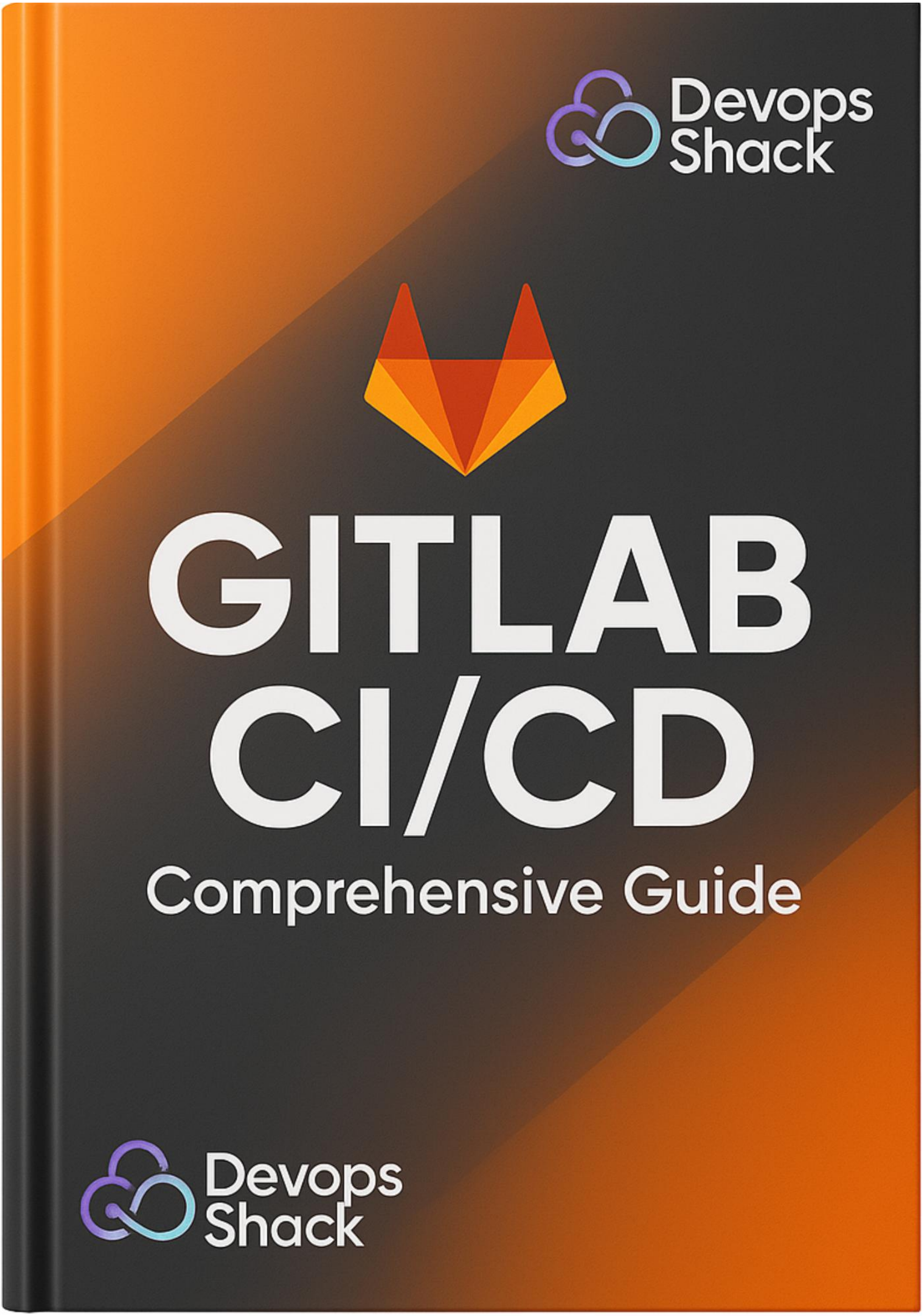

Devops Shack

GITLAB CI/CD

Comprehensive Guide

Devops Shack

DevOps Shack

GitHub Comprehensive Guide

Ultimate GitHub Actions CI/CD Pipeline Guide

Introduction to GitLab CI/CD

1. What is CI/CD?

CI/CD stands for Continuous Integration and Continuous Delivery (or Deployment), and it's one of the cornerstones of modern DevOps practices. It helps teams deliver software more frequently, reliably, and with better quality. The practice has evolved from a need to streamline the traditional software development lifecycle (SDLC), which was often plagued by long integration cycles, difficult testing, and error-prone manual deployments.

Continuous Integration (CI)

CI is the practice of automatically integrating code changes from multiple contributors into a shared repository several times a day. Each integration is verified by an automated build and test process, allowing teams to detect problems early.

Key aspects of CI include:

- Developers push code frequently to a central repository.
- Automated tools build and test the code.
- Failures are reported immediately, enabling quick resolution.

Continuous Delivery (CD)

CD extends CI by automatically preparing code changes for release to production. It ensures that software can be reliably released at any time. This includes automated testing, staging, and release processes.

Continuous Deployment

This is a step beyond continuous delivery. In continuous deployment, every change that passes all stages of the production pipeline is automatically deployed to users, without manual intervention.

2. Why CI/CD is Crucial in Modern Development

The modern software development landscape demands speed, agility, and quality. Here's why CI/CD is crucial:

- **Faster Time to Market:** Automating build, test, and deployment significantly reduces the time from development to release.
- **Improved Code Quality:** Frequent integration and automated testing help identify and fix issues earlier.
- **Reduced Risk:** Smaller, incremental changes are easier to test, review, and deploy.
- **Enhanced Collaboration:** CI/CD encourages developers to work together more closely, resulting in more cohesive software.
- **Supports Agile and DevOps Practices:** It aligns perfectly with the iterative nature of Agile development and the automation focus of DevOps.

3. GitLab's Role in CI/CD

GitLab is a complete DevOps platform delivered as a single application. It provides built-in CI/CD capabilities that are tightly integrated with its version control system, issue tracking, and security tools. This all-in-one approach eliminates the need to integrate multiple tools and simplifies DevOps workflows.

GitLab's Key CI/CD Features:

- Native support for pipelines via .gitlab-ci.yml
- Built-in Docker registry and Kubernetes integration
- Visual pipeline or and pipeline graphs
- Auto DevOps for intelligent pipeline generation
- Secret management and variable injection
- Extensive API and webhook support

4. Real-World Use Cases for GitLab CI/CD

GitLab CI/CD is versatile and can handle a variety of use cases:

A. Startups and Small Teams

- Fast setup, minimal tooling required
- Simplified deployment to cloud services (Heroku, AWS, etc.)
- Auto DevOps for getting started quickly

B. Enterprise Software Development

- Supports large teams and complex workflows
- Compliance, audit logs, and security integration
- Role-based access control (RBAC) and LDAP integration

C. Microservices Architectures

- Run services independently in their own pipelines
- Trigger child pipelines based on changes
- Isolated environments and deployments

D. Monorepo Management

- Conditional pipeline execution
- Directory-based triggers
- Efficient handling of large repositories

E. Machine Learning/AI

- Automate model training and evaluation
- Push trained models to storage or serve them via APIs
- Use CI/CD for reproducibility and model governance

F. Infrastructure as Code

- Deploy and manage infrastructure using Terraform, Ansible, etc.
- GitOps practices via GitLab environments
- Configuration drift detection and rollback

5. Key Benefits of GitLab CI/CD

GitLab provides several benefits that make it a preferred CI/CD tool for many organizations:

- **Single Source of Truth:** One platform for code, CI/CD, security, and monitoring.
- **Tight SCM Integration:** Seamless integration with merge requests, branches, and commits.
- **Security Built-in:** Static and dynamic analysis, dependency scanning, license compliance.
- **Scalability:** Works for small projects and large-scale enterprise pipelines.
- **Efficiency:** Speed up pipelines with caching, parallel jobs, and optimized runners.

6. Evolution of GitLab CI/CD

GitLab CI/CD has matured rapidly over the years:

- **2015:** Introduction of GitLab CI with .gitlab-ci.yml
- **2016:** Docker integration and shared runners
- **2017:** Auto DevOps introduced
- **2019:** Child pipelines and parent/child relationships
- **2020+:** Kubernetes integration, security scanning, GitOps workflows, pipeline efficiency improvements

Each version of GitLab introduces new features, often driven by community feedback. GitLab's open-core model enables rapid iteration and innovation.

7. Basic CI/CD Flow in GitLab

Understanding the basic flow helps visualize how GitLab CI/CD operates:

1. **Code Commit:** A developer commits code to a branch.
2. **Trigger:** The push triggers the pipeline defined in .gitlab-ci.yml.
3. **Runner Execution:** A GitLab Runner picks up the job and starts executing tasks.

4. **Build & Test:** Jobs defined in stages (build, test, deploy) run in order.
5. **Artifacts & Reports:** Output (e.g., test results, coverage reports) is stored as artifacts.
6. **Deployment:** If successful, the application is deployed to the target environment.
7. **Feedback Loop:** Notifications, dashboards, and logs provide instant feedback.

📘 How GitLab CI/CD Works

🚀 1. Overview of the GitLab CI/CD Workflow

At its core, GitLab CI/CD is a pipeline-based automation system that integrates tightly with your code repository. It watches your GitLab project for changes (commits, merges, pull requests), and based on your configuration (.gitlab-ci.yml), it automatically triggers a pipeline.

🔄 Basic Flow:

1. **Code Commit/Pull Request** → triggers a pipeline.
2. **Pipeline Initialized** → defined stages begin execution.
3. **Jobs Execute via GitLab Runners** → using executors (Docker, Shell, Kubernetes, etc.)
4. **Artifacts and Reports** → are stored/shared across jobs.
5. **Environment Deployment** → to dev, staging, or production.
6. **Feedback Loop** → integrates with GitLab Issues, MRs, and security/compliance checks.

⚙️ 2. Understanding Pipeline Triggers

There are several ways a pipeline can be triggered in GitLab:

- **Push-based triggers**: Whenever you push a commit to a specific branch (e.g., main, develop).
- **Merge Request triggers**: Pipelines run to validate MRs before merging.
- **Scheduled Pipelines**: Useful for nightly builds, test runs, or backups.
- **Manual Triggers**: Jobs that require human approval (e.g., production deployment).
- **API/Webhook Triggers**: You can invoke pipelines externally using GitLab's API or integrate with third-party systems.

🔎 Example: A pipeline:run webhook might be triggered from a JIRA card status change.

🧱 3. Anatomy of a Pipeline: Stages, Jobs, and Steps

A **pipeline** is the top-level container of your CI/CD workflow.

🔹 Stages:

These define the order of execution. Common stages:

- build
- test
- package
- deploy
- review
- cleanup

Stages run sequentially — all jobs in build must succeed before moving to test.

🔹 Jobs:

Each stage can contain one or more jobs. These are tasks like:

- Running tests
- Building Docker images
- Performing static code analysis

Jobs within a stage run **in parallel** (if runners are available).

🔹 Steps (within a job):

Defined using the script: keyword, they're the actual shell commands executed.

🔄 4. Lifecycle of a Job Execution

Each job execution follows this sequence:

1. **Runner picks up the job** from GitLab's queue.
2. **Pre-job phase**:
   - Cloning the repo
   - Setting up environment variables
3. **Job execution**:
   - Executes the script: commands in a clean environment
4. **Post-job phase**:
   - Uploading artifacts
   - Storing logs
   - Sending success/failure status back to GitLab

If a job fails, dependent jobs (unless allow_failure: true) won't execute.

🏃 5. GitLab Runners: The Workhorses

Runners are lightweight agents responsible for executing your pipeline jobs.

🛠️ Types of Runners:

- **Shared Runners**: Available across all GitLab projects (good for general use).
- **Specific Runners**: Bound to a particular project or group.
- **Group Runners**: Available to multiple projects under a group.

🧪 Executors:

- **Shell**: Executes commands in the host shell.
- **Docker**: Spins up containers to isolate jobs.
- **Kubernetes**: Executes jobs in a pod on a K8s cluster.
- **Custom**: You can build custom executors as needed.

🎯 Use Docker or K8s for consistent build environments and better isolation.

🔐 6. Authentication & Permissions

- Jobs run with permissions defined by GitLab's role system (Guest, Reporter, Developer, Maintainer).
- Protected branches and environments ensure only authorized users can deploy.
- Token-based authentication is used for runners and external services (e.g., image registries, AWS, Vault).

🧰 7. Integrations with the GitLab Ecosystem

One of GitLab CI/CD's key strengths is how deeply it's integrated with the GitLab ecosystem.

- **Merge Requests**: Pipeline status shows directly in the MR.
- **Issue Boards**: Auto-link commits, pipelines, and deployments to issues.
- **Security Scanning**: Built-in SAST, DAST, container scanning.
- **Release Management**: Ties releases to tags, jobs, and change logs.

📈 8. Feedback & Observability

GitLab provides built-in visibility into every step of your pipeline:

- **Pipeline Graph**: Shows stages and jobs with success/failure indicators.
- **Job Logs**: Each job's output is viewable in the UI.
- **Test Reports**: JUnit-style reports are parsed and displayed.
- **Code Quality Reports**: Summarized inline with MRs.
- **Security Dashboards**: Highlight vulnerabilities across projects.

🔁 9. Re-runs, Manual Actions & Rollbacks

- You can **re-run** failed jobs or entire pipelines from the UI or CLI.
- **Manual actions** (when: manual) allow for controlled deployments.

- Pipelines can also include **rollback steps** in case deployments fail.

💡 10. Use Case Walkthrough: Example Workflow

Let's walk through a typical GitLab CI/CD pipeline for a web app:

✅ Code Flow:

1. Developer pushes code to feature/new-ui.
2. Pipeline triggers on push:
   - Stage 1: lint job → checks syntax
   - Stage 2: test job → runs unit tests
   - Stage 3: build job → compiles app, creates Docker image
   - Stage 4: deploy_dev → deploys to dev environment
3. A Merge Request is opened into main.
4. A new pipeline runs with stricter checks (e.g., security scan).
5. Once approved and merged:
   - Stage 5: deploy_staging
   - Manual stage: deploy_prod (requires approval)
6. Rollback logic is included if health-check fails post-deployment.

🛡 11. Security & Compliance in Action

- Jobs can access masked CI/CD variables (e.g., API keys).
- GitLab Vault integration allows fetching secrets securely.
- Only authorized runners or branches can trigger sensitive jobs.
- Compliance reports help enforce security and audit policies.

🔗 12. Connecting External Systems

You can extend GitLab CI/CD to interact with:

- Slack (notifications)
- JIRA (issue updates)
- Terraform (infra deployment)
- AWS/GCP/Azure (cloud ops)
- DockerHub/ECR/GitLab Container Registry

🧩 13. Dynamic Pipelines & DAG

GitLab supports **Dynamic Pipelines**:

- Pipelines that are generated at runtime based on logic (e.g., presence of certain files).
- Great for monorepos or conditional logic.

DAG Pipelines (Directed Acyclic Graphs):

- Jobs run as soon as their dependencies are met (more efficient than linear execution).

🧠 14. Summary: From Code to Production

In summary, GitLab CI/CD simplifies the entire software lifecycle:

- Code → Commit → Pipeline → Build → Test → Deploy → Monitor
- Each step is automated, repeatable, and traceable.
- It scales from solo developers to massive enterprises.

Whether you're building a small app, deploying ML models, or managing infrastructure, GitLab CI/CD offers a robust, integrated, and powerful way to automate everything from source to production.

📘 Key Components and Terminologies in GitLab CI/CD

Understanding the key components of GitLab CI/CD is essential for anyone aiming to build, manage, or optimize pipelines effectively. Let's break down each of the major terms and concepts you'll encounter.

🔧 1. Pipeline

A **pipeline** is the core structure in GitLab CI/CD. It represents the sequence of automated processes that get triggered on code changes. Pipelines consist of **stages**, and within those stages are individual **jobs**.

Think of it like a factory assembly line:

- The **pipeline** is the whole line.
- **Stages** are the main steps in the process (e.g., build → test → deploy).
- **Jobs** are the workers in each step doing specific tasks.

Example:

```
stages:
  - build
  - test
  - deploy

build_job:
  stage: build
  script: echo \"Building...\"

test_job:
  stage: test
  script: echo \"Testing...\"

deploy_job:
  stage: deploy
  script: echo \"Deploying...\"
```

⚙️ 2. Jobs

A **job** is a single task that runs as part of a pipeline. Each job runs in its own isolated environment, and jobs in the same stage can run in parallel (if resources allow).

Job attributes:

- script: – the actual shell commands to run
- stage: – determines the order
- only: / except: – control execution logic
- artifacts: – files to persist and pass on
- tags: – used to assign jobs to specific runners

Example job:

```
test_job:
  stage: test
  script:
    - npm install
    - npm run test
```

🧱 3. Stages

Stages define the execution order of jobs in a pipeline. All jobs in one stage must complete before the next stage begins.

Common stages:

- build
- test
- review
- staging
- production

Stages are declared in the stages: keyword, and every job must specify which stage it belongs to.

Example:

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

🤖 4. GitLab Runner

A **GitLab Runner** is an agent that executes the jobs defined in the pipeline. It's the executor that runs your scripts, builds containers, runs tests, etc.

There are two types:

- **Shared Runners** – available for all projects in GitLab
- **Specific Runners** – dedicated to a particular project or group

Executors used by Runners include:

- shell – runs on host machine
- docker – jobs run in containers
- kubernetes – jobs run in pods
- custom – user-defined execution method

📦 5. Artifacts

Artifacts are files or directories generated by a job and saved to be used later in the pipeline or for download. Common artifacts include:

- Test reports
- Build binaries
- Coverage results
- Logs

Example:

```
build:
  stage: build
  script:
    - make build
  artifacts:
    paths:
      - build/
```

Artifacts can also be set to expire automatically after a certain period.

🚀 6. Environments

Environments represent where your application is deployed (e.g., dev, staging, prod). GitLab tracks deployments and gives you a UI to manage environment status, URLs, and rollback history.

You can define:

- Static environments (e.g., production)
- Dynamic environments (e.g., per-branch or per-merge-request preview apps)

Example:

```
deploy_prod:
  stage: deploy
  script:
    - ./deploy.sh
  environment:
    name: production
    url: https://app.example.com
```

🔐 7. Variables

GitLab CI/CD uses **variables** to manage environment settings, secrets, and dynamic values. They can be:

- **Predefined** (e.g., CI_COMMIT_SHA)
- **Custom** (defined in UI or .gitlab-ci.yml)

- **Masked** and **Protected** for sensitive data

Example:

```
script:
  - echo $MY_SECRET
```

They can also be defined at:

- Project level
- Group level
- Runner level
- Job level

📁 8. Cache

Cache helps speed up jobs by reusing previously downloaded files, like dependencies or build outputs. Unlike artifacts, caches are not meant for passing data between jobs, but for optimization.

Example:

```
cache:
  paths:
    - node_modules/
```

You can also define key: for versioning cache uniquely per branch or job.

🧬 9. Includes & Templates

To avoid repetition, GitLab allows you to **include** external files or use **templates**. This makes your .gitlab-ci.yml modular and easier to manage.

Example:

```
include:
  - project: 'my-group/common-templates'
    file: '/templates/deploy.yml'
```

You can also use built-in GitLab templates for things like Node.js, Docker, Python, etc.

📡 10. Triggers

Triggers are ways to start pipelines externally or conditionally:

- Manual jobs (when: manual)
- API Triggers (via token)
- Upstream/downstream pipelines
- Scheduled pipelines
- rules: for advanced control over pipeline logic

Manual Trigger Example:

```
deploy_prod:
  stage: deploy
  script: ./deploy.sh
  when: manual
```

🛡 11. Protected Branches & Jobs

Protected features in GitLab limit who can push or run jobs on sensitive branches or environments like main or production.

- Only certain users or roles can trigger jobs on protected branches.
- Protected variables are only available to protected branches.

🔁 12. Retry, Timeout & Fail Rules

You can fine-tune job behavior:

- retry: to auto-retry on failure
- timeout: to limit job runtime
- allow_failure: to let pipelines pass even if a job fails

Example:

```
job:
  script: run-something
  retry: 2
  timeout: 10 minutes
```

📊 13. Reports

GitLab supports various reports that can be uploaded via artifacts:

- **JUnit** (test results)
- **Code Coverage**
- **SAST / DAST / Dependency Scanning**
- **License Scanning**

These are automatically displayed in MRs if formatted correctly.

📌 Summary Table

Component	Description
Pipeline	Sequence of automated stages/jobs
Job	Individual unit of work
Stage	Ordered group of jobs
Runner	Executes pipeline jobs
Artifact	Files saved for future use
Environment	Target deployment context
Variables	Dynamic values/secrets
Cache	Optimized re-use of data

Component	Description
Includes	modularization
Triggers	Manual/API/conditional execution
Protected	Restricted access for security
Reports	Display build/test/security info

This section equips you with a solid understanding of all the essential terms and moving parts in GitLab CI/CD. You'll now be able to read, write, and debug .gitlab-ci.yml files with much more clarity.

📘 .gitlab-ci.yml Structure

The .gitlab-ci.yml file is the **heart** of GitLab CI/CD. It defines the entire pipeline, including what jobs to run, how to run them, and under what conditions. Written in syntax, this file lives at the root of your repository and gets picked up automatically by GitLab when changes are pushed.

Let's break it down step by step and explore everything you need to master this file.

🧩 1. Basic Structure

The structure of a basic .gitlab-ci.yml file looks like this:

```
stages:
  - build
  - test
  - deploy

build_job:
  stage: build
  script:
    - echo "Building the app"

test_job:
  stage: test
  script:
    - echo "Running tests"

deploy_job:
  stage: deploy
  script:
    - echo "Deploying to production"
```

Each job is mapped to a stage, and each job contains at minimum a script: section.

⚙️ 2. Defining Stages

Use the stages: keyword to define the order in which stages (and thus jobs) run. All jobs within a stage will run **in parallel**, but **stages themselves run sequentially**.

```
stages:
  - lint
  - build
  - test
  - deploy
```

🔹 All jobs in lint run → then build → then test → then deploy.

🔨 3. Writing Jobs

Each job is defined as a key-value pair:

```
job_name:
  stage: <stage_name>
  script:
    - <command 1>
    - <command 2>
```

Common job attributes:

- script: – the commands to run
- stage: – which stage this job belongs to
- image: – base Docker image for the job

- artifacts: – files to preserve
- cache: – speeds up builds
- only: / except: – control when the job runs
- tags: – runner targeting

🐳 4. Using Docker Images

You can specify a Docker image per job or globally:

Global image:

```
image: node:18


stages:

  - test


test_job:

  script:

    - npm install

    - npm test
```

Job-level image:

```
test_job:

  image: python:3.9

  script:

    - pip install -r requirements.txt

    - pytest
```

🧰 5. Advanced Job Configuration

Parallel Jobs

Run multiple instances of the same job (e.g., for matrix builds):

```
test:
  script: run_tests.sh
  parallel: 5
```

Retry on Failure

```
job:
  script: ./test.sh
  retry: 2
```

Timeout

```
job:
  script: heavy_task.sh
  timeout: 20 minutes
```

🔁 6. Rules vs. Only/Except

only / except (Legacy, but still used):

```
job:
  script: run.sh
  only:
    - main
```

rules: (More flexible):

```
job:
  script: run.sh
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      when: always
```

You can also use rules for dynamic pipelines (more on that later).

🧪 7. Artifacts & Caching

Artifacts:

Used to **save outputs** of a job (e.g., test reports, binaries).

```
build:
  script: make build
  artifacts:
    paths:
      - build/
    expire_in: 1 week
```

Cache:

Used to **speed up** pipelines (e.g., dependency folders).

```
cache:
  paths:
    - node_modules/
```

You can use key: for versioning cache:

```
cache:
  key: "$CI_COMMIT_REF_NAME"
  paths:
    - node_modules/
```

🔐 8. Using Variables

Define in GitLab UI or .gitlab-ci.yml:

```
variables:
  NODE_ENV: production
```

Then use in jobs:

```
job:
  script:
    - echo $NODE_ENV
```

Also supports secrets via CI/CD variables UI, Vault, or group-level settings.

📥 9. Includes and Templates

To reduce duplication, GitLab supports breaking up CI/CD config into reusable templates.

Local include:

```
include: 'templates/deploy.yml'
```

Remote project include:

```
include:
  - project: 'group/project'
    file: '/templates/test.yml'
    ref: 'main'
```

Multiple includes:

```
include:
  - 'common.yml'
  - 'docker-build.yml'
```

📋 10. Best Practices

- **Use anchors & aliases** to reuse job definitions:

```yaml
.default_job: &default_job
  script:
    - echo "Default behavior"

job1:
  <<: *default_job

job2:
  <<: *default_job
```

- **Avoid deeply nested structures**
- **Use comments** to document your pipeline logic
- **Split config** into includes if it gets too long

🔍 11. Debugging Pipelines

- Use echo statements to trace logic
- Check **Job Logs** in GitLab UI
- Use when: manual and allow_failure: true for experimentation
- Download artifacts for failed jobs
- Use CI_DEBUG_TRACE=true for verbose logging

🧠 12. Real-World Example: CI for Node.js App

```
image: node:18
stages:
  - install
  - test
  - deploy

install_dependencies:
  stage: install
  script:
    - npm install
  cache:
    paths:
      - node_modules/

unit_tests:
  stage: test
  script:
    - npm run test

deploy_prod:
  stage: deploy
  script:
    - npm run build
    - ./scripts/deploy.sh
  environment:
    name: production
    url: https://yourapp.com
  only:
    - main
```

✅ Summary of Key Directives

Directive	Description
stages	Defines execution order
script	Commands to run in job

Directive	Description
image	Docker image to use
variables	Env variables in job
cache	Preserved folders between jobs
artifacts	Files saved after job
only/rules	When to run job
tags	Match runners
include	Import external config
retry/timeout	Job control

This section gives you a full grasp of how .gitlab-ci.yml is structured and how to write clean, scalable, and powerful pipelines using GitLab's DSL.

7. Artifacts, Caching & Dependencies

🧳 Artifacts

Artifacts are job outputs that you want to **preserve after the job ends**. They're useful for:

- Sharing test results
- Keeping build files for deployment
- Retaining logs or coverage reports

```
build:
  script: npm run build
  artifacts:
    paths:
      - dist/
    expire_in: 1 week
```

📇 Caching

Caching helps **speed up pipeline execution** by reusing dependencies (e.g., node_modules/, .m2/, or .venv/).

```
cache:
  paths:
    - node_modules/
```

Use key: to control cache versioning and avoid invalidation:

```
cache:

  key: "$CI_COMMIT_REF_SLUG"
```

📦 Dependencies

The dependencies: keyword allows one job to **fetch artifacts from a previous job**, even if they're not in the same stage.

```
test:

  stage: test

  dependencies:

    - build
```

---

8. Environments & Deployments

🗺 What Are Environments?

An **environment** represents a deployment target — dev, staging, or prod. GitLab keeps a deployment history, URLs, and even enables **Review Apps**.

```
deploy_staging:

  environment:

    name: staging

    url: https://staging.myapp.com
```

🚀 Review Apps

Automatically spin up a temporary app per branch or merge request:

```
deploy_review:
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://$CI_ENVIRONMENT_SLUG.example.com
```

🧯 Rollbacks

You can use manual jobs or scripts to rollback:

```
rollback_prod:
  stage: deploy
  script: ./rollback.sh
  when: manual
```

---

9. Security, Secrets, and Vaults

🔐 CI/CD Variables

Use GitLab's UI to store secrets like API keys:

- Masked (not visible in logs)
- Protected (only on protected branches)

```
script:
  - curl -H \"Authorization: Bearer $API_TOKEN\" ...
```

🔒 HashiCorp Vault Integration

GitLab supports **dynamic secret injection** from Vault:

- Use JWT OIDC tokens to authenticate
- Map GitLab variables to Vault paths
- Secrets fetched at runtime, never stored in .yml

🛡️ Security Scanning

GitLab Ultimate offers:

- **SAST** (Static Application Security Testing)
- **DAST** (Dynamic Testing)
- **Dependency Scanning**
- **License Compliance**

All run as pipeline jobs with results shown in Merge Requests.

---

10. CI/CD Templates & Includes

📁 Includes

Modularize pipelines using include: to reuse configs.

```
include:
  - project: my-group/common
    file: '/templates/deploy.yml'
```

Supports:

- Local paths
- Remote URLs
- Project files
- Auto DevOps templates

🔄 Anchors

Define reusable job blocks:

```
.default-job: &default_job
  image: node:18
  script:
    - npm install
```

```
job1:
  <<: *default_job

job2:
  <<: *default_job
```

---

11. GitLab CI/CD for Monorepos & Microservices

🧬 Monorepo Strategy

Use rules: or changes: to run only what's needed:

```
job:
  script: ./run.sh
  rules:
    - changes:
        - service-a/**
```

Combine with dynamic pipelines for flexibility:

```
trigger:
  trigger:
    include: generated-pipeline.yml
```

📡 Microservices Setup

Each service has its own .gitlab-ci.yml:

- Run builds/test/deploy only for changed services
- Use parent-child pipelines to control flow
- Can run independent or orchestrated deployments

---

12. Best Practices

- **Keep Pipelines Fast**
  Use parallel jobs, proper caching, and limit unnecessary tasks.
- **Fail Fast**
  Run lightweight checks (lint, format) early in the pipeline.
- **Use rules: Instead of only/except:**
  More powerful and readable for complex logic.
- **Secure Sensitive Data**
  Mask variables, use protected runners, and never hardcode secrets.
- **Use Modular .yml Files**
  Especially for microservices or multi-stage apps.
- **Monitor & Clean Up Artifacts**
  Use expire_in: to avoid bloated storage.

---

13. Advanced Concepts & Use Cases

⛓ Parent-Child Pipelines

Organize complex systems into smaller units:

```
main:
  trigger:
    include: .child-pipeline.yml
```

🧠 Dynamic Pipelines

Generate .yml dynamically via script or conditional logic.

⚙️ Matrix Builds

Run the same job across multiple configs:

```
parallel:
  matrix:
    - VARIANT: [lite, full]
      OS: [ubuntu, alpine]
```

📈 Custom Metrics & Observability

Use:

- artifacts:reports: for test and code quality
- Monitoring integrations for logs, metrics, and alerts

---

14. Common Pitfalls

❌ Overly Complex .gitlab-ci.yml

Modularize using includes, templates, or anchors.

❌ Ignoring Job Failures

Use allow_failure: true sparingly. Investigate failing jobs early.

❌ Misusing Cache vs. Artifacts

Cache is for speed. Artifacts are for passing files between jobs.

❌ Running All Jobs on Every Push

Use rules: and changes: to avoid unnecessary builds/tests.

❌ Security Gaps

Never expose secrets. Avoid using public runners for sensitive work.

---

15. Final Thoughts

GitLab CI/CD is one of the most powerful and flexible pipeline tools in the DevOps ecosystem. Whether you're building simple web apps or managing enterprise-scale microservices, GitLab provides everything you need:

✅ Full pipeline automation
✅ Integrated security
✅ Dynamic configurations
✅ Multi-environment deployments
✅ Developer-first UX