the amount of completed but unreleased code.

---

2. Just-in-Time and Software Development

The **Just-in-Time (JIT)** principle focuses on making products in small batches, purchasing resources as needed, and maintaining low inventory. This approach can be applied to software development by focusing on smaller, frequent releases rather than large, infrequent updates.

- **Requirements and design**: These represent the raw materials of software.
- **Release batches**: The changes to be deployed together can be seen as the batch size in manufacturing.
- **Finished product inventory**: In software, this refers to the code that's complete but not yet deployed.

---

3. Theory of Constraints in Software Development

The **Theory of Constraints** highlights how limiting factors in a process can slow down the entire workflow. By identifying and resolving each constraint, the overall efficiency improves until the next constraint is encountered.

Example:

- If a company has **10 product designers** but only **1 software engineer**, the constraint lies with the engineer's ability to develop code.
- Once additional engineers are hired, the next bottleneck might be **QA testing**, requiring the company to invest in **QA automation**.

This approach is iterative, and each constraint is addressed step by step to optimise the process.

---

4. Increasing Cadence for Better Quality

In software development, maintaining a **faster release cadence** (releasing small changes frequently) offers several advantages over a **slow cadence** (releasing large changes less frequently).

Slow Cadence: With infrequent updates, a large number of changes (high delta) are introduced all at once. This increases the chances of critical bugs or failures.

```
Large change (high delta) → More places to break → Higher risk of major bugs
```

Faster Cadence: More frequent releases with smaller changes (low delta) reduce the likelihood of major disruptions. Bugs that are introduced are typically smaller, easier to fix, and less disruptive.

```
Small change (low delta) → Fewer places to break → Easier to identify and fix bugs
```

---

5. Benefits of Faster Release Cadence

The two major benefits of increasing release cadence in software development are:

Faster Feature Delivery: Users receive new features and updates more quickly, allowing them to benefit from improvements sooner.

```yaml
# Sample YAML CICD pipeline to ensure frequent and small deployments

stages:

  - build

  - test

  - deploy
```

```
deploy:

  stage: deploy

  script:

    - echo "Deploying small incremental changes"

    - npm run deploy-small
```

1. **Improved Stability**: With smaller releases, the number of changes per update (delta) is smaller, which means:
    - Fewer large-scale bugs are introduced.
    - Bugs are easier to track and fix.
    - Minor bugs can be quickly patched in the next release rather than waiting for a major update or requiring a hotfix.

---

6. Counterintuitive Stability

Although it might seem that more frequent releases would introduce more bugs, the **smaller scale of changes** in each release typically leads to a more stable application overall. This is because:

- The number and scope of bugs are reduced.
- Smaller updates make it easier to isolate and resolve issues.

Additionally, **minor bugs** are less disruptive since they can often be fixed in the next scheduled release instead of requiring an urgent patch.

Key Takeaways

- **Lean Manufacturing Models** like **Just-in-Time** and the **Theory of Constraints** translate effectively to software development.
- **Faster release cadence** reduces the risk of major bugs, improves stability, and allows quicker delivery of new features to users.

Notes on Continuous Integration (CI)

1. Introduction to CI

- **Continuous Integration (CI)** refers to the practice of continuously integrating changes into the codebase through an automated system.
- The core idea of CI is that changes are merged and tested automatically and immediately after being completed, eliminating the need for periodic manual integration events.

2. CI vs. Non-CI Workflows

- In **CI**, code changes are merged immediately after they're complete, which allows for continuous testing.
- The alternative to CI involves **scheduled merges** at specific times (e.g., end of a sprint), leading to a more complex merging process.
- In **non-CI workflows**, QA (Quality Assurance) is performed after all changes are merged, making it harder to track down bugs because many changes are bundled together.

3. Testing in CI

- CI relies heavily on **automated tests** to maintain code quality. This ensures that:
  - Bugs are easy to track down since changes are tested immediately.
  - The automated tests must pass before the code can be merged.

4. CI Process Flexibility

- **CI** workflows are not deadline-driven; merges happen when work is ready, leading to minor fixes closer to deadlines, rather than major changes.
- In contrast, **non-CI workflows** require code to be ready by specific dates, which can create last-minute crisis scenarios.

5. Automated Testing as Sifting Layers

- Testing in CI is likened to **sifting through layers of dirt** on an archeological dig, where each layer catches smaller and smaller artifacts.
  - **Big issues** (like syntax errors) are caught early by basic tests.
  - **Fine issues** (like integration bugs) are caught by more detailed tests later in the process.

6. Types of Tests in CI

- **Syntax Testing and Linting:**
  - Syntax testing ensures that the code is valid and error-free.
  - Linting ensures adherence to coding style standards.
- **Unit Testing:**

    - Focuses on testing individual functions or units of code with valid and invalid inputs.
- **Integration Testing:**
    - Ensures that different components of the system work well together (e.g., API calls).
- **Acceptance Testing:**
    - Simulates real-world user behaviour in an environment that mimics production as closely as possible.

```
# Example of Unit Test in Python using unittest framework

import unittest


def add_numbers(a, b):

    return a + b


class TestMathOperations(unittest.TestCase):

    def test_add_numbers(self):

        self.assertEqual(add_numbers(2, 3), 5)

        self.assertEqual(add_numbers(-1, 1), 0)

        self.assertEqual(add_numbers(-1, -1), -2)

    def test_invalid_input(self):

        with self.assertRaises(TypeError):

            add_numbers(2, "three")

if __name__ == '__main__':

    unittest.main()
```

7. Early Failures and Comprehensive Tests

- Tests should be designed to **fail early and often**:
    - **Fail early:** Tests should quit as soon as an error is found.
    - **Fail often:** Frequent test runs catch bugs early rather than relying on manual code review before merging.

8. Benefits of a Strong CI System

- A well-designed CI system with comprehensive tests increases confidence in the codebase, speeds up development, and reduces the anxiety that a change might break something critical.

9. Conclusion

- Implementing CI effectively can reduce merge crises, track down bugs faster, and improve the overall development experience, making the process smoother and more fun for developers.

Creating a Pipeline with GitLab CI/CD

1. Introduction to GitLab CI/CD Pipelines

- **GitLab CI/CD** makes setting up a Continuous Integration/Continuous Deployment (CI/CD) pipeline streamlined and straightforward.
- The pipeline configuration is stored in a **YAML file** which is highly readable and intuitive, even for users who aren't deeply familiar with the syntax.

2. Initial Setup

- To start, navigate to the **CI/CD menu** in GitLab.
- GitLab will prompt you to select a template. For basic pipeline setups, the **Bash template** is a good starting point.
    - The template is a **YAML file** that defines the pipeline's stages and jobs.

3. Understanding the Pipeline YAML File

- The pipeline configuration is structured into different sections:
    - **Build**
    - **Test**
    - **Deploy**
- Each section represents a stage in the pipeline and is fairly self-explanatory.

```
# Example of a basic pipeline YAML
```

```yaml
image: busybox:latest   # Docker image used for running scripts

before_script:

  - echo "Setting up environment"

build:

  stage: build

  script:

    - echo "Building the project"

test1:

  stage: test

  script:

    - echo "Running test 1"

test2:

  stage: test

  script:

    - echo "Running test 2"

deploy:

  stage: deploy

  script:

    - echo "Deploying application"
```

4. Key Sections in the Pipeline YAML

- **Image:**
    - The **image** section in the YAML specifies the Docker container image that will be used for running the pipeline's jobs.

    - In this example, GitLab uses the `busybox:latest` image, which is a minimal container ideal for running basic scripts.
- **Before Script and After Script:**
    - **Before script:** Defines tasks that will run before every job in the pipeline. This is useful for setting up the environment.
    - **After script:** Executes after each job, typically used for **cleanup tasks** or **artefact handoff** between stages.
- **Stages and Jobs:**
    - Each job in the pipeline is associated with a **stage**.
    - The stages in the template are `build`, `test`, and `deploy`.
    - Jobs within the same stage, such as `test1` and `test2`, will run **in parallel**.

```
# Example: Parallel Test Jobs in a GitLab Pipeline

test1:

  stage: test

  script:

    - echo "Running test 1"

test2:

  stage: test

  script:

    - echo "Running test 2"
```

5. Customization and Flexibility

- Although the template provided is basic, you can easily customise it:
    - You can use **custom Docker containers** or **omit Docker altogether**.
    - Instead of Docker, you can deploy directly to cloud infrastructure like **AWS EC2** or **Google Cloud**.

6. Committing the Pipeline

- After reviewing or modifying the pipeline YAML, you can commit it to your repository's main branch.

- GitLab automatically triggers the pipeline once the file is committed, and you'll see the **pipeline status** updating on GitLab.

7. Pipeline Execution

- After committing the file, GitLab will run the newly created pipeline and display its status.
- The jobs within the pipeline will be executed in their respective stages, with tests running in parallel where defined.

```
# Example of console output in GitLab pipeline

$ echo "Building the project"

Building the project



$ echo "Running test 1"

Running test 1



$ echo "Running test 2"

Running test 2



$ echo "Deploying application"

Deploying application
```

8. Conclusion

- GitLab provides an easy-to-use, intuitive environment for setting up CI/CD pipelines. The YAML-based pipeline configuration allows developers to define stages such as **build**, **test**, and **deploy**, with scripts running at each stage. You can further customise the pipeline by adding your own Docker containers, setting up environment configurations, or even skipping Docker entirely for direct deployments to cloud services.

Pipeline Overview and Execution

Viewing the Pipeline

- You can view your pipeline run even while it's in progress.
- To access the pipeline:
    1. Navigate from the commit or click on "Pipelines" in the left-hand menu.
    2. Click on the number below the pipeline title (do not click the title, as it will take you to the commit).
    3. This brings you to the pipeline run page.

Pipeline Stages Overview

- The items displayed in the pipeline view correspond to the different sections in the configuration file.
- For example, clicking on `build1` will display the console output:
    - **Docker** triggers the download and runs the `busybox latest` image.
    - The repository code is checked out.
    - The "before scripts" are executed to prepare the build.
    - **Build Output**: Simply echoes the string `echo "your build here"`.
    - The "after scripts" handle cleanup.

Code Snippet (Build Example)

```
before_script:

  - echo "Preparing build..."

script:

  - echo "your build here"

after_script:

  - echo "Cleaning up..."
```

Rerunning Pipeline Sub-stages

- Each step in the pipeline, such as `build1`, has a **retry** symbol allowing you to rerun a specific sub-stage.
  - This is useful for dealing with flaky tests or runtime issues without rerunning the entire pipeline.
  - Example: If your test fails due to a runtime issue, you can simply retry the build stage.

Parallel Test Jobs

- In the **test stage**, multiple jobs run in parallel.
- You can click into each job to view its console output, which looks similar to the build stage.
- Example:
  - A test suite is executed, followed by cleanup.

Code Snippet (Test Example)

```
test:

  parallel: 2

  script:

    - echo "Running test suite..."

    - echo "Cleaning up after test..."

test:

  parallel: 2

  script:

    - echo "Running test suite..."

    - echo "Cleaning up after test..."
```

Monitoring the Pipeline in Real Time

- By clicking the retry button on any sub-stage (e.g., `test1`), you can monitor the job in real time.

    - This feature is particularly useful for long-running tests.
    - Example: If a test is expected to take a significant amount of time, real-time monitoring allows you to determine whether it’s progressing as expected.

Code Snippet (Retry Logic)

```
retry:

  max: 3

  when: on_failure
```

Conclusion

- The pipeline interface allows you to retry specific jobs, monitor jobs in real time, and efficiently manage various stages such as build and test with detailed logs. This flexibility helps in dealing with runtime issues and managing parallel job executions efficiently.

Advanced Pipeline Concepts in GitLab CI/CD

Stage Independence and Interaction

- In the current pipeline example, stages like **build**, **test**, and **deploy** run independently without interaction.
- In some cases, you might want to pass an artefact from one stage to the next (e.g., build → test).

Defining Stage Order

- By default, stages run in the order **build**, **test**, and **deploy**, but you can make this explicit by defining them in your YAML file.

Code Snippet (Defining Stages)

```
stages:
  - build

  - test

  - deploy
```

- This explicitly defines the stages of the pipeline, but you can use any stage names that suit your project needs.

Creating and Passing Artefacts Between Stages

Creating an Artefact in Build Stage

- In the **build** stage, you can generate an artefact to pass to the **test** stage.
- The following example creates a text file `file.text` as an artefact.

Code Snippet (Artefact Creation)

```
build1:

  stage: build

  script:

    - echo "This is a file" | tee file.text

  artifacts:

    paths:

      - file.text
```

- The `tee` command outputs to both the console and a file (`file.text`).
- The `artefacts` section specifies that `file.text` will be stored as an artefact for future stages.

Consuming Artefacts in the Test Stage

- To consume the artefact created in the **build** stage in the **test** stage, you need to define **dependencies**.

Code Snippet (Artefact Consumption)

```
test1:

  stage: test

  dependencies:

    - build1

  script:

    - cat file.text
```

- The **dependencies** array specifies that **test1** depends on the **build1** job, ensuring the artefact (`file.text`) is available in the test stage.
- The `cat file.text` command checks if the file exists and prints its content.

Using the GitLab Editor

- GitLab offers several options for editing the pipeline:
  - **Built-in YAML editor**: Includes features like linting (syntax checking) and pipeline visualisation.
  - **Web IDE or command-line editors**: Can be used if preferred.
- **Editor features**:
  - **Visualise Button**: Provides a visual representation of the pipeline.
  - **Lint Button**: Validates YAML syntax.

Committing Changes and Triggering the Pipeline

- Once you've edited the pipeline file, committing it will trigger the pipeline run.
- GitLab will automatically execute the jobs in the defined stages.

Best Practices for Pipelines

- Start with a simple pipeline and gradually add complexity as needed.
- Avoid over-complicating the pipeline with too many stages or complex dependencies.

    - This can lead to maintenance challenges and longer pipeline runtimes.

Advanced Features of GitLab Pipelines

- **Directed Acyclic Graphs (DAG)**: Allow for more complex dependencies and parallelization.
- **Sub-pipelines**: Enable breaking down large pipelines into smaller, more manageable parts.

Conclusion

- In this example, we've covered the basics of passing artefacts between stages, explicitly defining stage order, and using the built-in GitLab editor for YAML file management. Starting with a simple pipeline is crucial for maintainability, only adding complexity when necessary.

Adding a Test to the Pipeline

Overview of Test Output

- In the **test1** job, the content of the file from the build stage was passed and displayed.
- However, the current test merely prints "This is a file", which doesn't offer pass/fail functionality.

Adding a Test with `grep`

- To add a functional test, we use the `grep` command in **test1** to check if a specific string is present in the file.

Code Snippet (Adding a Test)

```
test1:

  stage: test

  script:

    - echo "Testing for string GitLab"
    - grep "GitLab" file.text
```

- The `grep` command searches for the string **GitLab** in the `file.text`. If the string is not found, the test will fail.

Initial Pipeline Execution

- Running the pipeline will trigger the build and test stages.
- Since **GitLab** is not in the file, the **test1** job will fail.

Using Variables in GitLab CI

- To ensure the test passes, we can introduce a **variable** in the build stage.
- This variable will hold the string **GitLab**, which will be included in the file.

Code Snippet (Using Variables)

```
variables:

  MY_WORD: "Something"

build1:

  stage: build

  script:

    - echo "This is a file $MY_WORD" | tee file.text
```

- The variable `MY_WORD` is defined with a default value of "Something".
- The build job writes `This is a file $MY_WORD` to the `file.text` file, resulting in "This is a file Something".

Running the Pipeline with Updated Variable

- After running the pipeline, the **test1** job will still fail since **GitLab** is not in the file.
- To pass the test, manually rerun the pipeline and set the `MY_WORD` variable to **GitLab**.

Code Snippet (Running the Pipeline Manually)

```
# In the pipeline run settings

MY_WORD: "GitLab"
```

- Manually overriding the `MY_WORD` value to "GitLab" allows the test to pass, as `grep` will find the string.

Test Success and Continuous Integration

- After setting `MY_WORD` to **GitLab**, the pipeline passes successfully.
- Ideally, tests should be automated and configured to pass without manual intervention, ensuring smooth continuous integration (CI).

Key CI Concept

- Continuous integration involves running tests automatically with every commit to the codebase.
- Effective CI balances between test coverage and speed, ensuring that tests catch issues without slowing down development.

Conclusion

- The test now checks for the presence of the string **GitLab** in the file.
- Using variables enhances flexibility, allowing parameterized values to be tested.
- Continuous integration ensures frequent and automatic testing, promoting stable and reliable code.

Generating a Website with CI/CD Pipelines

Step 1: Create a Markdown File

- Start by creating a markdown file named **website.md**.

Add basic content to the file:

```
Welcome to my website.
```

```
This is the main content.
```

- Commit the changes.

Step 2: Modify the Pipeline Configuration

- In the CI/CD pipeline editor, update the image to use **alpine**
  instead of **busybox**
  . Alpine is a lightweight Linux distribution that supports more tools.

Code Snippet (Updating the Image)

```
image: alpine:latest
```

- Alpine will be used to install a markdown renderer to convert the markdown file into HTML.

Step 3: Render the Website

- Update the **build1** job to **render_website**.
- Remove unnecessary scripts and add the following commands to render the markdown file into HTML.

Code Snippet (Render Markdown to HTML)

```
render_website:

  stage: build

  script:

    - apk add markdown   # Install markdown renderer

    - markdown website.md | tee index.html   # Convert markdown to HTML

  artifacts:
    paths:

      - index.html
```

- This script instals the markdown renderer using Alpine's package manager, converts `website.md` to `index.html`, and saves the HTML file for later use.

Step 4: Test the HTML File

- Update the **test** job to **test_website**.
- Install **xmllint** to check the validity of the generated HTML file.

Code Snippet (Testing HTML)

```
test_website:

  stage: test

  script:

    - apk add libxml2-utils   # Install xmllint for testing

    - xmllint --html index.html   # Validate HTML file

  dependencies:

    - render_website
```

- **xmllint** verifies the HTML file, ensuring it is valid.

Step 5: Deploy Preparation

- Rename the deploy job to **deploy_website**, leaving the content unchanged for now. It will be updated in the next stage of the pipeline.

Code Snippet (Deploy Job Setup)

```
deploy_website:

  stage: deploy

  script:

    # Deploy logic will be added in the next chapter
```

Step 6: Commit and Run the Pipeline

- After making all the changes, commit the pipeline configuration. This will trigger the pipeline.
- The **render_website** job runs first, generating the HTML file.
- The **test_website** job runs next, validating the HTML.

Result:

- The pipeline successfully converts the markdown file to HTML, validates it, and prepares for deployment.

Conclusion

In this example, we set up a simple CI/CD pipeline to:

1. Render a markdown file into HTML using Alpine Linux.
2. Validate the generated HTML.
3. Prepare for deploying the website.

Next, you can set up the pipeline to deploy the generated HTML as a static website.

Continuous Delivery (CD) Concepts

Continuous Delivery (CD) is an extension of Continuous Integration (CI) but focuses on automating the deployment of applications. Below are the key concepts and stages involved in the CD.

What is Continuous Delivery (CD)?

- CD involves automating the deployment of code to different environments in a systematic and reliable way.
- The goal is to have code automatically deployed to production after passing necessary tests, reducing human intervention.

Environments in CD

CD operates across various **environments**, which are independent setups capable of running the entire application.

1. Development (Dev) Environment

- Typically a developer's laptop, a Vagrant box, or a cloud instance.
- Each developer should have their own environment to avoid conflicts.
- Designed to be easily reset if an issue arises (e.g., SQL query errors).
- Might have pared-down settings since it only supports one user.

2. Quality Assurance (QA) Environment

- A shared environment where **automated and manual tests** are performed.
- Should mimic the production environment but can have dummy data.
- The code in QA is frequently updated and might be buggy or incomplete.

3. Staging Environment

- A **clone of production** with similar configurations and, ideally, identical data to production (within data security constraints).
- Staging is used to test releases before pushing them to production.
- Code in staging is only updated when a release is ready to be tested for production deployment.

4. Production (Prod) Environment

- This is the live environment where the application is accessed by customers.
- Production needs to be a part of the CD pipeline to ensure features are deployed smoothly.
- Should include **automated tests** (sanity checks) to catch issues after deployment.

Key Benefits of CD:

- Continuous testing and releasing ensure a smooth transition of code from development to production.
- Allows for **automatic rollback** if an issue arises in production.

CI/CD Pipelines

- CD pipelines function similarly to CI pipelines. Jobs are executed sequentially in a pipeline, and processes can be halted if a failure occurs to avoid running unnecessary jobs.

Pipeline Configuration

GitLab's CD features allow for different pipelines for each environment.

Example Pipeline Structure:

- **Development Branch**: Where feature branches are developed and tested.
- **QA Branch**: Where tested features from the development branch are merged.
- **Release Branch**: After QA testing, a release is created, tagged, and deployed to staging.
- **Production**: After successful staging tests, the code is pushed to production.

Code Promotion Strategy

- **Feature branches** are first developed and tested in QA.
- Once confirmed, the code is merged into the long-lived **development branch**.
- When ready for release, it is pushed to **staging** and tagged with a version.
- Finally, identical code is pushed to **production**.

Git Flow in CD

One approach for managing environments and code promotion is **Git Flow**, where:

1. Features are developed in feature branches.
2. Merged into the development branch for QA testing.
3. A release is "cut" from the development branch and pushed to staging for final testing.
4. The tested code is then deployed to production.

Automated Testing in Production

- Post-deployment, production should run **sanity checks** to ensure everything is functioning as expected.

Sample GitLab CI Pipeline Configuration for Environments:

```
stages:

  - build

  - test

  - deploy

build_job:

  stage: build

  script:

    - echo "Building application"

test_job:

  stage: test

  script:

    - echo "Running tests"

  dependencies:

    - build_job

deploy_job:

  stage: deploy

  script:

    - echo "Deploying to production"

  when: manual   # Allows for manual trigger after tests
```

Conclusion

CD ensures a streamlined process from development to production with minimal manual intervention. The use of different environments (Dev, QA, Staging, and Prod) ensures that code is thoroughly tested and deployed with confidence.

Setting Up Environments for Static Website Deployment

In this chapter, we will set up multiple environments where we can deploy our static website using GitLab and AWS S3. The environments will include **QA**, **Staging**, and **Production**.

Step 1: Create a New User in AWS (IAM)

1. Go to the **IAM** (Identity and Access Management) service in AWS.
2. Add a new user named `gitlab_cicd` with **Programmatic access** to generate an access key and secret key.
3. Download the credentials (Access Key and Secret Key) as a CSV file. Save this file securely.
4. Copy the **User ARN** for future use and save it in your notes.

Code Example: Creating IAM User

```
aws iam create-user --user-name gitlab_cicd

aws iam create-access-key --user-name gitlab_cicd
```

Step 2: Set Up an S3 Bucket for QA

1. Navigate to the **S3** service in AWS.
2. Click **Create bucket** and provide a **unique bucket name** like `yourname-gitLab-qa`.
3. Select your region (e.g., `US West 2`) and **uncheck "Block all public access"** to allow public access to the website.
4. Acknowledge that you are allowing public access and click **Create bucket**.

Bucket Naming Example:

- `samuelson-gitlab-qa`

Step 3: Configure S3 Bucket Permissions

1. Open the created S3 bucket and go to the **Permissions** tab.
2. Under **Bucket Policy**, click **Edit** and use the **Policy Generator** to generate permissions.

Add Public Access Policy:

- **Principal**: `*` (allows anyone to access the bucket)
- **Service**: `S3`
- **Actions**: `GetObject`
- **Resource**: The bucket ARN with `/*` to allow access to all files in the bucket.

Add User Access Policy:

- Use the **User ARN** copied earlier as the principal.
- **Actions**: `PutObject` to allow the user to upload files to the bucket.

Sample Bucket Policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::yourname-gitLab-qa/*"
    },
    {
      "Effect": "Allow",
      "Principal": "arn:aws:iam::your-account-id:user/gitlab_cicd",

      "Action": "s3:PutObject",

      "Resource": "arn:aws:s3:::yourname-gitLab-qa/*"

    }

  ]

}
```

3. Click **Save Changes**.

Step 4: Enable Static Website Hosting

1. In the **Properties** tab of the bucket, scroll down to **Static website hosting** and click **Edit**.
2. Enable static website hosting and set the **Index document** as `index.html`.
3. Copy the **Bucket website URL** for later use.

Static Website Settings:

- **Index Document**: `index.html`

Step 5: Set Up Environments in GitLab

1. In your **GitLab** project, go to the **Deployments** section and select **Environments**.
2. Click **New environment**, name it `qa`, and paste the **S3 bucket website URL** copied earlier.
3. Repeat this process for **Staging** and **Production** by creating additional S3 buckets and copying their configuration from the QA bucket.

Example:

```
stages:

  - deploy
```

```
deploy_to_qa:

  stage: deploy

  script:

    - aws s3 cp index.html s3://yourname-gitLab-qa/ --acl public-read

  environment:

    name: qa

    url: http://yourname-gitLab-qa.s3-website-us-west-2.amazonaws.com
```

Step 6: Upload the Static Website

Once you have set up the environments, upload the `index.html` file to the S3 bucket. This will turn the bucket into a functioning website.

Upload Command:

```
aws s3 cp index.html s3://yourname-gitLab-qa/ --acl public-read
```

You can repeat these steps for setting up **Staging** and **Production** environments by creating separate S3 buckets for each.

This completes the setup of multiple environments for deploying a static website using GitLab and AWS S3.

Setting Up Environment Variables and Pipeline Configuration for CI Deployment

This lesson follows an example from GitLab's blog post on **CI deployment and environments**. The main goal is to set up environment variables for CI jobs, configure the pipeline, and deploy the website to an AWS S3 bucket.

Step 1: Setting Up Environment Variables

1. Navigate to **Settings > CI/CD** in your GitLab project.
2. Under the **Variables** section, click **Expand**.

Add the following environment variables:

1. AWS_ACCESS_KEY_ID

- Key: `AWS_ACCESS_KEY_ID`
- Value: Your AWS access key ID (from the CSV file).
- Leave the **Protected** option enabled.
- No need to mask this variable as it's safe to appear in logs.

2. AWS_SECRET_ACCESS_KEY

- Key: `AWS_SECRET_ACCESS_KEY`
- Value: Your AWS secret access key.
- Enable the **Masked** option to hide the value in logs, as this is a sensitive key.

3. AWS_DEFAULT_REGION

- Key: `AWS_DEFAULT_REGION`
- Value: Your AWS region (e.g., `us-west-2`).
- This helps avoid errors when certain AWS operations require a region setting.

Code Snippet for GitLab CI/CD Environment Variables

```
variables:

  AWS_ACCESS_KEY_ID: your-access-key-id

  AWS_SECRET_ACCESS_KEY: your-secret-access-key

  AWS_DEFAULT_REGION: us-west-2
```

Step 2: Updating the GitLab Pipeline Configuration

1. Go to **CI/CD > Editor** in your GitLab project.
2. Modify your pipeline configuration (`.gitlab-ci.yml`) to set up the **QA** environment and define the **S3 bucket** where the website will be deployed.

Adding S3 Bucket to Pipeline

- Define a variable for the S3 bucket that will store the deployed website files:

```
variables:

  S3_BUCKET: Samuelson-GitLab-QA
```

Step 3: Organising Artefacts

- Modify the pipeline job to place website artefacts in a **public** subdirectory to keep them separate from the source code.

Code for Artefact Directory and Paths

```
render_website:

  script:

    - mkdir -p public

    - some-command-to-generate-website -o public/index.html

  artifacts:

    paths:

      - public/index.html
```

Step 4: Installing AWS CLI in the Deploy Job

1. Install the **AWS CLI** tool in the deploy job using **Alpine Linux**.

Install AWS CLI:

```
deploy_to_qa:

  script:

    - apk add --no-cache aws-cli
```

2. Use the AWS CLI to upload the files in the `public` directory to the S3 bucket.

Uploading Files to S3 Using AWS CLI

```
 script:

    - aws s3 cp ./public/ s3://$S3_BUCKET/ --recursive
```

- The `--recursive` flag ensures that all files in the `public` directory are copied to the S3 bucket.

Step 5: Defining Dependencies in Deploy Job

- Since the **deploy** job depends on the artifacts generated in the **render_website** job, specify the dependency to ensure correct execution order.

Adding Dependencies

```
deploy_to_qa:

  dependencies:

    - render_website
```

Final GitLab CI/CD Pipeline Configuration

```
stages:
  - build

  - deploy

render_website:

  stage: build

  script:

    - mkdir -p public

    - some-command-to-generate-website -o public/index.html

  artifacts:

    paths:

      - public/index.html

deploy_to_qa:

  stage: deploy

  script:

    - apk add --no-cache aws-cli

    - aws s3 cp ./public/ s3://$S3_BUCKET/ --recursive

  dependencies:

    - render_website
```

Step 6: Committing and Testing the Pipeline

1. Once you have made these changes, **commit** them in the GitLab editor.
2. The pipeline will run, and if successful, you can visit the **QA environment** by navigating to the S3 bucket URL to view your deployed website.

By following these steps, you have set up environment variables and modified the GitLab CI/CD pipeline for deploying your website to an AWS S3 bucket.

First Deployment Pipeline Setup and Enhancements

In this guide, we'll walk through setting up your first deployment pipeline using GitLab, with the focus on deploying to multiple environments (QA, Staging, and Production). We will also add environment-specific configurations and manual triggers to control deployment.

Step 1: Successful Pipeline Run

Once the first deployment pipeline has been successfully executed, we can verify it:

1. Go to the View **pipeline**.
2. Check the **deploy_website** job status.
3. Navigate to **Deployments > Environments**, and click on **qa** to view the website.

Step 2: Making the Pipeline Environment-Aware

1. Open the **CI/CD Editor**.
2. Scroll to the deploy job, copy it, and rename it from `deploy_website` to `deploy_qa`.
3. Add an environment tag to indicate the environment being deployed to.

Example: QA Deployment Job

```yaml
deploy_qa:

  stage: deploy

  script:

    - apk add --no-cache aws-cli

    - aws s3 cp ./public/ s3://$S3_BUCKET/ --recursive

  environment:

    name: qa
```

4. Commit the changes.

After running this, you will see that the deployment to **QA** is now listed with a handy link to the environment page.

Step 3: Setting Up Staging and Production Environments

Now we will set up jobs for **staging** and **production** environments using similar steps.

1. **Manual Trigger for Deployments**: Add a `rules` section to control when the deployment should be triggered.
   - Use the `when: manual` rule to ensure the pipeline only runs when manually triggered.

Adding Manual Trigger to Deployment Jobs

```
deploy_qa:

  stage: deploy

  script:

    - apk add --no-cache aws-cli

    - aws s3 cp ./public/ s3://$S3_BUCKET/ --recursive

  environment:

    name: qa

  rules:

    - when: manual
```

2. Copy the **QA** job and create jobs for **staging** and **production**.
   - Modify the environment name and S3 bucket for each job.

Example: Staging and Production Deployment Jobs

```
deploy_staging:

  stage: deploy
```

```yaml
  script:

    - apk add --no-cache aws-cli

    - aws s3 cp ./public/ s3://$S3_BUCKET/ --recursive

  environment:

    name: staging

  rules:

    - when: manual

  variables:

    S3_BUCKET: your-name-gitlab-staging

deploy_prod:

  stage: deploy

  script:

    - apk add --no-cache aws-cli

    - aws s3 cp ./public/ s3://$S3_BUCKET/ --recursive

  environment:

    name: prod

  rules:

    - when: manual

  variables:

    S3_BUCKET: your-name-gitlab-prod
```

3. Commit the changes and wait for the pipeline to run.

Step 4: Managing and Triggering Deployments

Once the jobs are defined, you'll notice the following improvements:

1. The pipeline now shows three separate deployment jobs: **deploy_qa**, **deploy_staging**, and **deploy_prod**.
2. Each job is manually triggered, meaning they won't run automatically with every commit.

To manually trigger the deployment:

- Click the **Play** button next to the deployment job in the pipeline UI.

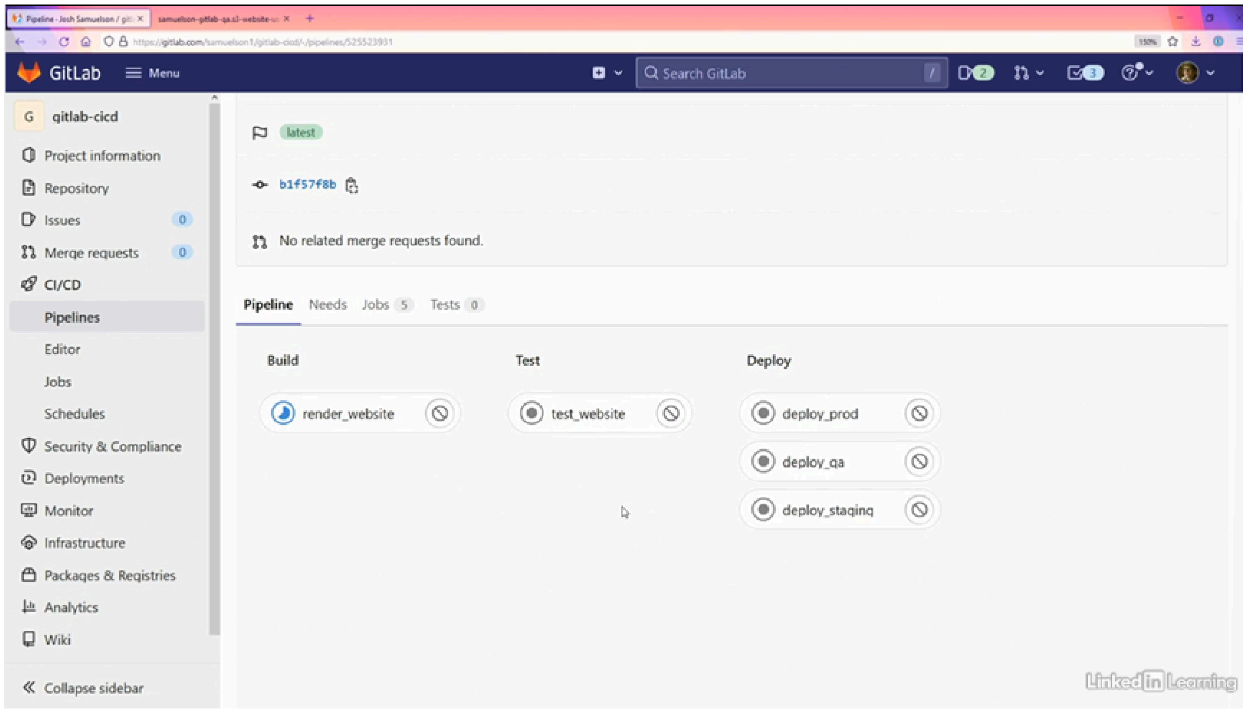
Step 5: Verifying Successful Deployments

After manually triggering the deployment jobs, you can verify the deployments:

1. Check the job status to ensure each job has completed successfully.
2. Visit the **Environments** page in GitLab to view the deployments.
    - You'll see separate entries for **QA**, **Staging**, and **Production**.

Each environment will now host the same website content, but you can independently control when and where to deploy.

This setup allows flexibility in managing deployments to different environments while ensuring that deployment to sensitive environments like **Production** is manual and deliberate.



Automating Deployments in a CI/CD Pipeline

In this section, we will automate deployments in our CI/CD pipeline using GitLab. The idea is to automatically deploy changes to specific environments based on the branch being worked on. We will also discuss how to use GitLab's built-in variables and branching strategies to control the flow of our deployments.

Step 1: Branching Strategy for Automation

We will consider the `main` branch as the main development branch, where all commits trigger an automatic deployment to QA. This will help automate the QA environment without manual intervention.

Automating QA Deployment for Main Branch

1. Open the **CI/CD Pipeline Editor** in GitLab.

In the `deploy_qa` job, replace the manual trigger rule with an automated rule based on the branch:

```
deploy_qa:
  stage: deploy
  script:
    - apk add --no-cache aws-cli
    - aws s3 cp ./public/ s3://$S3_BUCKET/ --recursive
  environment:
    name: qa
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
```

This ensures that any commit to the `main` branch automatically triggers the QA deployment.

Step 2: Testing the Automation

To test this automated deployment:

1. **Create a new branch** (e.g., `website_update`) and commit changes to it.

Navigate to the root of the project, click on the plus sign next to `main`, and create a new branch:

```
git checkout -b website_update
```

2. Edit a file, such as `website.md`, and commit the changes. This branch will not trigger a QA deployment, as it is not the `main` branch.
3. Verify the pipeline:
    - Go to **Pipelines** and observe that there is no QA deployment for this branch. The deployment jobs (Prod and Staging) are still manual.

Step 3: Merging Feature Branch to Main

1. Create a **merge request** to merge the feature branch (`website_update`) into `main`.
    - GitLab will automatically suggest creating a merge request if changes were recently committed.
    - Upon merging, a new commit is generated in the `main` branch, which will trigger the QA deployment as defined in the rules.
2. After the merge, check the pipeline again:
    - You will see the **QA job** automatically reappear and execute.

Example: Merge Request

```
# Merge feature branch into main

git merge website_update
```

Step 4: Using Rules for Advanced Branch Control

You can further customise the behaviour of jobs based on branch names. For example, you can run different test suites or deployment jobs based on naming patterns of branches using regular expressions.

Example: Running Tests Based on Branch Names

To limit tests to run only on branches that start with "feature", you can use the following rule in the test job:

```
test:

  stage: test

  script:

    - run-tests.sh

  rules:

    - if: $CI_COMMIT_BRANCH =~ /^feature/
```

This ensures that test jobs only run for branches starting with "feature."

By setting up these automation rules, we can reduce the need for manual deployment, streamline the development process, and ensure that the correct environments are updated based on branch activity.

Complete CI/CD Pipeline for Staging and Production

In this section, we will enhance our CI/CD pipeline to include staging and production environments. We will utilise Git tags to trigger these deployments and ensure that the same commit tested in QA gets promoted to staging and production. This setup avoids creating additional branches and simplifies the deployment process.

Step 1: Using Git Tags for Deployments

Instead of creating a separate branch for releases, we will use **Git tags**. A tag is a static reference to a specific commit in Git history, which allows us to deploy the same code that was tested in QA without making new changes.

1. **Creating a Tag**:

To create a new tag:

```
git tag 1.0
```

Push the tag to the repository:

```
git push origin 1.0
```

2. **Pipeline Trigger by Tag**:
   - A pipeline will automatically trigger when a tag is created, running the same commit that was previously tested in QA.

Step 2: Modifying the Pipeline for Staging and Production

To ensure that staging and production deployments are triggered only when a tag is created, we modify the deployment rules.

1. **Edit Staging Deployment**:
   - Replace the existing rule with a condition to trigger only on a tag:

```
deploy_staging:

  stage: deploy

  script:

    - apk add --no-cache aws-cli

    - aws s3 cp ./public/ s3://$S3_BUCKET/ --recursive

  environment:

    name: staging

  rules:

    - if: $CI_COMMIT_TAG
```

2. **Edit Production Deployment**:
   - The production deployment should be manual but triggered when a tag is created. To achieve this, we combine the rules:

```
deploy_prod:

  stage: deploy

  script:

    - apk add --no-cache aws-cli

    - aws s3 cp ./public/ s3://$S3_BUCKET/ --recursive

  environment:
    name: prod

  rules:

    - if: $CI_COMMIT_TAG

      when: manual
```

Step 3: Deploying with a New Tag

Once the pipeline is updated, you can deploy the code to staging and production by creating a new tag.

1. **Create a New Tag** (e.g., `1.1`):
   - This will trigger the pipeline but not run the QA deployment, as the code has already been tested in QA.

```
git tag 1.1

git push origin 1.1
```

2. **View the Pipeline**:
   - The pipeline will automatically deploy to staging, while production will require manual intervention.
   - You can manually deploy to production by clicking the **manual deploy button** in the GitLab pipeline UI.

Step 4: Handling Protected Branches

If you are using **protected branches** and encounter issues with environment variables not being available for tags, you need to adjust your pipeline settings.

1. **Unprotect Variables**:
   - Go to **Settings > CI/CD > Variables**.
   - Uncheck the **Protect variable** option to make the variables available for tag-based deployments.

Step 5: Retry and Deploy

If a staging deployment fails due to missing environment variables, you can retry the job after making the necessary adjustments.

- **Retry the Staging Deployment**:
  - In the pipeline view, click the **Retry** button to redeploy the staging environment.

Optional: Enhancing the Pipeline with Automated Tests

To further enhance your pipeline, you can add automated acceptance tests after the staging deployment. These tests can act as a gate for the production deployment.

Add an Acceptance Test Job:

```
acceptance_test:

  stage: test

  script:

    - ./run-acceptance-tests.sh

  dependencies:

    - deploy_staging
```

- **Make Production Deployment Dependent on Test Results**:
  - Add the acceptance test as a prerequisite for the production deployment:

```
deploy_prod:

  stage: deploy

  script:

    - apk add --no-cache aws-cli

    - aws s3 cp ./public/ s3://$S3_BUCKET/ --recursive

  environment:
```

```
  name: prod

dependencies:

  - acceptance_test

rules:

  - if: $CI_COMMIT_TAG

    when: manual
```

Conclusion

This setup provides a fully functional CI/CD pipeline, automating deployments to QA, staging, and production. Using Git tags ensures that the same commit that passes QA is deployed to production, while the manual intervention for production adds an extra level of control.

Releasing to Production with CI/CD

In this section, we will walk through the process of releasing code to production using a CI/CD pipeline, handling potential issues like bugs, rolling back deployments, and improving the pipeline with automated tests. This demonstrates how continuous integration and continuous deployment (CI/CD) pipelines enhance the software release process.

Step 1: Releasing to Production

Once your code has passed all stages in QA and staging, you are ready to release it to production.

1. **Manual Testing in Staging**:
   - The QA team performs manual tests in staging.
   - Once satisfied, the code is ready for production.

Production deployment in a CI/CD pipeline can be as simple as clicking a button:

```
deploy_prod:
```

```
stage: deploy

script:

  - echo "Deploying to production..."

environment:

  name: prod

when: manual
```

2. **Trigger Production Deployment**:
   - Navigate to the pipeline and click the **deploy_prod** button to release the code to production.
3. **View Production Deployment**:
   - Once the deployment is complete, view the live website or application to ensure the deployment is successful.

Step 2: Handling Bugs in Production

If a bug slips through QA and staging, it can still be deployed to production. Here's how to identify and resolve it.

1. **Simulate a Bug**:

Make a change in the code that introduces a bug:

```
<p>This is a bug</p>
```

   - Commit the change, which triggers the pipeline, deploying the code to QA.

2. **Check QA and Staging**:

After deployment to QA, verify that the bug is present:

```
git push origin main
```

  - If the bug isn't caught in QA or staging, it might reach production during the next deployment.
3. **Deploy to Production with Bug**:

If the bug is deployed to production, users will notice and the support team will report the issue. For example:

```
<p>This is a bug</p>
```

Step 3: Rolling Back a Bug in Production

CI/CD pipelines make it easy to roll back to a previous working state in production.

1. **Rollback the Deployment**:
   - In GitLab, you can roll back to a previous deployment version:
     - Go to **Deployments > Environments**.
     - Click **Rollback** on the version before the bug (e.g., `1.1`).

This triggers a redeploy of the working version:

```
deploy_prod:

  stage: deploy

  script:

    - echo "Rolling back to version 1.1..."
```

2. **Verify the Rollback**:
   - After the rollback, view the production environment to confirm that the bug has been removed.

Step 4: Adding Automated Tests to Prevent Bugs

To prevent similar bugs from being deployed again, you can add automated tests to your pipeline.

1. **Create a Test for the Bug**:

Use the following script to search for the bug in the code. This test will fail if the bug is found:

```
test_bug:

  stage: test

  script:

    - ! grep bug public/index.html
```

2. **Commit the Changes**:

After adding the test, commit the changes. The pipeline will automatically run and should catch the bug in future deployments:

```
git add .

git commit -m "Add test for bug"

git push origin main
```

3. **Fix the Bug**:

Once the bug is identified, fix it in the code:

```
<p>This is a feature</p>
```

- Commit the fix, and the pipeline will re-run, ensuring the bug no longer exists.

Step 5: Deploying the Fixed Code to Production

After fixing the bug and passing the tests, you can safely deploy the code to production.

1. **Create a New Tag**:

After the tests pass, tag the new release (e.g., `1.3`):

```
git tag 1.3

git push origin 1.3
```

2. **Deploy to Staging and Production**:

The pipeline will automatically deploy the fixed code to staging, and you can manually release it to production:

```
deploy_prod:

  stage: deploy

  script:

    - echo "Deploying feature to prod..."

  environment:

    name: prod

  when: manual
```

Conclusion

This demonstrates the power of CI/CD pipelines, allowing for smooth production releases, easy rollbacks in case of bugs, and automated testing to prevent recurring issues. Starting with a simple pipeline setup provides immediate benefits, and you can build complexity as needed to further enhance the deployment process.