Day 1: Introduction to DevOps & Version Control

Welcome to a practical, theory-first introduction designed for developers ready to understand the DevOps mindset and master Git fundamentals. This one-day journey will transform how you think about software delivery and collaboration.

What is DevOps?

The Core Philosophy

DevOps fundamentally combines **Development + Operations** into a unified approach that emphasises collaboration, automation, and delivery speed. It breaks down traditional silos between writing code, deploying it, and maintaining systems in production.

The ultimate aim? Deliver reliable software **faster and continuously**, responding to user needs with agility and confidence.

Why DevOps Matters

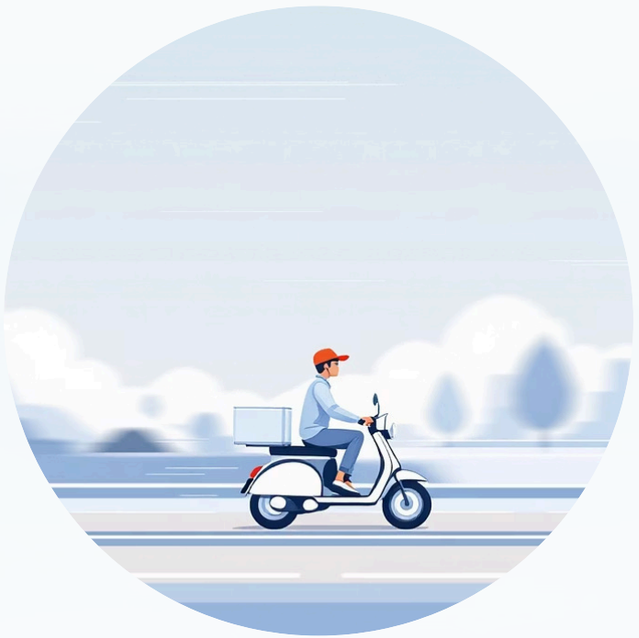
The Old Model

Manual processes, slow release cycles, and error-prone deployments created bottlenecks and frustration across teams.

The DevOps Way

Automation, continuous feedback loops, and shared responsibility transform how software reaches users.

Key Benefits



Faster Delivery

Ship features and fixes to production in hours, not weeks

Stable Deployments

Reduce downtime and rollback incidents through automation

Better Quality

Continuous feedback catches issues early in the development cycle

Agile vs DevOps

Whilst both methodologies revolutionise software development, they focus on different aspects of the delivery pipeline. Understanding their relationship helps clarify DevOps' unique value.

Agile

- Iterative development process
- Focus: speed of development
- Tools: Jira, Scrum boards
- Output: working code

DevOps

- End-to-end automation
- Focus: delivery & operations
- Tools: Jenkins, Docker, Git
- Output: working code in production

Think of Agile as perfecting *how you build*, whilst DevOps perfects *how you deliver and maintain* what you've built.

The DevOps Lifecycle

Plan

Define requirements and strategy

Code

Write and version your software

Deploy

Push to production environments

Build

Compile and package applications

Release

Prepare for deployment

Test

Validate functionality and quality

The Core Principle

This continuous loop of feedback and improvement represents the heart of DevOps. Each phase informs the next, creating a virtuous cycle.

The Goal

Automate everything possible to achieve faster cycle times and dramatically fewer failures in production.

Git & Version Control Overview

Version control systems meticulously track changes made to source code and other digital assets over time, providing a comprehensive history of every modification.

Git is the industry-standard distributed version control system, indispensable for almost all modern software development teams due to its speed, efficiency, and robust data integrity features.

Comprehensive Tracking

Every change is recorded, providing a complete, auditable history of the codebase, essential for debugging and understanding evolution.

Seamless Collaboration

Developers can work simultaneously on different features or fixes without the risk of overwriting each other's contributions.

Safe Experimentation

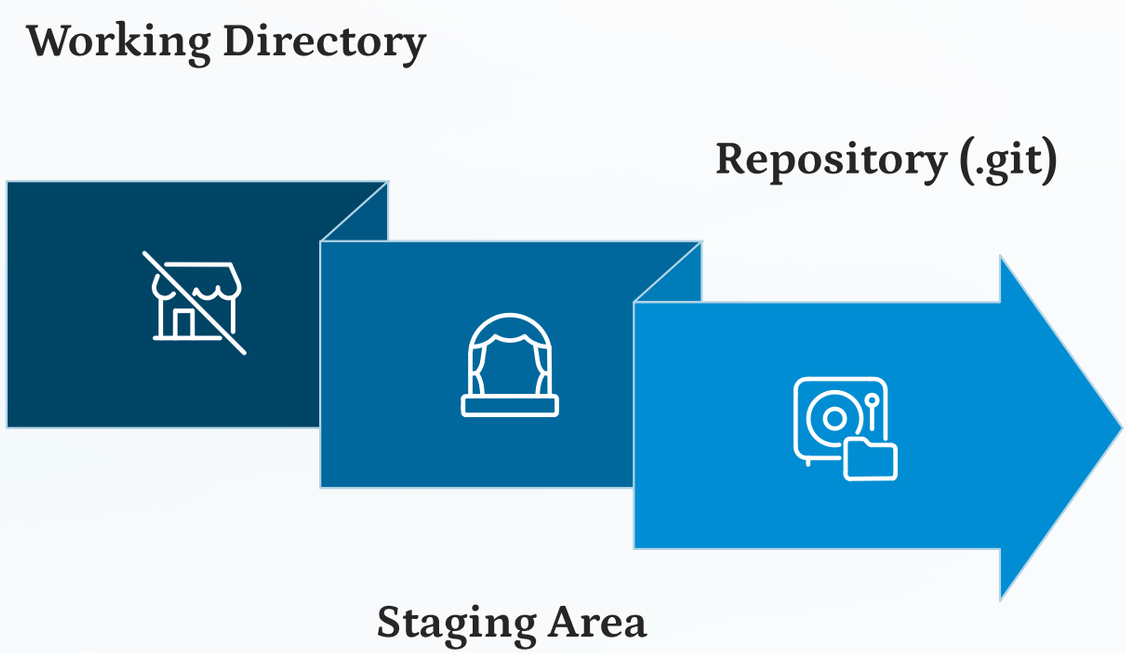
Easily create branches for new features or experiments and merge them back, or revert to previous states if issues arise.

CI/CD Integration

Git integrates deeply with Continuous Integration and Continuous Delivery (CI/CD) tools like Jenkins and GitHub Actions, automating build and deployment pipelines.

Git Architecture

Git's core principle revolves around storing content as a series of complete **snapshots** of your project, rather than just recording changes (diffs) between files. This design ensures data integrity and efficiency.



Understanding these distinct areas is fundamental to effectively managing your code and collaborating with others in Git.

Git Installation & Setup

Get Git up and running on your system with these essential installation and configuration steps.

Installation

```
sudo apt install git       # Linux
git --version              # Verify installation
```

Configuration

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

Always configure your identity before your first commit — it’s critical for traceability in shared repositories.

Create & Initialize a Repository

To begin version controlling your project, you'll either start a new local Git repository or download an existing one.

Option 1: New Local Repository

Use these commands to initialise a brand new Git repository in your project's directory.

```
mkdir devops-demo
cd devops-demo
git init
```

Option 2: Clone an Existing Repository

Clone a complete copy of a repository from a remote server, such as GitHub.

```
git clone https://github.com/org/repo.git
cd repo
```

Regardless of the method chosen, your project directory will now feature a hidden `.git` folder – this directory is the core of your Git repository, containing all the necessary version control data.

Git Status & Staging

Before you commit your changes, it's crucial to understand the current state of your working directory and what changes are ready to be saved.

Git provides essential commands to inspect these states:

```
git status          # View tracked/untracked files
git add file.txt    # Add specific file to the staging area
git add .           # Add all changes in the current directory to the staging area
```

Pro Tip: Always check `git status` before committing — it helps avoid unintentionally including temporary or debug files in your commit.

Committing Changes

Once you've staged your changes, a commit acts as a snapshot of your project at a specific point in time. It's how you permanently record your modifications in the local repository.

```
git commit -m "Add Docker setup documentation"
```

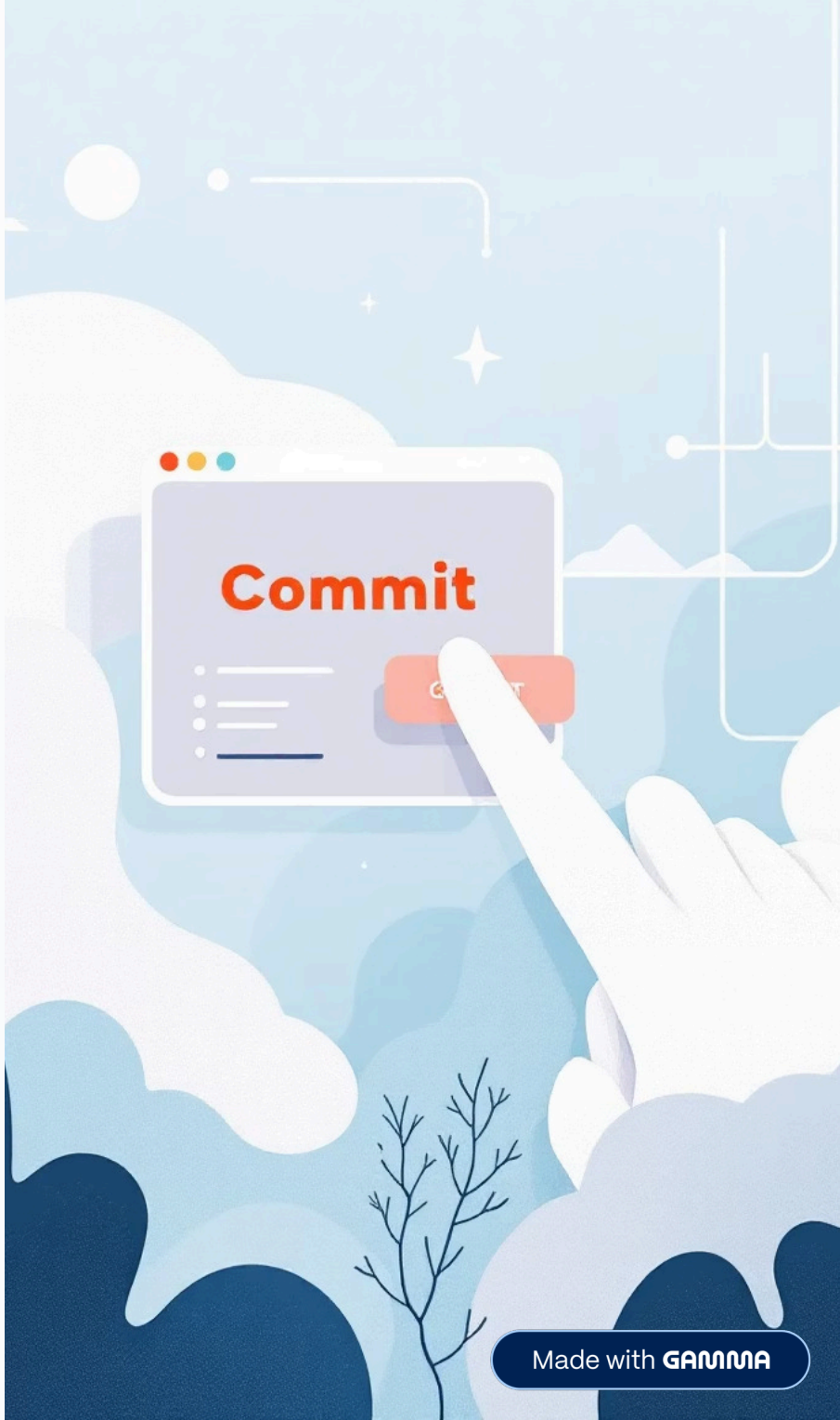
Best Practices for Commit Messages

Clear and descriptive commit messages are crucial for understanding project history and collaborating effectively.

- Use clear, concise messages that summarise the change.
- Prefix your messages with a type (e.g., `feat` for a new feature, `fix` for a bug fix, `docs` for documentation, `chore` for maintenance).

For example:

```
feat: add Jenkins pipeline section
fix: correct docker-compose file
```



Viewing Commit History

Understanding the history of your project is fundamental for collaboration and debugging. Git provides powerful tools to navigate through every change ever made.

The Project's Evolution: `git log`

```
git log --oneline --graph --decorate
```

This command offers a condensed, graphical view of your commit history, showing branches, merges, and the relationships between commits at a glance. It's an indispensable tool for visualising the project's journey.

Inspecting Changes: `git diff` & `git show`

```
git diff              # See unstaged changes
git diff --staged     # See staged changes
git show <commit_id>  # Inspect a specific commit's changes and metadata
```

Before committing, `git diff` allows you to review modifications, ensuring only intended changes are staged. Once committed, `git show` lets you examine the full details of any historical commit, including its metadata and the exact changes introduced.

> Viewing commit history is incredibly useful for debugging. If something breaks after a deployment, you can quickly trace back to the problematic commit and identify the cause.

Branching

Branches allow developers to isolate new features or bug fixes from the main codebase, enabling parallel development without disrupting the stable version of the project.

```
git branch feature/docker-setup
git checkout feature/docker-setup
```

For efficiency, you can create and switch to a new branch in a single command:

```
git checkout -b feature/docker-setup
```

Each branch represents an independent line of development, allowing for safe experimentation and collaborative work on different features simultaneously.

Merging Branches

Once you've completed and thoroughly tested your changes on a separate branch, the next crucial step is to integrate them back into the main development line. This process combines the histories of the two branches.

```
git checkout main
git merge feature/docker-setup
```

During a merge, Git attempts to combine changes automatically. However, if identical lines of code have been modified differently in both branches, Git will highlight these as **merge conflicts**. You must manually resolve these conflicts within the affected files before committing the final merged state.

Pro Tip: Avoid long-lived branches. Merging frequently reduces the likelihood and complexity of conflicts, making integration smoother and faster.

Working with Remote Repositories

Remote repositories are shared versions of your project hosted on the internet or network, enabling collaboration and secure backups. This section covers how to link your local repository to a remote one and common synchronisation operations.

Connecting Your Local Repository

Link your local repository to a remote server, such as GitHub, and push your initial changes to establish the connection.

```
git remote add origin https://github.com/your/repo.git
git push -u origin main
```

Common Synchronisation Operations

Keep your local repository up-to-date with changes from the remote and share your own modifications with the team.

```
git pull      # Fetch + merge latest changes
git fetch     # Only download changes, don't merge
```

Pro Tip: Regularly push your local changes to the remote repository. This acts as a crucial backup and keeps your team updated with your progress, preventing data loss and simplifying collaboration.

Collaborating via Pull Requests

Pull Requests (PRs) are a core mechanism for collaborative development in Git, especially when using platforms like GitHub or GitLab. They facilitate code review, discussion, and integration of changes.

1

Push Feature Branch

Upload your completed feature branch to the remote repository, making it accessible to others.

2

Create a Pull Request

Initiate a PR on your chosen platform (e.g., GitHub) to propose your changes for integration into the main branch.

3

Request Code Review

Notify teammates to review your code, provide constructive feedback, and suggest improvements.

4

Merge After Approval

Once your changes have been reviewed and approved, merge them into the target branch, typically 'main' or 'develop'.

Pull Requests are vital for maintaining code quality and project stability. They ensure **peer review**, provide **traceability** for all changes, and often trigger **Continuous Integration (CI)** checks to validate code before it's merged into the main codebase.

Git Ignore & Clean Working Tree

The `.gitignore` file tells Git to ignore specified files and directories, preventing temporary files, build artefacts, or sensitive configurations from being accidentally committed.

```
# .gitignore example
node_modules/
.env
build/
```

To tidy your working directory by removing untracked files that are no longer needed, use the `git clean` command.

```
git clean -fd
```

The `-f` (force) option is required to delete files, and `-d` allows the removal of untracked directories. Always review untracked files with `git status` before running `git clean`.

Real-world DevOps Integration

Git is the foundational element of any robust DevOps pipeline, enabling seamless collaboration, automation, and end-to-end traceability throughout the software development lifecycle.

1

CI/CD Trigger

Every `git push` can trigger a Continuous Integration (CI) build in tools like Jenkins, ensuring immediate validation and feedback.

2

Release Markers

Git tags are crucial for explicitly marking stable points in the repository's history, typically used to identify official software releases.

3

Branching Strategy

- `main`: Production deployments
- `dev`: Staging and testing
- `feature/*`: Active development

This integrated approach facilitates highly traceable, automated deployments, enhancing efficiency and reliability from development to production.

Recap & Expected Outcomes

By the end of Day 1, developers should be able to:

1. **Explain DevOps culture and lifecycle.**
2. **Use Git confidently for version control.**
3. **Collaborate effectively using branches and Pull Requests.**
4. **Prepare for Continuous Integration/Continuous Delivery (CI/CD) integration on Day 2.**

Next Step

Transition from version control to Continuous Integration using Jenkins, setting the stage for automated build and test processes.