==========

Git — From Basics to Advanced

A Complete Guide with Deep Dives, Examples & CLI Cheat Sheets

==========

Author : Ram Nimse

Published : August 2025

Contact : ramnimse92@gmail.com

Version : 1.0.0

Sections Covered

1. **Git Basics — Fundamental Concepts with Examples.**
2. **Git Intermediate Concepts — Deep Dive with Examples.**
3. **Git Advanced Concepts — Detailed Deep Dive.**
4. **Git Scenario-Based Usage — Real-World Problems & Solutions.**
5. **Git CLI Commands Cheat Sheet — From Basic to Advanced**

Git Notes for DevOps Engineers – Basic (Deep Explanation)

1. What is Git?

1. Git is a **Distributed Version Control System** (DVCS).
2. Unlike older systems (like SVN) that have one central server, Git gives each developer a full copy of the project history.
3. This means:
   - You can work offline.
   - You can experiment without breaking the main code.
   - You can roll back to any version if needed.

Example (DevOps Use Case):
Imagine your team is maintaining Terraform scripts for AWS. If a new change breaks deployment, you can quickly roll back to a stable commit using Git.

2. Why DevOps Engineers Need Git?

- **Collaboration** → Multiple team members can update Kubernetes manifests simultaneously.
- **Traceability** → Each commit has an author and timestamp → "Who changed what, and why?"
- **Rollback** → If a wrong Helm chart is deployed, you can revert the code in Git and redeploy.
- **Automation (CI/CD)** → Git acts as the source of truth for Jenkins, GitHub Actions, or Azure DevOps pipelines.
- **GitOps** → Tools like ArgoCD and Flux automatically deploy infrastructure changes from Git.

Real-world scenario:
Your CI/CD pipeline triggers a deployment whenever a commit is pushed to the **main** branch. Without Git, this automation wouldn't be possible.

3. Git vs GitHub

- Git: Installed on your local machine → manages version history.
- GitHub/GitLab/Bitbucket: Cloud platforms → host repositories, manage pull requests, enable team collaboration.

Example:
You use git commit on your laptop → that saves changes locally.
You then use git push → that uploads your code to GitHub, where your team can review it.

4. Git Installation & Configuration

```
# Install Git (Ubuntu/Debian)
sudo apt install git -y

# Configure user info (needed for commits)
git config --global user.name "<your-username>"
git config --global user.email "abc.cde@example.com"

# Verify settings
git config --list
```

Explanation:

- user.name → Your identity in commits.
- user.email → Helps track who made changes.
  If you don't set this, commits may appear as "unknown" in GitHub.

---

5. Git File Lifecycle

Files in Git can be in 4 states:

1. **Untracked** → File exists but Git doesn't track it.
2. **Staged** → File is marked for commit.
3. **Committed** → File snapshot saved in repository.
4. **Pushed** → Commit sent to remote (GitHub).

Example:

```
touch app.py                    # Create new file
git status                      # Shows: untracked
git add app.py                  # Now staged
git commit -m "Add app.py"      # Now committed
git push origin main            # Now pushed to GitHub
```

6.Basic Git Commands

a) Repository Setup

```
git init                    # Initialize local repo
git clone <repo-url>        # Clone remote repo
```

Example:

- **git init** → Start a project from scratch (e.g., writing Ansible playbooks).
- **git clone** → Copy existing repo (e.g., pulling Kubernetes Helm charts from GitHub).

B) Tracking Changes

```
git status              # Show file states
git add file.txt        # Stage a specific file
git add .               # Stage all files
git commit -m "Message"   # Save snapshot
```

Example:

```
echo "print('Hello')" > script.py
git add script.py
git commit -m "Add Python script for test"
```

- Now script.py is permanently tracked in history.

---

c) Pushing & Pulling

```
git push origin main   # Upload local commits
git pull origin main   # Fetch + merge remote commits
```

Scenario:

- You worked on Jenkinsfile locally.
- Teammate already updated Jenkinsfile on GitHub.
- Before pushing, run:

  git pull origin main

- to avoid conflicts.

---

7. Branching Basics

```
git branch feature1     # Create new branch
git checkout feature1    # Switch branch
git checkout -b bugfix   # Create + switch
```

Why Branches?

- Main branch = Stable production-ready code.
- Feature branches = Safe place for new work.

Example Workflow:

- main → Production Kubernetes manifests.
- feature/logging → Add monitoring sidecar container.

- Later → Merge into main after testing.

---

8. Undoing Mistakes (Beginner Level)

```
git checkout -- file.py    # Discard local changes
git reset HEAD file.py     # Unstage file
git commit --amend         # Edit last commit
```

Example:

- You accidentally committed a typo in a commit message.

  git commit --amend -m "Fix typo in README"

- Now the commit history looks cleaner.

---

9. .gitignore

A file to tell Git which files to ignore.
Example .gitignore:

```
*.log
node_modules/
.env
```

DevOps Example:

- Ignore .env (secrets).
- Ignore node_modules/ (dependencies can be reinstalled).
- Ignore *.log (runtime logs shouldn't go to Git).

---

10. Basic Git Workflow (Step by Step)

- **Clone repo** → git clone
- **Edit files** → Add new code (e.g., Dockerfile)
- **Stage changes** → git add .
- **Commit** → git commit -m "Added Dockerfile"
- **Push** → git push origin main
- **Pull updates** → git pull origin main

Example:

If you're updating a Terraform script for AWS EC2:

```
git pull origin main
git checkout -b feature/ec2
# edit ec2.tf
git add ec2.tf
git commit -m "Add EC2 instance config"
git push origin feature/ec2
```

---

11. Best Practices for Beginners

✓ **Commit frequently** → "small commits > big commits"

✓ **Use clear commit messages** → e.g., Fix Nginx config error instead of update

✓ **Never commit secrets** → Use .gitignore for .env

✓ Always **git pull** before **git push**

✓ Use branches with naming conventions:

- **feature/** → for new features
- **bugfix/** → for fixing issues
- **hotfix/** → urgent production fixes

Intermediate Git Notes for DevOps Engineers (Deep Explanation + Examples)

---

1. Git Stash (Save Work Temporarily)

What it is:

- Imagine you are working on a new Jenkins pipeline (Jenkinsfile) but suddenly you need to switch to another branch to fix a production bug.
- Instead of committing incomplete code, you can stash your work.

```
git stash          # Save uncommitted changes
git stash list     # Show saved stashes
git stash apply    # Apply latest stash back
git stash pop      # Apply + remove stash
```

Example (DevOps use case):
You are editing deployment.yaml for Kubernetes, but you get an urgent request to fix ingress.yaml.

- **Run git stash** → your work is saved aside.
- **Switch to bugfix/ingress branch** → fix & commit.
- **Come back** → git stash pop → continue where you left.

---

2. Git Tag (Marking Releases)

What it is:

- Tags mark specific commits as release points.
- Commonly used in DevOps to mark production deployments.

```
git tag v1.0.0                  # Create lightweight tag
git tag -a v1.1.0 -m "Release"  # Annotated tag
git push origin v1.1.0          # Push tag to remote
git tag                         # List tags
```

Example:

- You deploy an app successfully → mark that commit as v1.0.0.
- Later, if a rollback is needed, just checkout that tag:

  git checkout v1.0.0

---

3. Branching Workflows (Real-World)

There are different branching strategies depending on the project.

Git Flow (Common in DevOps projects)

main → production-ready code

develop → integration/testing branch

feature/* → new features

release/* → preparing a release

hotfix/* → emergency fixes

Example Workflow:

- New Terraform feature → **git checkout -b feature/terraform-s3**
- Merge into **develop** for testing.
- After testing, merge into **main**.

This prevents breaking production while still allowing collaboration.

---

4. Merge vs Rebase

Merge: Combines histories of branches.
Rebase: Moves commits on top of another branch (keeps history cleaner).

```
# Merge
git checkout main
git merge feature1

# Rebase
git checkout feature1
git rebase main
```

Example:

- You're working on feature/docker branch while teammates pushed new commits to main.
- Instead of merging (which creates an extra commit), you run:

  git rebase main

- Now your commits appear on top of the latest main branch (clean history).

Warning: Never rebase public/shared branches. Safe for local feature branches only.

---

5. Resolving Merge Conflicts

Happens when two developers edit the same line in the same file.
Example Conflict (in app.py):

```
<<<<<<< HEAD
print("Hello from main")
=======
print("Hello from feature")
>>>>>>> feature-branch
```

To fix:

- Edit file manually to choose correct code.
- Stage & commit:

```
git add app.py
git commit -m "Resolve conflict in app.py"
```

DevOps Example:
If two engineers update the same nginx.conf, you must resolve which configuration to keep before pushing.

---

6. Cherry-Pick

What it is:

- Copy a commit from one branch to another.

git cherry-pick <commit-id>

Example (DevOps):

- Bug fix was committed in develop but production (main) urgently needs it.
- Instead of merging the whole branch, you cherry-pick just that commit.

---

7. Git Reset vs Revert

- **Reset** → Moves HEAD to an older commit (rewrites history).
- **Revert** → Creates a new commit that undoes changes (safe for shared repos).

```
git reset --hard <commit-id>   # Dangerous (loses history)
git revert <commit-id>         # Safe (keeps history)
```

Example:
- A bad Terraform script broke infra in main.
- Instead of deleting history, use git revert so teammates can see the rollback commit.

---

8. Git Log & Blame

```
git log --oneline --graph --decorate --all      # Visual history
git blame file.txt                              # Who changed each line
```

Example:
If **deployment.yaml** has a wrong replica count, use **git blame** to find who last changed that line.

---

9. Forking & Pull Requests (PRs)

- **Forking** → Copy a repo into your own GitHub account.
- **PR** → Propose changes to the original repo.

DevOps Example:

- You fork an open-source Helm chart repo.
- Customize it for your project.
- Create a PR back to the main project → collaborate with community.

---

10. Git Hooks (Automation)

Scripts that run automatically on Git actions.
Examples:

- **pre-commit** → Run lint check before committing.
- **pre-push** → Run unit tests before pushing.

Example:
In a DevOps project, create a pre-commit hook that checks Terraform formatting:

```
#!/bin/sh
terraform fmt -check
```

---

11. Best Practices for Intermediate Users

✓ Always pull latest changes before starting work

✓ Use meaningful branch names (feature/api-logging, bugfix/nginx-404)

✓ Prefer rebase for feature branches (cleaner history)

✓ Use revert (not reset) on shared branches

✓ Tag releases (v1.0.0, v2.0.0) for easy rollback

Advanced Git Notes for DevOps Engineers (Deep Explanation + Examples)

---

1. Git Internals (How Git Actually Works)

Git is not just a "version control" tool — it's a **content-addressable** filesystem.

Commit → Snapshot of your project at a given time.

Tree → Represents directories.

Blob → Represents file contents.

HEAD → Pointer to the latest commit in the current branch.

Example:

git cat-file -p HEAD

Shows what the HEAD commit actually stores.

DevOps Relevance: Understanding Git internals helps when debugging issues in CI/CD pipelines where commits are missing or wrong tags are deployed.

---

2. Submodules (Managing Multiple Repos Together)

Used when you want to include another Git repo inside your repo.

```
git submodule add https://github.com/example/ansible-playbooks.git ansible/
git submodule update --init --recursive
```

Example (DevOps):

- You have a main repo with Kubernetes manifests.
- Another team manages Helm charts in a different repo.
- Use Git submodules to link both.
- **Warning**: Submodules can be tricky in CI/CD, always document exact commit SHA.

---

3. GitOps Workflows

GitOps = Managing infrastructure & apps via Git.

- Git is the single source of truth.
- Tools like ArgoCD or Flux continuously sync Kubernetes cluster with Git repo.

Example Workflow:

- Dev pushes new deployment.yaml with updated Docker image.
- GitOps tool (e.g., ArgoCD) detects change.
- Automatically applies to Kubernetes cluster.

Advantage: No manual kubectl apply, everything is declarative & auditable in Git.

---

4. Reflog (Time Machine for Git)

Tracks everything you did in Git (even things you "lost").

```
git reflog
```

Example:

- You did git reset --hard and lost commits.
- Run git reflog, find commit SHA, restore with:

```
git checkout <commit-id>
```

DevOps Example: If a teammate accidentally resets a production branch, you can recover it via reflog.

---

5. Sparse Checkout (Partial Clones)

- Sometimes repos are HUGE (e.g., monorepos with Terraform, Helm, Dockerfiles).

  You may not need everything.

```
git sparse-checkout init
git sparse-checkout set terraform/ docker/
```

DevOps Use Case: In a microservices repo, you only want the serviceA/ code for debugging, not the entire repo.

---

6. Advanced Merge Strategies

```
git merge --squash feature-branch   # Merge without keeping history
git merge --no-ff feature-branch    # Keep history, create merge commit
```

Example:

- If you want a clean history of feature completion → use **--squash.**
- If you want to preserve collaboration history → use **--no-ff.**

---

7. Bisect (Find Buggy Commit)

Git can automatically find which commit introduced a bug.

```
git bisect start
git bisect bad                  # mark current commit as bad
git bisect good <commit-id>     # mark last known good commit
```

Git will keep checking commits until it finds the first bad commit.

DevOps Example:

- A pipeline suddenly started failing.
- Use git bisect to pinpoint which commit broke the Jenkinsfile.

---

8. Signed Commits (Security in DevOps)

Ensure commits are from trusted developers.

```
git config --global user.signingkey <GPG_KEY>
git commit -S -m "Signed commit"
```

Example: In regulated industries (finance, healthcare), all Git commits must be signed & verifiable.

---

9. Advanced CI/CD Integration

- Git is the trigger point for pipelines in Jenkins, GitHub Actions, GitLab CI, Azure DevOps.
- Best practice:

  main branch → deploy to prod

  develop branch → deploy to staging

  Feature branches → run unit tests only

Example Jenkins pipeline trigger:

```
pipeline {
  triggers {
    githubPush()
  }
}
```

---

10. Monorepo vs Multirepo Strategies

Monorepo → All microservices & infra in one repo.

Multirepo → Each service has its own repo.

DevOps Example:

- **Monorepo**: Easier CI/CD for a tightly integrated project.
- **Multirepo**: Better isolation for teams (Terraform infra separate from app code).

---

11. Advanced Branching Models

- **Trunk-Based Development**

  -Everyone commits to main frequently.

  -CI/CD ensures nothing breaks.

- **Release Branching**

  -Long-term branches for major releases (release/2.0).

  -Maintains stability while allowing new features in develop.

---

12. Git Best Practices for Enterprises

✅ Protect main & production branches with branch protection rules
✅ Enforce PR reviews before merging
✅ Use Git Hooks to run tests before pushing
✅ Always tag releases (v1.0.0, v2.1.0)
✅ Automate deployment via GitOps/CI-CD

Final Takeaway

- **Basic Git** → Day-to-day commands (clone, add, commit, push, branch, pull).
- **Intermediate Git** → Workflows, stashing, rebasing, tagging, cherry-picking, resolving conflicts.
- **Advanced Git** → Git internals, GitOps, submodules, reflog recovery, bisect debugging, signed commits, enterprise strategies.
- As a DevOps Engineer, Git is not just code versioning — it becomes the control center of your CI/CD pipelines, infra automation, and GitOps deployments.

Scenario-Based Git Notes for DevOps Engineers

1.Scenario: Accidentally deleted a branch
Problem: You deleted a feature branch locally & remotely.
Solution: Use git reflog to recover.

```
git reflog
git checkout -b feature_branch <commit-id>
git push origin feature_branch
```

✓ Recovery is possible because Git stores history in reflog.

2.Scenario: Need to revert a production deployment
Problem: A wrong commit was pushed to main and deployed to production.
Solution: Use git revert (safe option, doesn't break history).

```
git revert <commit-id>
git push origin main
```

✓ Difference: git reset rewrites history → avoid in production.

3. Scenario: CI/CD pipeline failing after merge
Problem: After merging a PR, Jenkins/GitHub Actions fails.
Solution: Use git bisect to find buggy commit.

```
git bisect start
git bisect bad
git bisect good <last-stable-commit>
```

✓ Git finds the commit that introduced the failure.

4. Scenario: You only want one folder from a large monorepo
Problem: Repo has 5GB of code, but you need only /terraform.
Solution: Use sparse checkout.

```
git clone --no-checkout <repo-url>
cd repo
git sparse-checkout init
git sparse-checkout set terraform/
```

✓ Saves time & bandwidth in CI/CD pipelines.

5.Scenario: Wrong commit pushed with secrets
Problem: API key accidentally pushed to GitHub.
Solution: Use git filter-repo or BFG Repo-Cleaner.

```
bfg --delete-files id_rsa
git push --force
```

✓ Also rotate the secret in your cloud provider (AWS/Azure/GCP).

6.Scenario: Developers using feature branches, but history is messy
Problem: PR history full of small commits like "fix typo", "WIP".
Solution: Use interactive rebase before merging.

```
git rebase -i HEAD~5
```

✅ Choose squash for messy commits → clean commit history.

7.Scenario: Deploy different environments (Dev, Staging, Prod)
Problem: You need same codebase, but different configs for environments.
Solution: Use Git branching strategy.

develop → Dev environment

staging → Pre-production testing

main → Production

Deploy pipelines based on branch:

```
if: github.ref == 'refs/heads/main'
```

✅ Automates multi-environment deployments.

8.Scenario: Team needs to release multiple versions
Problem: Some clients use v1.0, others v2.0.
Solution: Use tags & release branches.

```
git tag v1.0.0
git checkout -b release/2.0
```

✅ Tags help CI/CD pipelines deploy exact versions.

9.Scenario: You need to audit who changed what in critical files
Problem: A config file was changed, production is down.
Solution: Use git blame.

```
git blame docker-compose.yaml
```

✅ Shows who last modified each line → useful in RCA (Root Cause Analysis).

10.Scenario: Monorepo with multiple microservices
Problem: CI/CD builds every service even if one is changed.
Solution: Use Git diff to detect changes.

```
git diff --name-only HEAD~1 HEAD
```

✅ Example: If only /serviceA/ changed, run only serviceA's pipeline.

Final Summary

Scenario-Based Git Notes (for DevOps) help you solve:

- Branch deletion & recovery
- Wrong commits in production
- CI/CD pipeline failures
- Handling secrets
- Multi-environment deployments
- Version releases with tags
- RCA with git blame
- Monorepo optimization

Git Cheat Sheet with Explanation

1. Basic Git Commands (Beginner Level)

These are the most common commands every DevOps engineer needs to know.

Command	Explanation	Example
git init	Initialize a new Git repo in a directory	git init
git clone <url>	Copy an existing repo from GitHub/GitLab/Bitbucket	git clone https://github.com/user/repo.git
git status	Check status of files (modified, staged, untracked)	git status
git add <file>	Stage a file for commit	git add index.html
git add .	Stage all files	git add .
git commit -m "msg"	Save changes with a message	git commit -m "Added login feature"
git log	Show commit history	git log --oneline
git diff	Show changes not staged	git diff
git branch	List branches	git branch
git checkout <branch>	Switch branch	git checkout dev
git checkout -b <branch>	Create + switch branch	git checkout -b feature/auth
git merge <branch>	Merge a branch into current	git merge feature/auth
git push origin <branch>	Push branch to remote	git push origin dev
git pull	Fetch + merge changes from remote	git pull

2. Intermediate Git Commands (Team Collaboration)

Useful when working in teams, handling branches, and managing remote repos.

Command	Explanation	Example
git remote -v	Show remote repos	git remote -v
git fetch	Download latest commits, but don't merge	git fetch origin
git stash	Save changes temporarily	git stash
git stash pop	Reapply stashed changes	git stash pop
git reset <file>	Unstage a file	git reset index.html
git reset --hard <commit>	Reset repo to a commit (dangerous, removes changes)	git reset --hard abc123
git rebase <branch>	Reapply commits on top of another branch	git rebase main
git tag <tag>	Create a tag (release versioning)	git tag v1.0.0
git push origin --tags	Push tags to remote	git push origin --tags
git cherry-pick <commit>	Apply a specific commit to current branch	git cherry-pick abc123
git revert <commit>	Undo a commit safely (adds a new commit)	git revert abc123
git log --graph --oneline	Visualize commit tree	git log --graph --oneline --all
git rm <file>	Delete file from Git	git rm config.yaml

3. Advanced Git Commands (DevOps & Production Use)

These are powerful commands for debugging, recovery, and optimizing workflows.

Command	Explanation	Example
git reflog	Show all history (even deleted commits/branches)	git reflog
git bisect	Find buggy commit by binary search	git bisect start
git blame <file>	Show who changed each line of a file	git blame app.py
git archive	Export repo as .zip or .tar	git archive --format=zip HEAD > repo.zip
git gc	Cleanup unnecessary files & optimize repo	git gc
git filter-repo (or BFG)	Remove sensitive data from history	bfg --delete-files password.txt
git submodule	Add another repo inside repo (monorepo use)	git submodule add https://github.com/lib.git
git sparse-checkout	Checkout only part of repo (useful for monorepos)	git sparse-checkout set terraform/
git worktree	Work on multiple branches simultaneously	git worktree add ../branch2 branch2
git clean -fd	Remove untracked files & dirs	git clean -fd
git show <commit>	Show details of a commit	git show abc123
git shortlog -sn	Show commit count per author	git shortlog -sn