TO

Transitioning from TFS to GitLab

4 Days workshop

By: Kusai Esseid

ABOUT THE TRAINER

KUSAI MOSBAH ESSEID

Senior DevOps Engineer

qosaiassiad@gmail.com
+601123408314
https://www.linkedin.com/in/kusaiesseid/

Trainer Profile: Kusai Mosbah Esseid

- Kusai is an IT professional with a strong academic and professional background in **Information Technology and Cloud Computing**. He holds a **Bachelor's Degree in Information Technology (2014–2017)** with a GPA of 3.4, followed by a **Master's Degree in Information Technology (2018–2019)** where he graduated with a GPA of 3.87 and received the **Best Student Award**. Currently, he is pursuing a **PhD in Computer Science**, focusing on *Online News Recommendation using Deep Learning*, demonstrating his continuous pursuit of innovation and technical excellence.

- Professionally, Kusai has developed extensive experience in **DevOps, Cloud Infrastructure, and Security Operations** through his work at **Ørsted**, a global leader in renewable energy. As part of the **Engineering Cloud System Team**, he contributes to the organization's **multi-cloud transformation**, leading initiatives that span **automation, governance, and platform reliability** across **Azure and AWS**. He plays a key role in **migrating workloads from Azure DevOps and AKS to GitHub Actions and AWS EKS**, implementing **Infrastructure as Code (Terraform)**, **Kubernetes cluster lifecycle management**, and **secure CI/CD automation**. Kusai also drives **Velero-based backup strategies**, **KEDA auto-scaling configurations**, and **JFrog Artifactory governance**, ensuring operational excellence, compliance, and resilience across environments.

- Kusai is fluent in **Arabic and English**, enhancing his ability to communicate effectively with diverse global teams. His technical credentials include **Certified Kubernetes Administrator (CKA)**, **Alibaba Cloud Professional (Cloud Computing & Security)**, **Microsoft Azure Fundamentals**, **ITIL 4**, and **Certified Trainer (TTT)** — reflecting his depth in **cloud infrastructure, DevOps automation, and IT service management**.

- His career journey includes roles such as **Linux System Administrator** at ITRANSCEND (M) SDN BHD, **Cloud Computing Architect** at ARINAA KAMBYAN BERHAD, and **System Administrator** at AppAsia Tech Sdn Bhd. Currently, he serves as **Cloud Manager at AIA Digital+ in Kuala Lumpur**, overseeing cloud-native operations and managing **Azure Kubernetes Service (AKS)** for multiple business units.

- Kusai's skills span **Cloud Computing, DevOps Automation, Kubernetes and Linux Administration, GitLab, GitHub, Network Security, Firewall and FortiGate Management, Virtualization (Proxmox, VMware, KVM), Database and Web Server Administration, and Technical Training**. His blend of deep technical expertise and real-world DevOps experience enables him to design, secure, and optimize cloud-native environments while mentoring teams to adopt best practices in automation and governance.

Class Material

- https://tinyurl.com/tfs-git

Agenda

Day 1: From TFS to GitLab: Foundations

Understanding the Shift, Git & GitLab Basics, Hands-on Lab

Day 2: Collaboration & Code Quality in .NET Projects

Collaboration, Traceability, Code Quality, Hands-on Lab

Day 3: CI/CD for .NET in GitLab

Pipelines, Delivery Efficiency, Rollback, Hands-on Lab

Day 4: Standardized Practices & Developer Autonomy

Standardization, Empowering Developers, Capstone Simulation, Wrap-Up

Workshop Objectives

By the end of this course, participants will be able to:

- Successfully migrate TFS projects into GitLab repositories.
- Manage .NET solution files and dependencies under Git branching workflows.
- Use GitLab CI/CD to build and test .NET Core/.NET Framework projects automatically.
- Replace TFS Build/Release pipelines with GitLab pipelines.
- Integrate quality gates such as SonarQube, NUnit, and code coverage tools into GitLab CI.
- Confidently debug and fix issues in Git pipelines without relying on legacy TFS tools.


DAY 1

TO

From TFS to GitLab: Foundations

Understanding the Shift, Git & GitLab Basics, Hands-on Lab

How TFS (TFVC) Differs From Git

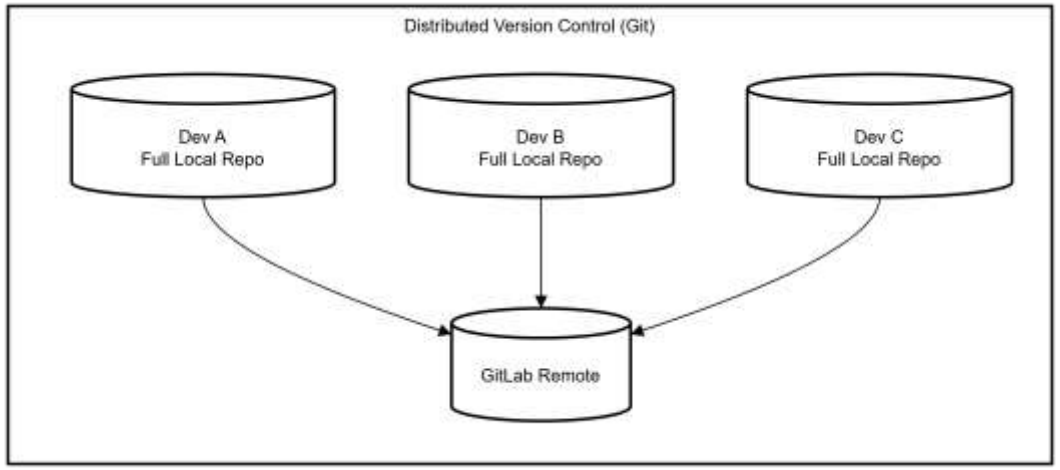
- **Centralized Version Control (TFS/TFVC)**
  - One central server stores all versions
  - Developers depend on server availability
  - Check-in / check-out workflow
  - Weak offline support
  - Branching is heavy and slow
  - Changes are serialized (historically locking-based model)

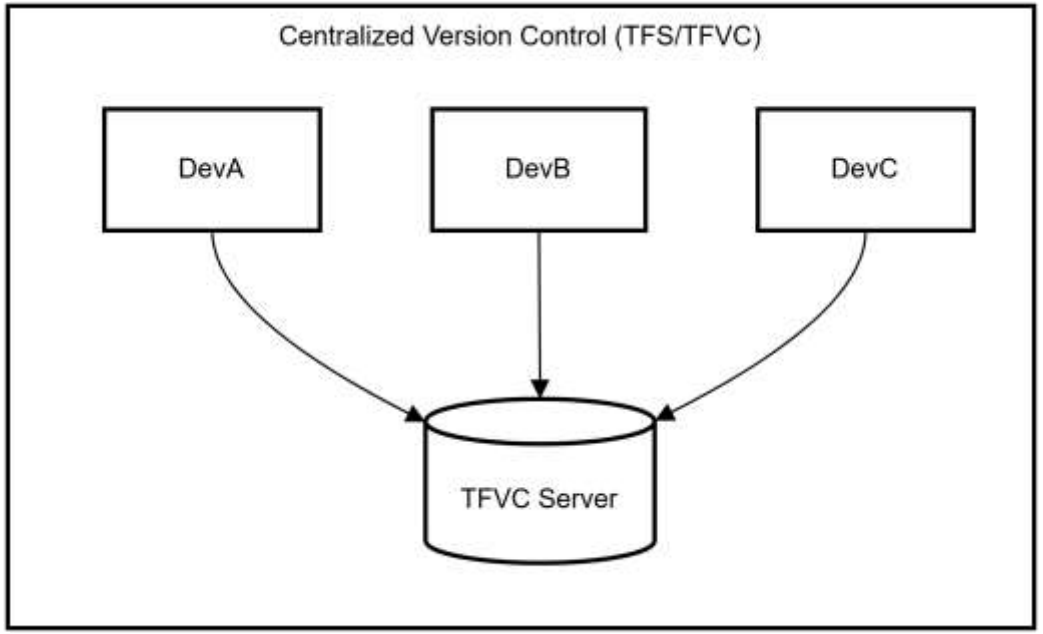
How TFS (TFVC) Differs From Git

- **Distributed Version Control (Git)**
  - Every developer has a full local repository
  - Commits & operations are local (faster)
  - Offline work is fully supported
  - Branching is lightweight and encouraged
  - Push/pull for collaboration instead of check-in

TFS vs Git

Area	TFS (Centralized)	Git (Distributed)
Architecture	Single central repo	Local repo per developer
Offline Work	Very limited	Full offline workflow
Branching	Expensive, avoided	Cheap, used frequently
Merging	Hard, error-prone	Optimized for merging
Speed	Dependent on server	Local operations = very fast
Collaboration	Check-in policies, locking	Merge Requests, branching model
History	Stored only on server	Each user has full history
Suitable For	Monolithic legacy apps	Modern DevOps, microservices


Distributed Version Control (Git)
Dev A
Full Local Repo
Dev B
Full Local Repo
Dev C
Full Local Repo
GitLab Remote


Centralized Version Control (TFS/TFVC)
DevA
DevB
DevC
TFVC Server

Why the Architecture Matters

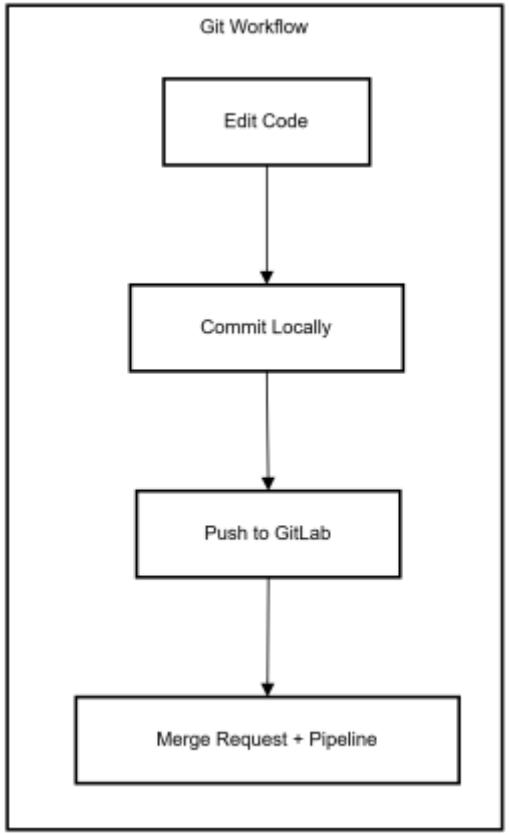
Impact of Centralized (TFS)

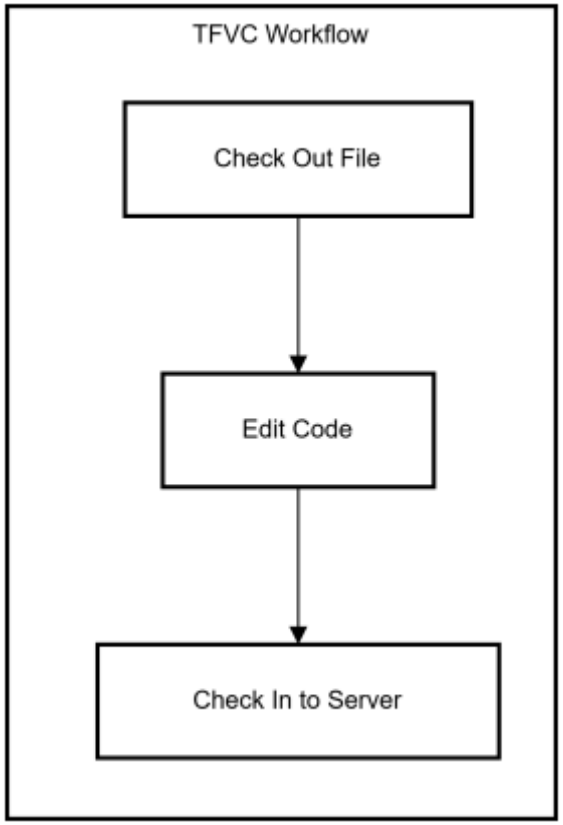
- Slow operations (server round-trips)
- Merging becomes bottleneck
- Branches avoided → larger changes → more conflicts
- No autonomy when offline
- Hard to scale for modern DevOps

Impact of Distributed (Git)

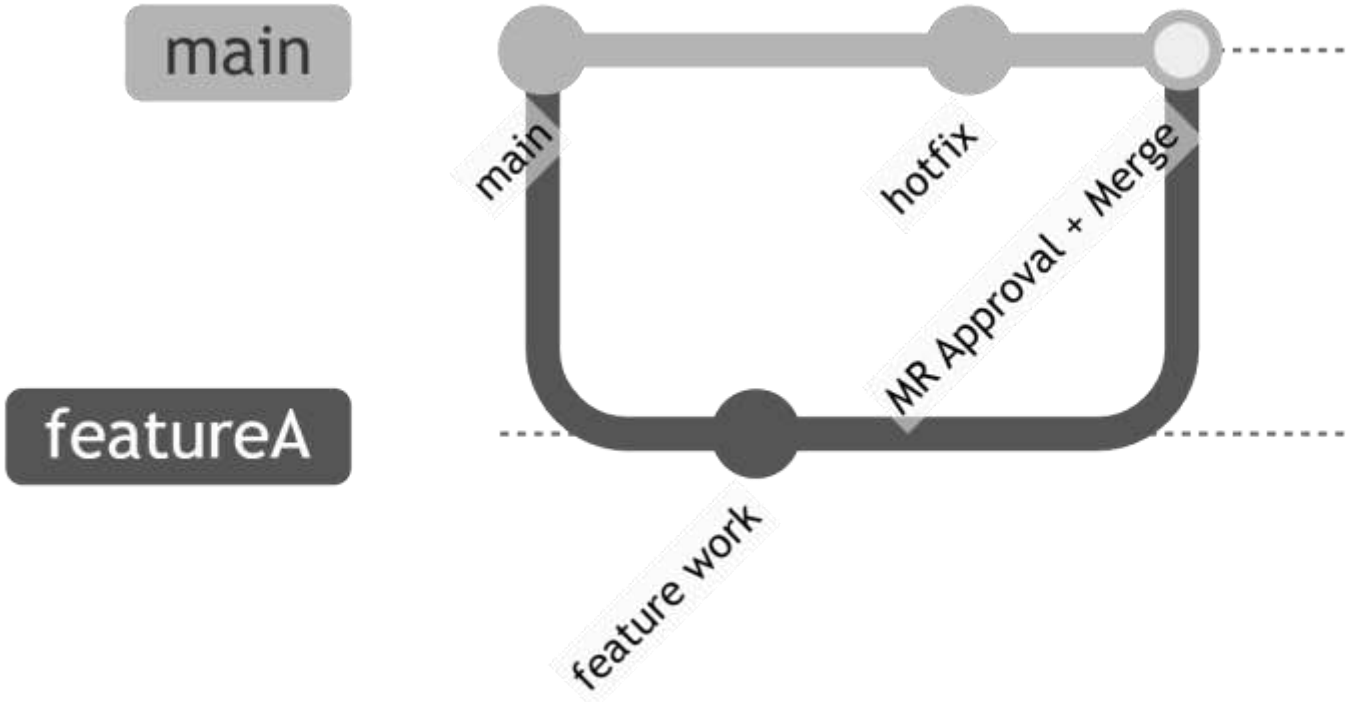
- Developers make progress anytime (offline supported)
- Fast commits, fast diffs, fast merges
- Small branches → clean change history
- Enables GitLab CI/CD for every branch
- Better collaboration & traceability

Workflow Differences (.NET Developer View)





Branching



What Changes for Teams Migrating from TFS

What stays the same

- You still edit code in Visual Studio
- You still commit changes
- You still have branches

What improves drastically

- Local commits = faster & safer
- Conflicts resolved earlier, not at release time
- CI/CD runs on every branch automatically
- Pull Requests (MRs) enforce quality
- Traceability: commits → issues → pipelines → artifacts

New mindset

- Think in **small units of work**
- Commit locally often
- Create branches freely
- Use MR as your “check-in policy replacement”

Migration Considerations Overview

Before migrating from TFS → GitLab, three areas must be addressed:

- **History:** Do we migrate all changesets or just the latest snapshot?
- **Permissions:** TFS folder-based security must be mapped to GitLab project/branch-based permissions.
- **Branch Structures:** TFVC's long-lived branches don't map 1:1 to Git; we must redesign branching strategy.

Key Message:

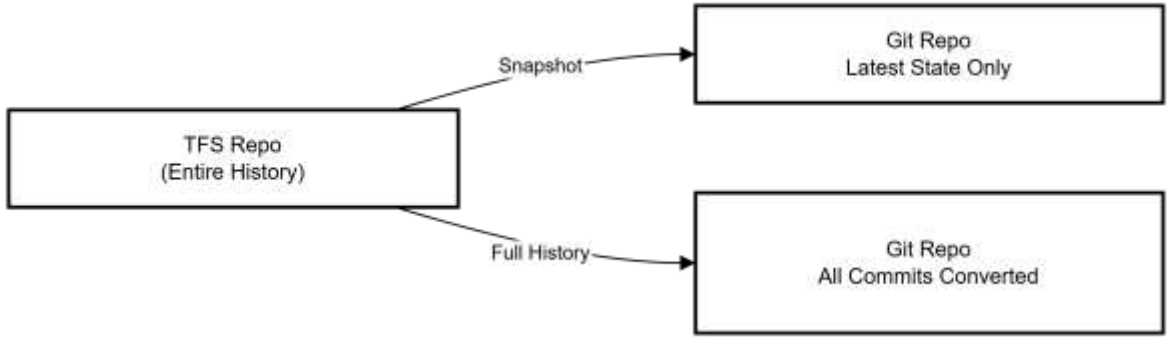
- Migration is not a "copy–paste" of source code — it is a **workflow transformation**.

History Considerations

Two migration approaches:

1. Snapshot Migration (recommended)

- Only migrate the latest version of the code
- Clean start → no baggage
- Great for modernizing branching & pipelines
- Easiest to automate



2. Full History Migration

- Converts TFS changesets → Git commits
- Useful only if compliance or audits require full history
- Slower, risk of corrupt paths, large repo size

Permissions Considerations

TFS permission model ≠ GitLab permission model.

TFS Permissions

- Folder-level security
- Check-in permissions
- Locking / exclusive checkout

GitLab Permissions

- No folder-level security
- Access is granted at **project** or **group** level
- Fine control via:
  - **Branch protection**
  - Who can push
  - Who can merge
  - Required approvals
  - Pipeline requirements

Mapping Approach

TFS Role	GitLab Equivalent
Reader	Reporter
Contributor	Developer
Lead / Senior Dev	Maintainer
Admin	Owner

Branch Structure Considerations

Typical TFVC branching

Main, Dev branch, Feature branches rarely used, Release branches created late, Heavy merges

Git branching philosophy

- Small, short-lived feature branches
- Merge Requests (MR) enforce review
- Clear separation:
  - main → stable
  - develop (optional)
  - feature/
  - hotfix/
  - release/

Branch Structure Considerations

Actions during migration

- Identify old TFVC branches
- Keep only meaningful ones
- Decide on **GitLab Flow** or **GitFlow**
- Rebuild branches in Git, not replicate old TFVC structure as-is

Git &GitLab

- **Git** – is a source code versioning system that lets locally track changes and push or pull changes from remote resources.
  - A Git repository tracks and saves the history of all changes made to the files in a Git project.
  - Allow you to know who made what changes and when.
  - Allow you to revert any changes and go back to a previous state.
  - Allow for collaborative development.

https://git-scm.com

- **GitLab, GitHub, and Bitbucket** – are services that provide remote access to Git repositories.

What is GitLab?

- GitLab is a complete **DevOps platform** built around Git.
- Enables teams to **collaborate on code**, **build CI/CD pipelines**, **track issues**, and **manage deployments** in one application.
- Supports **both cloud-hosted (GitLab.com)** and **self-managed installations**.
- **Note:** GitLab ≠ Git — GitLab **uses Git** under the hood.

Git vs GitLab

Git	**GitLab**
Tracks code changes locally	A cloud platform for teams using Git
Command-line tool	Web-based (with Git support)
You work on your laptop	You push/pull your work to/from GitLab
No UI	Has full UI, CI/CD, issues, merge reqs

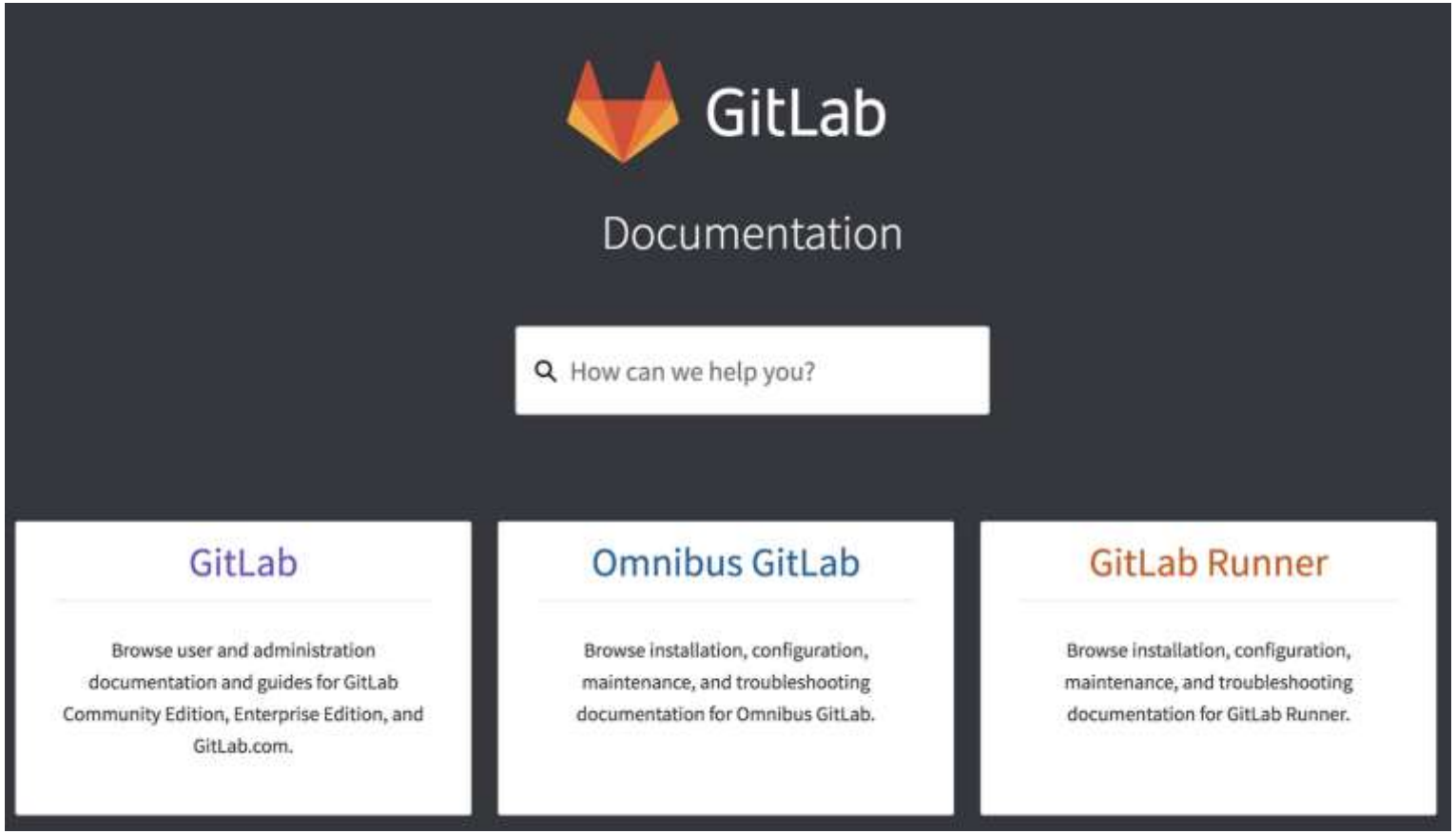
Year	Milestone
2011	Created by **Dmitriy Zaporozhets** as an open-source Git repository manager
2013	**GitLab Inc.** founded by **Sid Sijbrandij** to commercialize the platform
2015	**CI integrated** into GitLab — marked the beginning of the **single DevOps platform** vision
2017	**Auto DevOps** introduced with initial **Kubernetes integration**
2018	Launched **GitLab Ultimate** with enhanced **security** and **compliance** features
2021	GitLab **went public** on NASDAQ under the ticker **GTLB**
2023+	Added **AI-powered DevSecOps**, expanded **remote-first culture**, and achieved **global scale**

A single application for the entire DevOps lifecycle

Manage	Plan	Create	Verify	Package	Release	Configure	Monitor	Secure
Since 2016	Since 2011	Since 2011	Since 2012	Since 2016	Since 2016	Since 2018	Since 2016	Since 2017
Cycle Analytics	Issue Trackers	Source Code Management	Continuous Integration (CI)	Container Registry	Continuous Delivery (CD)	Auto DevOps	Metrics	SAST
DevOps Score	Issue Boards	Code Review	Unit Testing	Maven Repository	Pages	Kubernetes Configuration	Logging	DAST
Audit Management	Service Desk	Wiki	Integration Testing	Roadmap	Review apps	ChatOps	Cluster Monitoring	Dependency Scanning
Authetication and Authorization	Portfolio Management	Snippets	Code Quality		Incremental Rollout	Roadmap	Roadmap	Container Scanning
Roadmap	Roadmap	Web IDE	Performance Testing		Roadmap			License Management
		Roadmap	Roadmap					Roadmap

GitLab Documentation

docs.gitlab.com



GitLab Cloud vs Self-Managed

Feature	**GitLab Cloud (GitLab.com)**	**GitLab Self-Managed**
Hosting	Managed by GitLab Inc.	You manage on-prem or cloud
Maintenance	Handled by GitLab	Your responsibility
Customization	Limited	Full control
Infrastructure Cost	None	Your infrastructure
Setup Time	Instant	Medium to High
Use Case	Startups, SMBs	Enterprises, regulated orgs

GitLab vs GitHub vs Bitbucket

Feature	GitLab	GitHub	Bitbucket
CI/CD	✓ Built-in	✓ GitHub Actions	✓ Pipelines
Self-hosted Option	✓ Yes	⚠ Limited (GH Enterprise)	✓ Yes
DevOps Lifecycle	✓ Full stack	✗ Partial	✗ Partial
Permissions	Fine-grained	Good	Good
Boards & Issues	✓ Advanced	✓ Basic	✓ Good
Used by	Enterprises, DevOps	Open source, community	Atlassian shops

Key GitLab Features

Developer Tools

- Full Git repository management
- Merge Requests (MRs) with code reviews
- Built-in Web IDE for editing code directly in the browser

CI/CD Capabilities

- Native support for Continuous Integration and Deployment
- Easy-to-use .gitlab-ci.yml configuration file
- Pipeline visualization and multi-stage jobs
- GitLab Runners for automation

Project Management

- Issue tracking system with labels, assignees, and due dates
- Agile boards for visualizing work (Kanban-style)
- Milestones, epics, and roadmaps for planning

Security & Compliance

- Static Application Security Testing (SAST)
- Dynamic Application Security Testing (DAST)
- Dependency scanning and secret detection
- Built-in compliance reporting

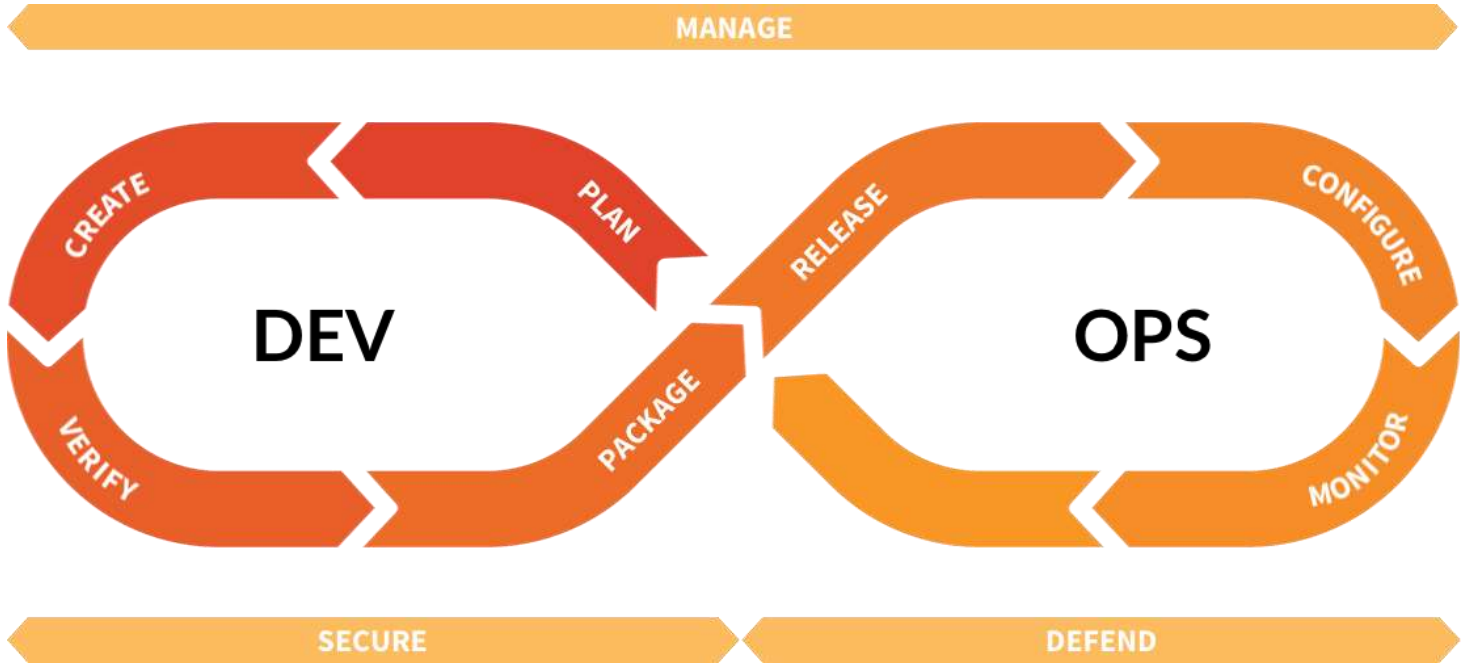
DevOps Integrations

- Seamless Kubernetes integration
- Package registries (Docker, Maven, NPM)
- Integrates with tools like Slack, Opsgenie, Prometheus, and more

GitLab in the DevOps Lifecycle

Plan → Code → Build → Test → Release → Deploy → Monitor → Secure

- GitLab supports the **entire DevOps lifecycle** in **one tool**, with seamless handoff between stages.



Lab 1

GitLab

What we'll see in the application

Git & Repository Management

Working with Git in GitLab and Following Modern Team Practices

Key Git terms

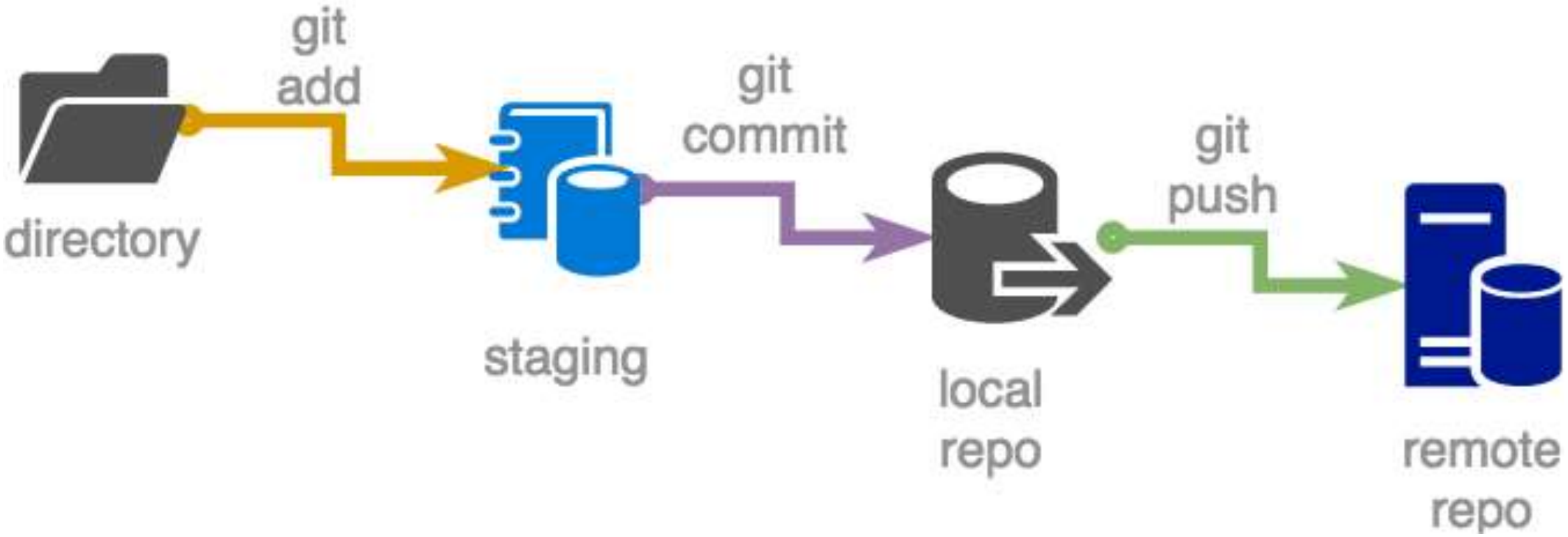
Local Repo	Local copy of the upstream repo
Remote (Upstream repo)	Hosted repository on a shared server (GitLab)
Untracked files	New files that Git has not been told to track previously
Working area	Files that have been modified but not committed
Staging area	Modified/Added files that are marked to go into the next commit

Key Git terms (cont.)

Branch	Independent line of development (typically scoped to a feature or bug fix)
Main (or master) Branch (trunk)	The canonical branch for the repository
Fork	A person copy of the ENTIRE repository
Merge	Bring two branches together

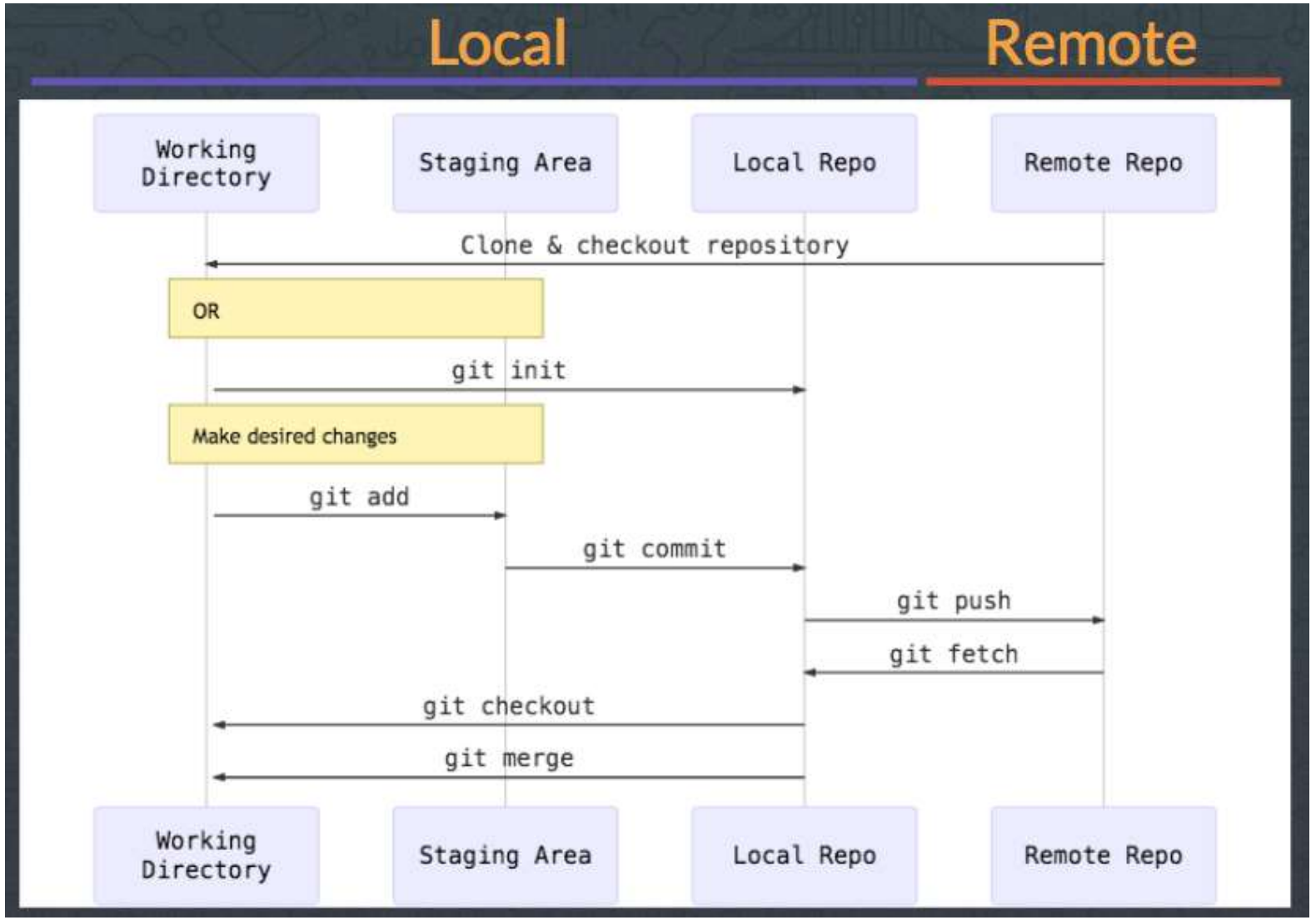
Local Repo consists of three **trees**

- A Working Directory with the actual files,
- the Staging Area and
- the last commit you made which is called HEAD



Let's put it all together!

Basic Git Workflow



Basic Git Workflow (Cont.)

Core Git Commands for Daily Use:

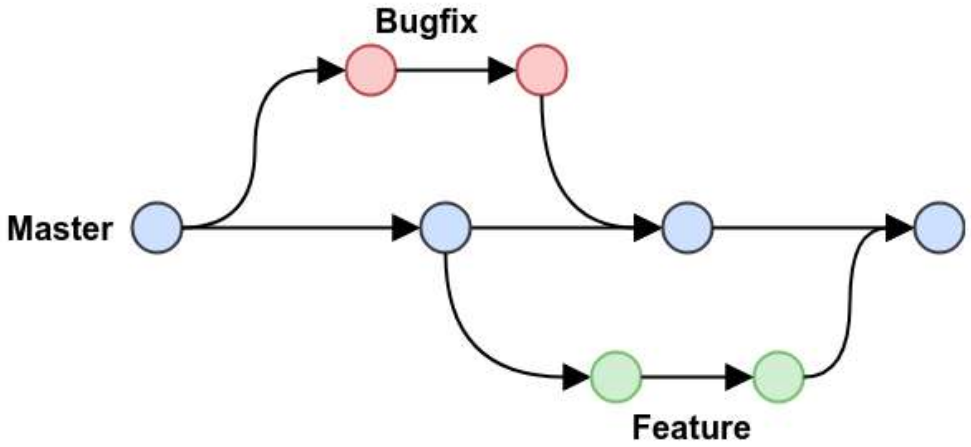
- git clone <repo-url> – Copy a remote repository to your local machine
- git add <file> – Stage file changes
- git commit -m "message" – Save staged changes locally
- git push – Upload local changes to the remote repository
- git pull – Fetch and merge changes from remote into local
- git branch <name> – Create a new branch
- git checkout <name> – Switch between branches


DAY 2

Git Branching Strategies

Why Branching Matters:

- Isolate features, bug fixes, and experiments
- Support parallel development
- Prevent instability in the main codebase



Git Branching Strategies (Cont.)

Common Branching Models:

- Choose the strategy based on your team size and deployment frequency.

Strategy	Description
Feature Branch	Each feature or fix has its own branch, merged when complete
Git Flow	Structured model with main, develop, feature/*, release/*, hotfix/* branches
Trunk-Based	Short-lived branches, frequent merges to main, fast delivery model

Merge Requests (MRs) and Code Reviews

- A **Merge Request** (MR) in GitLab is a way to propose changes from one branch to another.

- **It allows developers to:**
  - Compare code differences
  - Discuss changes with teammates
  - Run automated CI/CD pipelines
  - Get approvals before merging into protected branches (e.g., main or develop)

Key Components of an MR

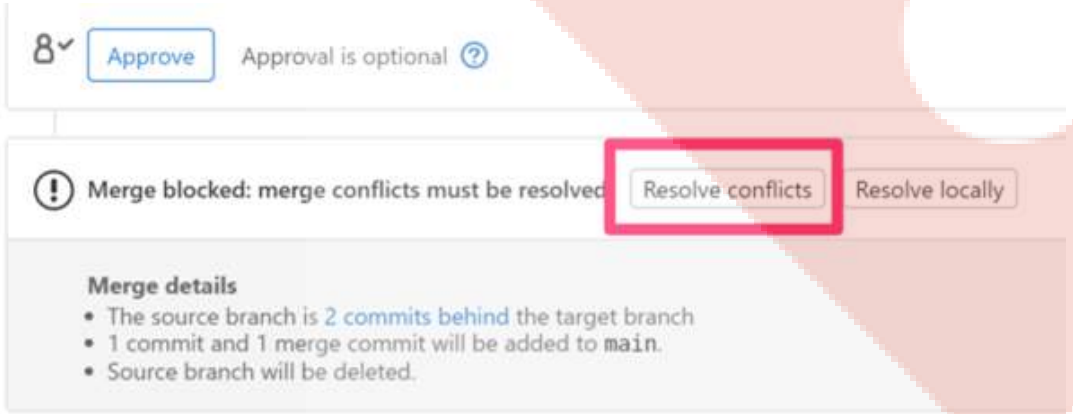
- Source branch: where changes are made (e.g., ***feature/login***)
- Target branch: usually **main, develop**, or a release branch
- Title & Description: explain what the MR does
- Assignees & Reviewers: people responsible for reviewing or approving the MR
- Approvals & Comments: inline code discussions and change suggestions
- Pipeline Status shows if CI jobs (tests, builds) have passed
- Merge Conditions: e.g., "only allow merge if pipeline passes and 1 approval received"

Benefits of Code Reviews via MRs

- Catch bugs early before they reach production
- Improve code quality and consistency
- Promote shared understanding and team learning
- Enable compliance with security and workflow standards

Merge Conflicts

- Conflicts happen frequently - if you don’t merge often enough
- Try to stay up-to-date with master
- Learning to fix merge conflicts can be difficult
- Be careful about force pushing after fixing conflicts

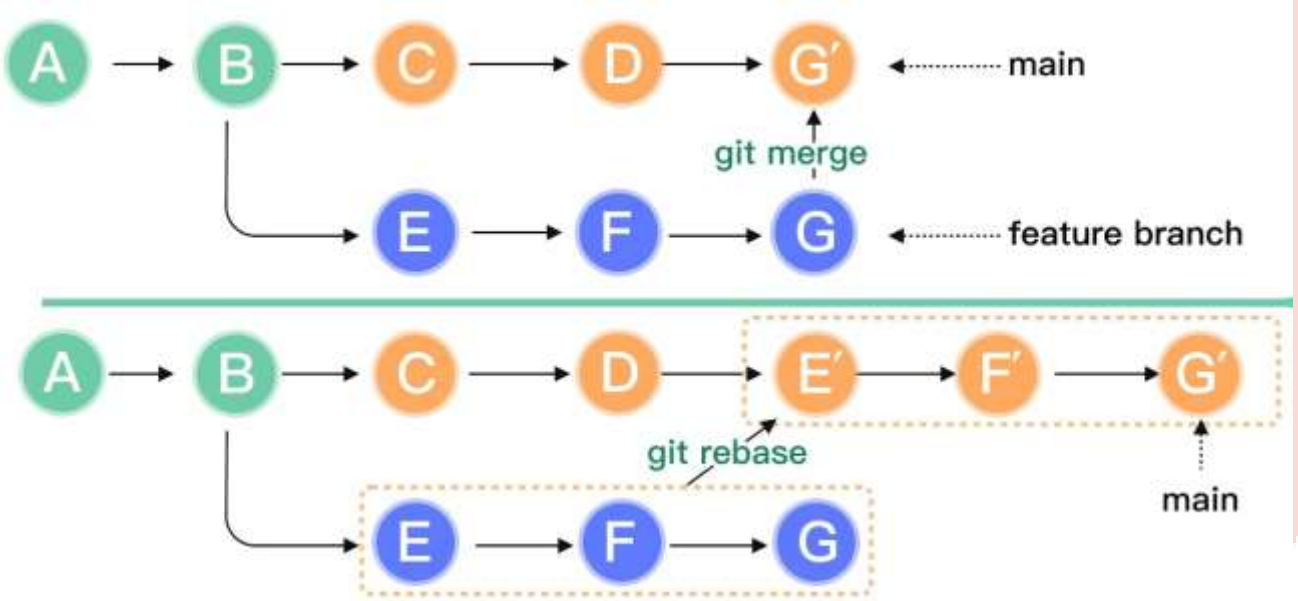


Best Practices for MRs

- Keep MRs small and focused
- Write clear MR titles and descriptions
- Tag the right reviewers
- Address review comments promptly
- Use GitLab’s "suggest changes" for cleaner collaboration

Rebasing vs. Merging

- Rebase when updating your branch from master
- Merge when bringing changes from feature branch into master



GitLab Project & Access Management

Organizing Projects, Managing Users, and Enabling Integrations

Creating Groups, Projects, and Repositories

Groups

- A **group** is a top-level container that holds projects and subgroups.
- Useful for team or department organization.
- Permissions and settings can be inherited by all contained projects.

Projects

- A **project** is a single Git repository with built-in issue tracking, CI/CD, wikis, and more.
- Projects live inside **groups** or under personal namespaces.

Repositories

- Git repositories store your code, tracked by Git versioning.
- Each project in GitLab is backed by a Git repository.

Cont.

- Example Structure:

```
TechCorp (Group)
├── web-team (Subgroup)
│   ├── frontend-site (Project)
│   │   └── Git repository: frontend-site.git
│   └── backend-api (Project)
│       └── Git repository: backend-api.git
├── devops-team (Subgroup)
│   └── ci-cd-templates (Project)
│       └── Git repository: ci-cd-templates.git
```

User Roles and Access Levels

- GitLab provides five default roles:

Role	Permissions Summary
Guest	View-only access to issues, public repo files
Reporter	Can view and download code, see pipelines, read-only access
Developer	Push to branches, create MRs, run pipelines
Maintainer	Merge code, manage issues, runners, and project settings
Owner	(Group-level only) Manage everything, including members/roles

Best Practice:
Use **least-privilege principle**: assign only the access level needed.

Public vs Private Projects

Project Visibility	Who Can View / Access?
Private	Only invited members
Internal (Self-Managed only)	Any logged-in GitLab user (not available on GitLab.com)
Public	Anyone can view project contents without authentication

Note: Even public projects can have private CI/CD variables or protected branches.

Webhooks and Integrations

- **What are Webhooks?**
  - A way to automatically send HTTP POST requests when certain events happen in your project.
  - **Examples:** Push events, merge requests, pipeline status changes

Common Use Case:

Notify Slack/Teams when:

- A new commit is pushed
- A merge request is opened
- A pipeline fails

Integrating with Slack (Example)

1. Go to Project > Settings > Integrations
2. Select Slack Notifications
3. Enter your Slack Webhook URL and channel
4. Choose events to trigger (e.g., push, MR, pipeline fail)
5. Save

Other Integration Options:

Microsoft Teams, Opsgenie, Jira, GitLab Webhooks with Jenkins or other CI tools

Summary

- **Groups** organize related projects and control access.
- Assign roles carefully using **GitLab’s permission model**.
- Control visibility with **private or public** project settings.
- Use **webhooks and built-in integrations** to automate notifications and workflow triggers.


DAY 3

Understanding GitLab CI/CD

Automating Software Delivery with Pipelines

What is GitLab CI/CD?

- GitLab CI/CD is GitLab's **built-in Continuous Integration and Continuous Deployment** system that automates the process of building, testing, and deploying code.

Key Features:

- Integrated directly into every GitLab project
- Triggered automatically by Git events (e.g., push, merge)
- Fully customizable with .gitlab-ci.yml
- Supports Docker, shell, Kubernetes runners, etc.

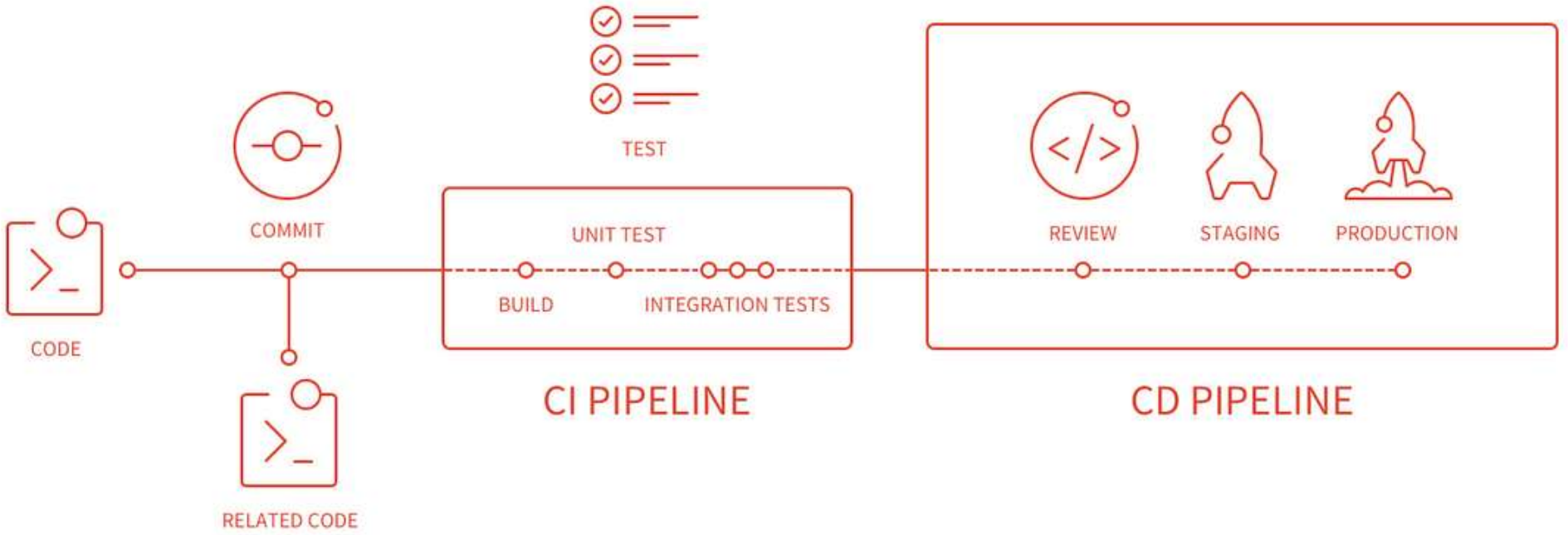
Benefits:

- Detect bugs early, Enforce consistent workflows, Automate deployments, Increase developer productivity

How GitLab CI/CD Works

Workflow:

- Developer pushes code to GitLab
- GitLab checks for a .gitlab-ci.yml file in the project root
- Pipelines defined in the YAML file are triggered automatically
- CI jobs (build/test/deploy) are run using GitLab Runners


TEST
COMMIT
UNIT TEST
REVIEW
STAGING
PRODUCTION
CODE
BUILD
INTEGRATION TESTS
CI PIPELINE
CD PIPELINE
RELATED CODE

Cont.

Terms to Know:

- **Pipeline:** A sequence of stages/jobs
- **Job:** A single task (e.g., run tests, build app)
- **Runner:** An agent that executes jobs

The .gitlab-ci.yml File

Purpose:

- Defines CI/CD workflow and instructions for GitLab to follow when a pipeline runs.

Key Sections:

- **stages:** Declares the pipeline steps (build, test, deploy)
- **jobs:** Scripts tied to stages (e.g., build-app, run-tests)

```yaml
# Define the stages of the pipeline
stages:
  - build
  - test
  - deploy

# First job: compiles or prepares the application
build-job:
  stage: build           # Assigned to the "build" stage
  script:                # List of shell commands to run
    - echo "Installing dependencies..."
    - echo "Compiling the app..."

# Second job: runs tests
test-job:
  stage: test            # Runs after the "build" stage
  script:
    - echo "Running unit tests..."
    - echo "Checking code quality..."

# Final job: deploys the application
deploy-job:
  stage: deploy          # Runs after "test" stage
  script:
    - echo "Deploying to staging environment..."
    - echo "Deployment successful."
```

Understanding Pipeline Stages

- **Stages define the flow** of your pipeline:

Stage	Purpose	Example Tasks
Build	Compile and package the code	Install dependencies, compile
Test	Verify the functionality	Run unit/integration tests
Deploy	Ship to staging/production	Upload to server, trigger Helm

Rules:
- Jobs in the same stage run in parallel
- Stages run in sequence

Best Practice: Keep stages small and focused.

Summary

- GitLab CI/CD automates the software lifecycle
- It runs pipelines based on the .gitlab-ci.yml file
- Pipelines consist of stages (e.g., build → test → deploy)
- Each job runs on a GitLab Runner
- Helps teams deliver faster and with higher quality

GitLab Runner Setup

Running Your Pipelines with Custom Executors

What is GitLab Runner?

- GitLab Runner is an **open-source application** used to **run CI/CD jobs** in GitLab pipelines.

Purpose:

- It’s the worker agent that executes jobs defined in your .gitlab-ci.yml file.

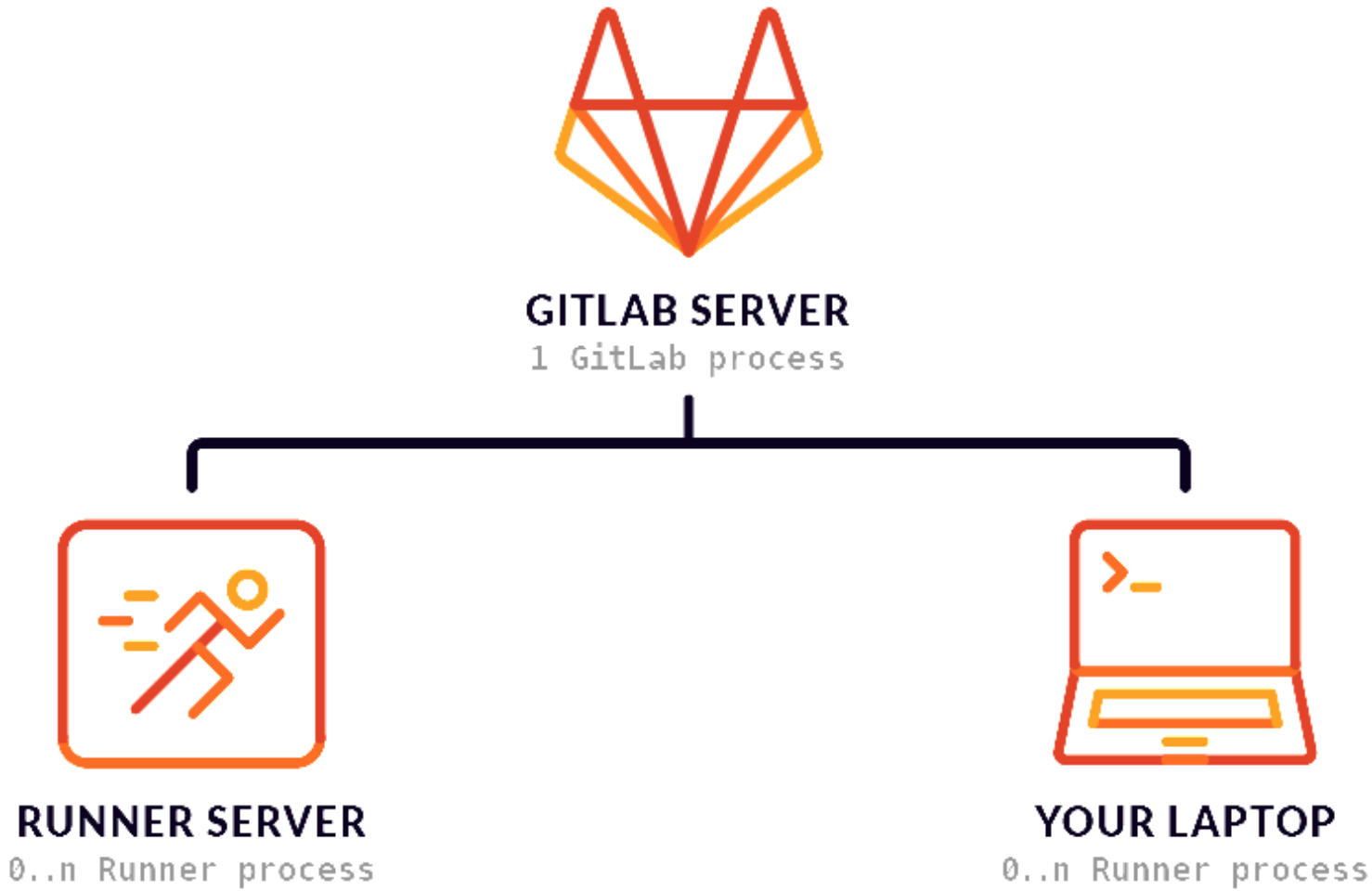
Key Points:

- Can be installed on any platform (Linux, Windows, Mac, Docker)
- Executes jobs either locally or in containers (Docker, Kubernetes)
- Supports shared runners (global) and specific runners (project-level)

Architecture:

- GitLab Server <-- triggers --> GitLab Runner <-- executes --> Your build/test/deploy scripts

What is GitLab Runner?



Install GitLab Runner

Registering the Runner with a Project

- Command to Register: *gitlab-runner register*

- Interactive Setup Prompts:
  - GitLab URL: https://gitlab.com or your GitLab instance
  - Token: Found in GitLab → Settings → CI/CD → Runners
  - Description: e.g., ubuntu-runner-01
  - Tags: Optional labels for job selection
  - Executor: Choose from Shell, Docker, Kubernetes, etc.

- Tip: Use different tags for specific runners and reference them in .gitlab-ci.yml

https://docs.gitlab.com/runner/register/

Runner Executor Types

Executor	Description	Use Case
Shell	Executes jobs in the shell of the host OS	Simple environments (VMs, servers)
Docker	Runs jobs inside Docker containers	Most flexible, clean environment
Docker+Machine	Auto-spins up Docker machines on demand	Scalable CI in the cloud
Kubernetes	Uses Kubernetes pods for job execution	Cloud-native CI/CD
Custom	Plug in your own logic or infrastructure	Advanced enterprise setups

Summary

- GitLab Runner executes your .gitlab-ci.yml jobs
- It can be installed on Linux, Windows, or as a Docker container
- Runners must be registered with your GitLab project or group
- Multiple executor types give flexibility depending on your infrastructure

Advanced GitLab CI/CD

Secure, Performant, and Flexible Pipelines

Secrets and CI/CD Variables

What are CI/CD Variables?

- Key-value pairs used to store **configurable or sensitive data** in your pipeline
- Examples: API keys, deployment paths, Docker usernames/passwords

- **Types of Variables:**

Type	Scope	Example
Project Variables	Per-project	PROD_DB_PASSWORD
Group Variables	Across projects	SLACK_WEBHOOK_URL
Masked & Protected	Secure	Values hidden in logs and restricted to protected branches

Cont.

- How to use in .gitlab-ci.yml:

```
script:
  - echo "Deploying with token: $DEPLOY_TOKEN"
```

Best Practices:

- Mark secrets as **masked**
- Avoid hardcoding secrets into your YAML
- Use **protected variables** for production-only branches

Artifacts and Caching

Artifacts

- Files or directories **saved after a job finishes**
- Used to **pass results to later jobs** (e.g., compiled code, test reports)

```yaml
build-job:
  stage: build
  script:
    - npm run build
  artifacts:
    paths:
      - dist/
    expire_in: 1 hour
```

Artifacts and Caching Cont.

Caching

- Speeds up jobs by **reusing previously downloaded dependencies**
- Cache is **shared between jobs/stages**

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

Artifacts and Caching Cont.

- **Key Difference:**

Feature	Purpose	Scope
Artifacts	Share files between jobs	Pipeline
Cache	Speed up repeated work	Runner-wide

Conditional Execution

Conditional execution allows you to **control whether a job runs** based on **specific conditions** such as:

- The branch name
- The type of pipeline (push, merge request, schedule)
- The presence or value of a variable

Conditional Execution Cont.

Recommended Method: rules

Use the **rules**: keyword to define when a job should run, be skipped, or require manual approval.

```yaml
deploy-prod:
  stage: deploy
  script:
    - ./deploy.sh
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      when: always
```

This job runs only when a commit is pushed to the main branch.

Conditional Execution Cont.

Common Conditions You Can Use

Condition	Description
$CI_COMMIT_BRANCH == "main"	Only run for a specific branch
$CI_COMMIT_TAG	Only run when a tag is pushed
$CI_PIPELINE_SOURCE == "schedule"	Run only for scheduled pipelines
$MY_CUSTOM_VAR == "true"	Run based on custom variable value

Conditional Execution Cont.

Best Practices:

- Prefer rules: over the old only: / except:
- Use manual jobs for sensitive steps (like production deployment)
- Combine multiple conditions using multiple rules

Matrix Builds & Parallel Jobs

A **Matrix Build** runs the **same job multiple times** with **different sets of variables**.

This is useful for:

- Testing your app on multiple environments (e.g., OS, Node versions)
- Avoiding copy-pasting multiple jobs for each combination

Matrix Builds & Parallel Jobs Cont.

Example: Matrix Build for OS and Node Versions

```yaml
test:
  stage: test
  script:
    - echo "Testing on $OS with Node $VERSION"
  parallel:
    matrix:
      - OS: ["ubuntu", "alpine"]
        VERSION: ["14", "16"]
```

Result:

This will create 4 jobs:

- ubuntu + node14
- ubuntu + node16
- alpine + node14
- alpine + node16

Each job runs **in parallel**, with its own OS and Node version combination.

Matrix Builds & Parallel Jobs Cont.

Parallel jobs let you **split a single job** into multiple **identical instances** to save time.

Use it for:

- Splitting long-running tests across multiple runners
- Load testing
- Processing data chunks

Matrix Builds & Parallel Jobs Cont.

Example: Run the same job 3 times in parallel

```yaml
load-test:
  stage: test
  script:
    - ./run-load-test.sh
  parallel: 3
```

Result:

Three identical jobs will run at the same time, each independently executing the same script.

Matrix vs Parallel – Key Difference

Feature	Matrix Build	Parallel Job
Purpose	Run job with different variables	Run same job multiple times
Custom logic	Yes – values change per job	No – logic is identical
Use case	Test multiple OS/language versions	Speed up job or simulate load

- Use matrix builds for environment/version testing
- Use parallel jobs to speed up test execution
- Combine both for advanced CI/CD pipelines

Summary

- Use CI/CD variables for secrets and environment configuration
- Artifacts transfer files between jobs; caching accelerates builds
- Use rules for conditional execution of jobs
- Matrix and parallel jobs help you test faster across many configs

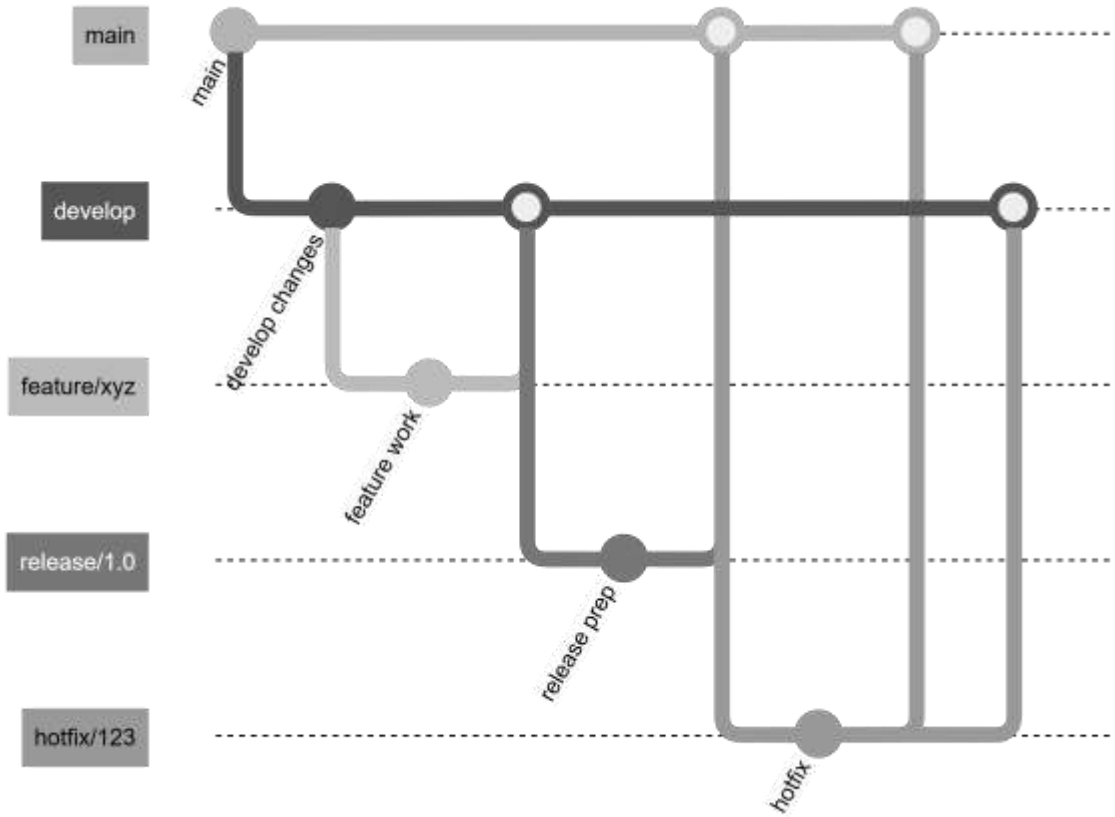

DAY 4

What is GitFlow?

GitFlow is a **multi-branch** model with strict separation of development stages.

Core branches:

- main (production-ready code)
- develop (integration branch)
- feature/*
- release/*
- hotfix/*



GitFlow

Strengths

- Highly structured
- Good for large applications
- Clear separation between development & release preparation

Weaknesses

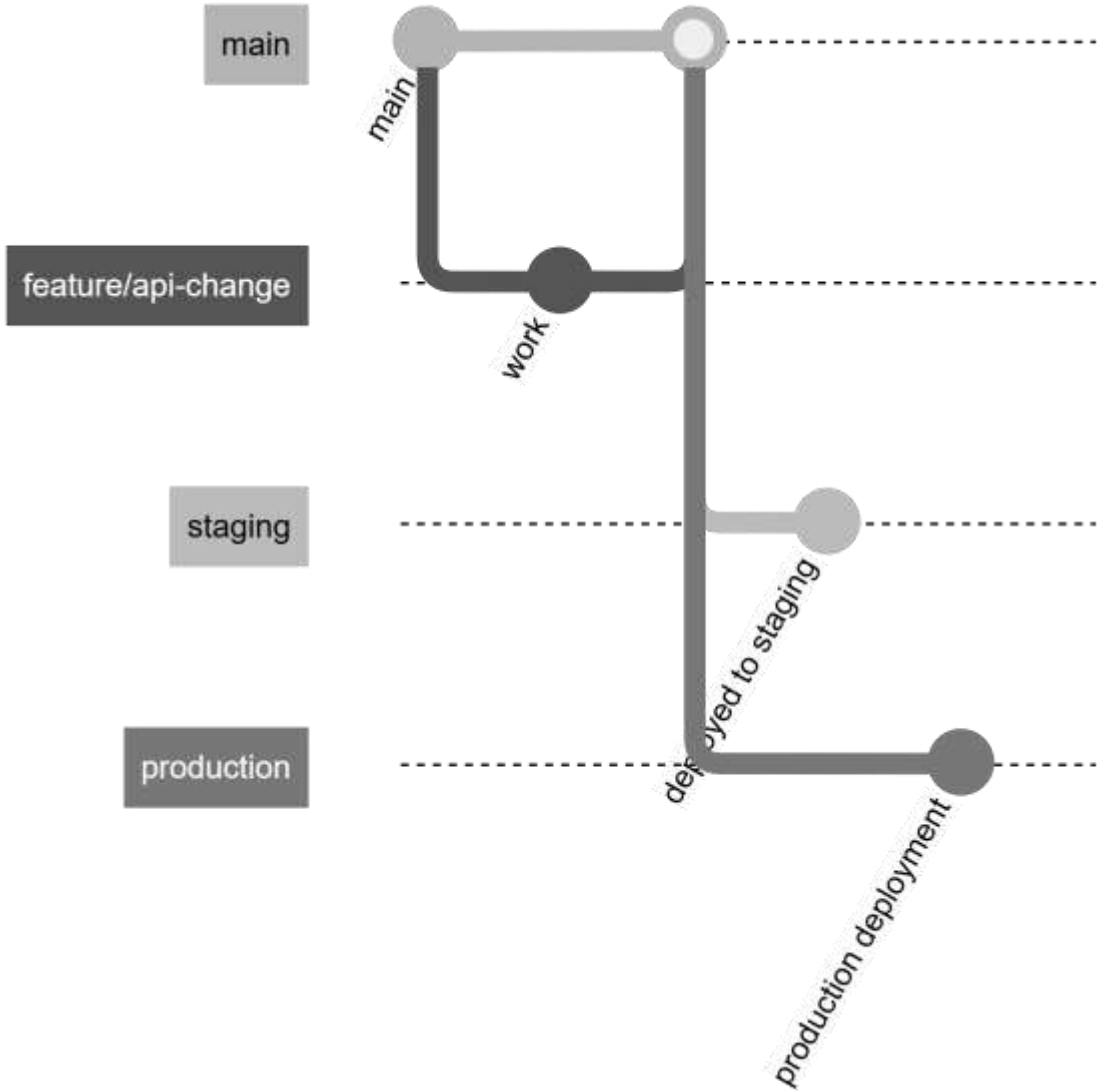
- Too many long-lived branches
- Slower cycle time
- Overkill for teams practicing continuous delivery
- Merge churn increases complexity in CI/CD

What is GitLab Flow?

GitLab Flow is a **simpler, more modern approach** combining feature branches + environment branches.

Core Model

- main = source of truth
- feature/* → merge via MR
- Optional branches per environment:
  - production
  - staging
  - pre-production



GitLab Flow

Strengths

- Simple branching → fewer merges
- Works perfectly with GitLab CI/CD
- Faster delivery cycle
- Clear mapping between branches & environments
- Ideal for cloud-native & containerized .NET apps

Weaknesses

- Requires disciplined MR usage
- Less suited for heavily manual release processes

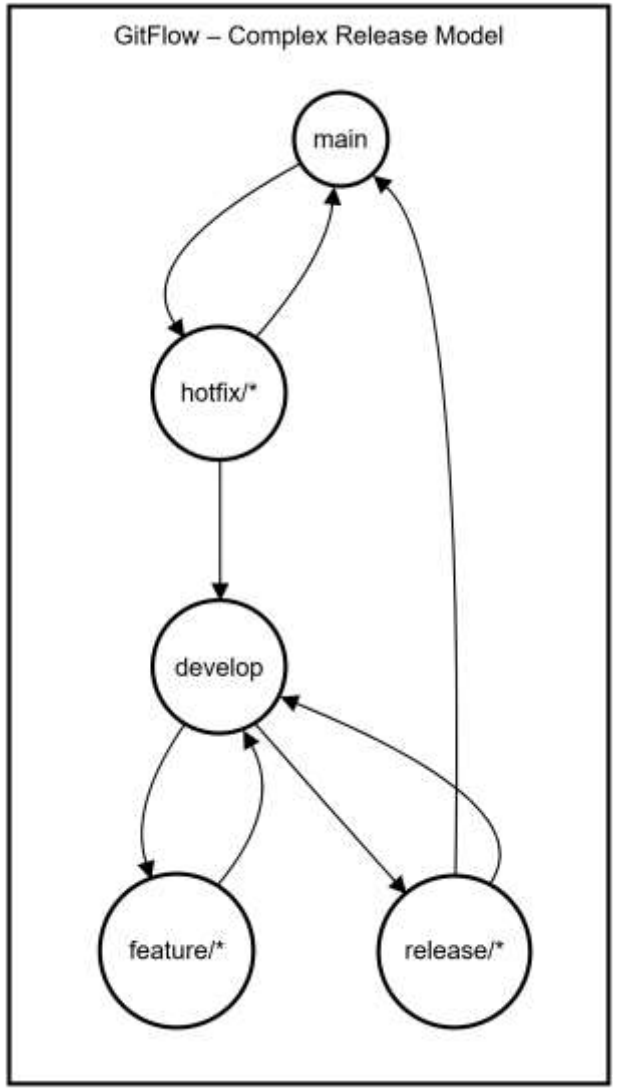
GitFlow vs GitLab Flow

Criteria	GitFlow	GitLab Flow
Complexity	High (many branches)	Low (simple model)
Release Model	Suited for scheduled releases	Suited for continuous delivery
CI/CD Fit	Requires extra automation	Designed for GitLab CI/CD
Team Size	Large teams with heavy processes	Any size, especially agile teams
Branch Lifetime	Long-lived branches	Mostly short-lived feature branches
Hotfix Handling	Dedicated hotfix branch	Easy: branch from main
Suitable For .NET monoliths?	Yes	Yes
Suitable For microservices?	No	Best choice

GitFlow

GitFlow separates all stages of development into distinct branches:

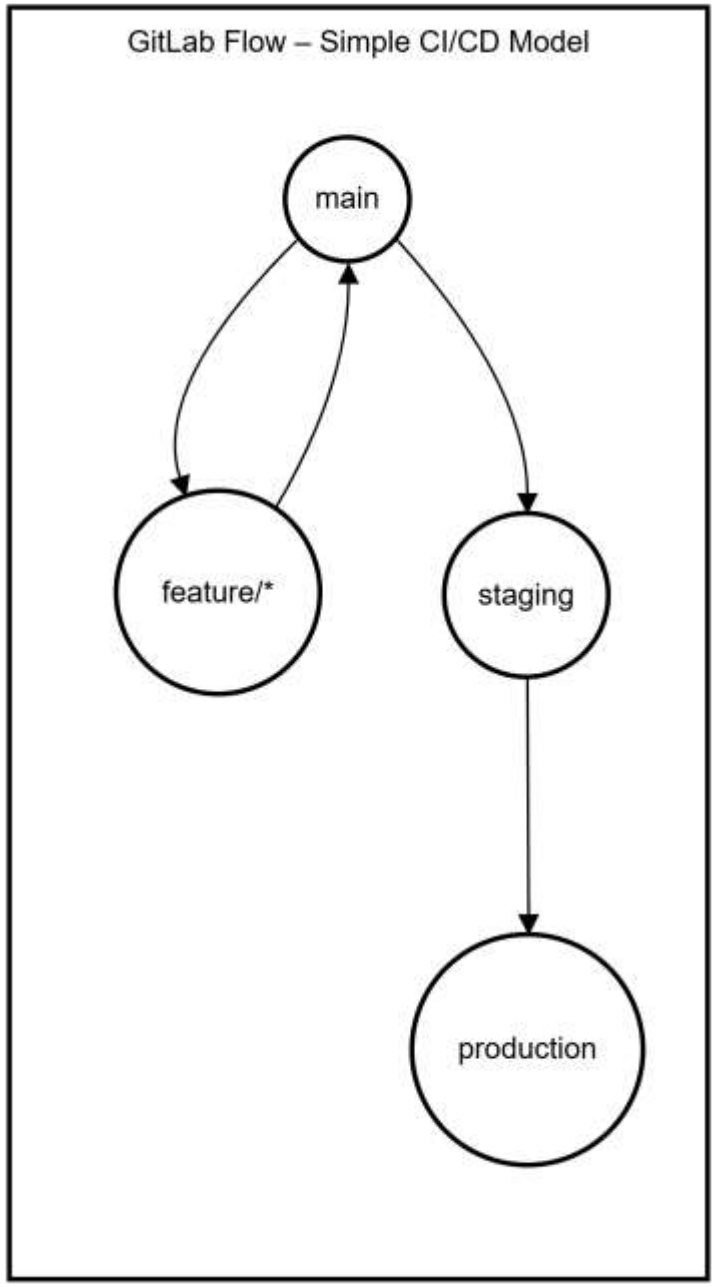
- **main** → always production-ready
- **develop** → integration branch where features are combined
- **feature/*** → short-lived branches per feature
- **release/*** → stabilization before production
- **hotfix/*** → urgent production fixes



GitLab Flow

GitLab Flow streamlines the branching model to reflect modern DevOps practices:

- **main** → single source of truth
- **feature/*** → short-lived work branches
- Optional environment branches (e.g., **staging**, **production**)



What TFS Check-In Policies Used to Do

- **Enforced rules before allowing check-in:**
  - Work item required
  - Code comment required
  - Tests must pass
  - Code analysis must run
  - Review required (optional)

- Policies were client-side rules inside Visual Studio

What GitLab Does Instead

- GitLab replaces **check-in policies** with **Merge Request (MR) Templates + CI + Branch Protections**

- MR Templates define **what every developer must include** before merging:
  - Linked issue
  - Description of change
  - Test evidence
  - Architecture/security notes
  - Impact analysis
  - Checklist for reviewers

Cont.

- Branch protection rules enforce:
  - Who can merge
  - Required approvals
  - CI pipeline must pass
  - No direct pushes to protected branches (like main)
- **Result: Same governance as TFS, but more flexible and automated.**

Example MR Template (GitLab)

```
## Summary
Describe the purpose of this change

## Linked Issue
Closes #ISSUE_ID

## Changes
- What was changed?
- Why was it needed?

## Testing
- [ ] Unit tests added/updated
- [ ] Manual testing done
- [ ] CI pipeline passed

## Impact
- Does this change affect other services?
- Breaking changes?

## Checklist
- [ ] Code reviewed
- [ ] Code follows conventions
- [ ] Security considerations addressed
```

SonarQube

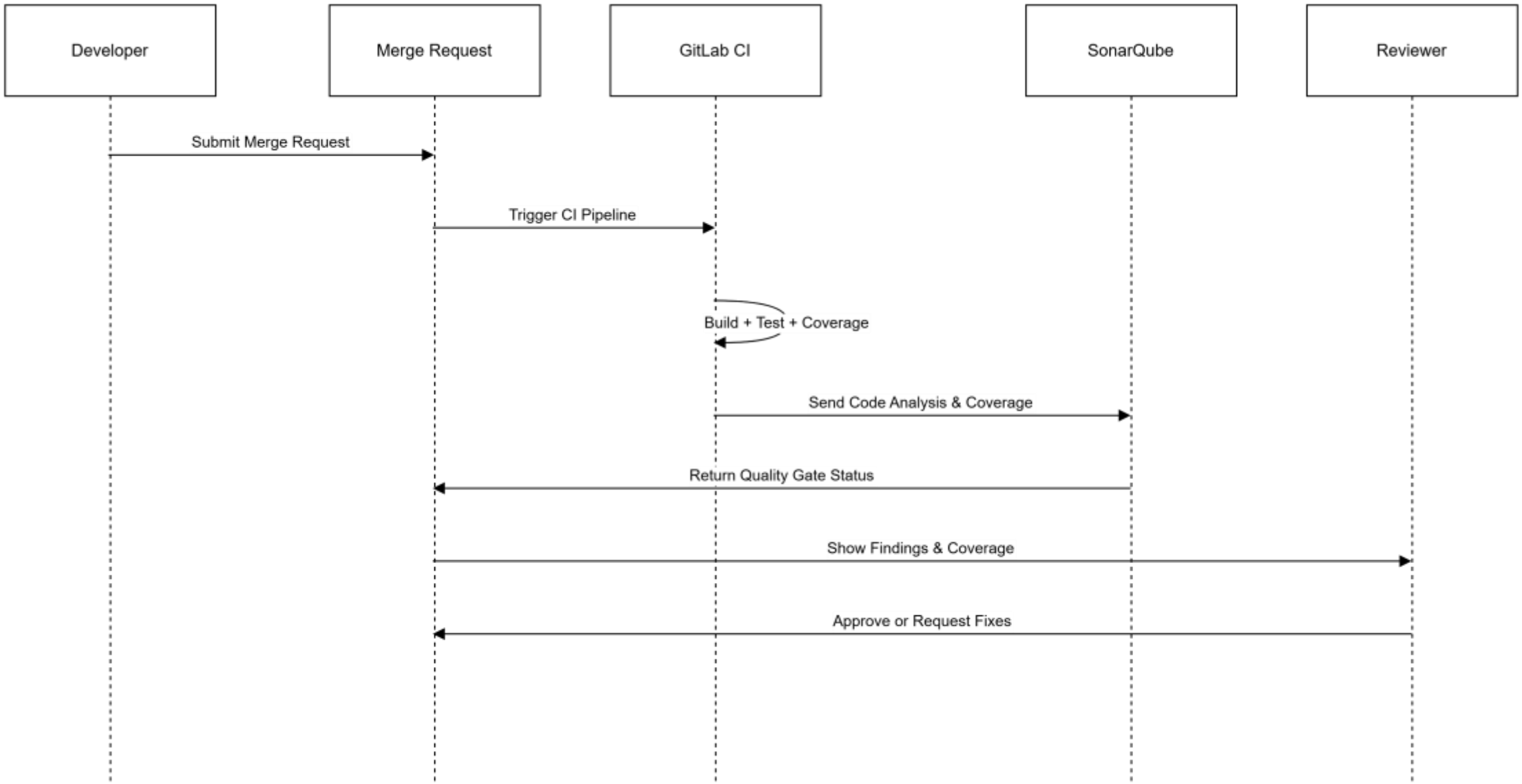
SonarQube is an automated code quality and security platform that analyzes source code to find:

- Bugs
- Security vulnerabilities (SAST)
- Code smells (bad practices)
- Duplicate code
- Maintainability issues
- Test coverage gaps

- It acts as a **“quality gate”** in your CI/CD pipeline meaning code cannot be merged unless it meets required standards.
- **SonarQube = Automated code reviewer + security scanner + quality gate.**

What SonarQube Checks in a .NET Project

- **SonarQube automatically scans:**
  - C# code quality & styling
  - Security vulnerabilities (SAST)
  - Code smells
  - Duplications
  - Unit test coverage
  - Roslyn analyzer results
  - Cyclomatic complexity
  - Deprecated API usage
- **For .NET Teams:**
  - SonarQube integrates fully with:
  - dotnet build
  - dotnet test
  - .csproj file metadata
  - GitLab Merge Requests


Developer
Merge Request
GitLab CI
SonarQube
Reviewer
Submit Merge Request
Trigger CI Pipeline
Build + Test + Coverage
Send Code Analysis & Coverage
Return Quality Gate Status
Show Findings & Coverage
Approve or Request Fixes

GitLab UI: What Developers See in Merge Requests

Developers see:

- SonarQube summary directly in MR
- Coverage % for changed files
- New bugs, vulnerabilities, or code smells
- Line-by-line annotations in MR diffs
- “Quality Gate Passed” or “Failed” badge

Outcome

- Issues are caught early
- Code reviewers get better context
- Safer and more stable deployments


Thank
you