

Git for Data Engineers

Version control workflows tailored to data pipelines, SQL transformation projects, and team collaboration.

Last reviewed 29 Sep 2026 ·

Prerequisites: [Linux & Bash](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/linux-bash/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/linux-bash/)

Related: [dbt](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/) · [Terraform](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/terraform-for-de/) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/terraform-for-de/) · [Testing and CI/CD](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/testing-cicd/) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/testing-cicd/) · [Claude Code](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/claude-code/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/claude-code/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Overview

Challenge: Data pipelines are code — SQL transformations, orchestration definitions, Spark jobs, and infrastructure configuration. Without version control there is no reliable record of who changed what, no review process, and no safe way to roll back a change that breaks a production load.

Solution: Git records every change as a commit with an author, timestamp, and message. Work happens on *branches* isolated from production, changes are reviewed in *pull requests*, and they are merged only after automated tests pass. When a regression does reach production, the responsible commit can be identified and reverted.

Without Git:
edit prod SQL directly
→ PR → CI tests → review → merge
"who changed this join?"
models/fct_orders.sql
"it worked yesterday"
git bisect → the exact commit

With Git: 
branch → change

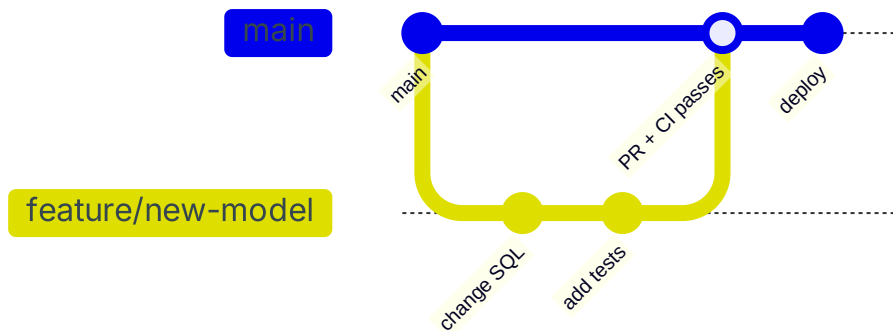
git blame

git log -p /

copy files to back them up
is kept, forever

every version

Relevance to data engineering: Git is the foundation of CI/CD for data: transformation tests, orchestration deployments, and infrastructure plans all run on a push or a pull request.



On this page

Basics - [Core Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#core-concepts) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#core-concepts) - [Essential Commands](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#essential-commands) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#essential-commands) - [Staging & Committing](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#staging--committing) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#staging--committing) - [Branching](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#branching) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#branching)

Intermediate - [Remote Repositories](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#remote-repositories) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#remote-repositories) - [Merging & Rebasing](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#merging--rebasing) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#merging--rebasing) - [Undoing Changes](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#undoing-changes) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#undoing-changes) - [.gitignore for DE Projects](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#gitignore-for-de-projects) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#gitignore-for-de-projects)

Advanced - [Git Workflow for Data Teams](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#git-workflow-for-data-teams) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#git-workflow-for-data-teams) - [Git for Data Pipeline Projects](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#git-for-data-pipeline-projects) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#git-for-data-pipeline-projects) - [Git Hooks in Pipelines](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#git-hooks-in-pipelines) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#git-hooks-in-pipelines) - [Useful Aliases & Tips](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#useful-aliases--tips) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#useful-aliases--tips)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/#further-reading)

Core Concepts

Concept	Definition
Repository	A directory tracked by Git; contains the full history of all changes
Working tree	Your local files as they are right now
Staging area	Files marked to include in the next commit (<code>git add</code>)
Commit	A snapshot of staged changes with a message, timestamp, and author
Branch	A lightweight pointer to a commit; branches diverge and merge
HEAD	Pointer to the current commit (usually the tip of the current branch)
Remote	A copy of the repo hosted elsewhere (GitHub, GitLab, Bitbucket, Azure DevOps)
origin	Default name for the primary remote
Clone	A full local copy of a remote repo
Fork	A personal copy of someone else's repo (GitHub concept)

```
Working tree → (git add) → Staging area → (git commit) →  
Local repo → (git push) → Remote  
  
↑  
  
pull / git fetch + merge) (git
```

Essential Commands

```
# — Setup —  
git config --global user.name "Alice Smith"  
git config --global user.email "alice@example.com"  
git config --global core.editor "vim"  
git config --list # view all  
config
```

```
# — Init / Clone —————
git init                                # init new repo
in current directory
git clone https://github.com/org/repo.git # clone a remote
repo
git clone https://github.com/org/repo.git my-dir # clone
into specific folder

# — Status & Inspection —————
git status                              # what's changed
/ staged / untracked
git diff                                # unstaged
changes
git diff --staged                       # staged changes
(about to be committed)
git log                                  # commit history
git log --oneline                       # compact one-
line log
git log --oneline --graph --all         # visual branch
tree
git log --author="Alice" --since="1 week ago"
git show abc1234                        # show a
specific commit
git blame models/fct_orders.sql         # who changed
each line
```

Staging & Committing

```
# Stage files
git add file.py                        # stage a specific file
git add models/                        # stage an entire
directory
git add *.sql                          # stage all SQL files
git add -p                            # interactive staging –
pick specific hunks

# Commit
git commit -m "feat: add incremental load to fct_orders"
git commit                             # opens editor for
```

longer message

```
# Stage + commit in one step (only for tracked files)
git commit -am "fix: correct NULL handling in stg_orders"
```

```
# Amend last commit (before pushing)
git commit --amend -m "new message"      # change message
git commit --amend --no-edit             # add more staged
changes to last commit
```

Good commit messages

```
# Format: <type>: <short summary in imperative mood>
# Body explains WHY, not WHAT (the diff shows what)
```

```
feat: add incremental load strategy to fct_orders
fix: handle NULL customer_id in stg_orders join
refactor: extract clean_phone_number into a macro
test: add unique and not_null tests to dim_customer
docs: document bronze-silver-gold naming convention
chore: bump pyarrow to 17.0.0
```

```
# Types: feat, fix, refactor, test, docs, chore, perf, ci
```

Branching

```
# Create and switch
git branch feature/incremental-orders      # create branch
git checkout feature/incremental-orders    # switch to it
git checkout -b feature/incremental-orders # create +
switch in one step
git switch -c feature/incremental-orders   # modern syntax
(Git 2.23+)
```

```
# List branches
```

```
git branch          # local branches
git branch -r       # remote branches
git branch -a       # all branches
```

```
# Delete
git branch -d feature/done      # delete (safe — won't
delete if unmerged)
git branch -D feature/abandon   # force delete
git push origin --delete feature/old-branch # delete
remote branch

# Rename current branch
git branch -m new-name
```

Branch naming conventions

```
main / master      — production-ready code
develop            — integration branch (some teams)

feature/<ticket>-<description> # feature/DE-123-add-
orders-pipeline
fix/<ticket>-<description>      # fix/DE-456-null-
handling
hotfix/<description>           # hotfix/critical-prod-
fix
release/<version>              # release/v1.2.0
```

Remote Repositories

```
# Remotes
git remote -v          # list remotes
git remote add origin https://github.com/org/repo.git
git remote set-url origin <new-url> # change remote
URL

# Fetch vs Pull
git fetch origin        # download
remote changes, don't merge
git fetch --all         # fetch all
remotes
git pull                # fetch + merge
(or rebase if configured)
git pull origin main    # pull specific
```

```
branch

# Push
git push origin feature/my-branch      # push branch to
remote
git push -u origin feature/my-branch    # push + set
upstream (then just git push)
git push                                # push current
branch to upstream
git push --force-with-lease              # safer force
push (fails if remote changed)

# Tracking remote branches
git checkout --track origin/feature/remote-branch # track
+ checkout remote branch
```

Merging & Rebasing

```
# Merge – creates a merge commit, preserves history
git checkout main
git merge feature/incremental-orders
git merge --no-ff feature/branch        # always create a
merge commit
git merge --squash feature/branch        # squash all
commits into one staged change

# Rebase – replays your commits on top of another branch
(linear history)
git checkout feature/my-branch
git rebase main                          # replay feature commits on top
of latest main

# Interactive rebase – rewrite, squash, or reorder commits
git rebase -i HEAD~3                     # rewrite last 3 commits
# Commands: pick, squash (s), reword (r), drop (d), fixup
(f)

# Resolve merge conflicts
# Git marks conflicts in the file:
# <<<<<<< HEAD (your changes)
```

```
# your changes
# =====
# their changes
# >>>>> feature/branch (incoming)

# After resolving:
git add resolved_file.sql
git merge --continue      # or: git rebase --continue

# Abort if it gets messy
git merge --abort
git rebase --abort
```

When to use which

	Merge	Rebase
History	Non-linear (merge commits)	Linear
Shared branches	Yes	No — never rebase shared/public branches
Feature branches	Both work	Rebase on main before PR = clean history
Hotfixes	Merge	Not typical

Undoing Changes

```
# Discard unstaged changes in a file
git restore file.py          # modern syntax
git checkout -- file.py      # old syntax

# Discard all unstaged changes
git restore .

# Unstage (remove from staging area, keep changes)
git restore --staged file.py
git reset HEAD file.py       # old syntax

# Undo last commit, keep changes staged
git reset --soft HEAD~1
```

```

# Undo last commit, keep changes unstaged
git reset HEAD~1

# Undo last commit, DISCARD changes (DESTRUCTIVE)
git reset --hard HEAD~1

# Revert a commit (safe – creates a new commit that undoes it)
git revert abc1234                    # use on shared/public
branches

# Stash – save work temporarily without committing
git stash                            # stash current
changes
git stash push -m "WIP: orders fix" # stash with a name
git stash list                       # list stashes
git stash pop                        # apply latest stash
and remove it
git stash apply stash@{1}            # apply specific
stash, keep it
git stash drop stash@{1}             # delete a stash
git stash branch feature/recover     # create branch from
stash

```

.gitignore for DE Projects

```

# Python
__pycache__/
*.py[cod]
.venv/
venv/
.env
*.egg-info/
dist/
build/
.pytest_cache/
.mypy_cache/

# Jupyter

```

```

.ipynb_checkpoints/
*.ipynb_metadata

# SQL transformation tools (e.g. dbt)
target/                # compiled SQL and run artifacts
dbt_packages/          # installed packages (like
node_modules)
logs/
.dbt/                  # local profiles (if inside project)

# Airflow
logs/
airflow.db
airflow.cfg            # may contain local config

# Data files – never commit large data
*.csv
*.parquet
*.json
*.avro
*.orc
data/
raw/
output/

# Secrets – NEVER commit these
.env
*.pem
*.key
secrets/
credentials.json
service_account.json
~/.aws/credentials    # (gitignore on system level)

# OS
.DS_Store
Thumbs.db

# IDE
.vscode/
.idea/
*.swp

```

```
*.SWO

# Terraform
.terraform/
*.tfstate
*.tfstate.backup
terraform.tfvars      # often contains secrets
```

Git Workflow for Data Teams

Trunk-based development (recommended for data teams)

```
main (always deployable)
  ↑
  ├── feature/DE-123-add-orders-pipeline (short-lived, <
1-2 days)
  ├── fix/DE-124-null-in-join            (short-lived)
  └── feature/DE-125-kafka-consumer      (short-lived)

PR → review → squash merge → delete branch
```

Feature branch workflow

```
# 1. Start from latest main
git checkout main
git pull origin main

# 2. Create a feature branch
git checkout -b feature/DE-123-incremental-orders

# 3. Work, commit often (small, atomic commits)
git add models/marts/fct_orders.sql
git commit -m "feat: add incremental strategy to
fct_orders"

git add tests/
git commit -m "test: add row count assertion for
fct_orders"
```

```
# 4. Keep branch up to date with main
git fetch origin
git rebase origin/main    # or: git merge origin/main

# 5. Push and open a PR
git push -u origin feature/DE-123-incremental-orders
# Open PR on GitHub/GitLab

# 6. After PR approval – squash merge into main
# (done via GitHub UI or CLI)
gh pr merge 42 --squash --delete-branch

# 7. Sync local main
git checkout main && git pull
```

Git for Data Pipeline Projects

Repo structure

A single repository typically holds every layer of a pipeline, so one pull request can change ingestion, transformation, and orchestration together:

```
data-platform/
├── ingestion/           # extract/load jobs (Python,
connectors config)
├── transformations/    # SQL models or Spark jobs, by
layer
│   ├── staging/
│   ├── intermediate/
│   └── marts/
├── orchestration/      # DAG / workflow definitions
├── tests/              # unit tests, data tests,
fixtures
├── infra/              # Terraform or other IaC
├── .github/workflows/  # CI/CD
├── pyproject.toml      # pinned dependencies
└── .gitignore          # build artifacts, local
profiles, data files
```

CI/CD for data pipelines

```
# .github/workflows/ci.yml
name: Data pipeline CI

on:
  pull_request:
    paths: ['ingestion/**', 'transformations/**',
'orchestration/**', 'tests/**']

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Install dependencies
        run: pip install -e ".[dev]"

      - name: Lint Python and SQL
        run: |
          ruff check .
          sqlfluff lint transformations/

      - name: Unit tests
        run: pytest tests/unit

      - name: Validate orchestration definitions
        run: pytest tests/orchestration          # e.g. every
DAG imports without errors

      # Build only what changed, in an isolated CI schema,
      then run data tests
      - name: Build and test changed transformations
        run: ./scripts/build_changed.sh --target ci
        env:
          WAREHOUSE_USER: ${ secrets.CI_WAREHOUSE_USER }
          WAREHOUSE_PRIVATE_KEY: ${ secrets.CI_WAREHOUSE_PRIVATE_KEY }
```

Principles: use a dedicated CI schema or database per pull request · build and test only changed models and their downstream dependents · use a least-privilege CI service account with key-based auth · deploy to production only from `main`.

Tool example — dbt "slim CI": dbt compares the pull request with the production manifest and builds only modified models and their children, reading unchanged upstream models from production:

```
dbt build --target ci --select state:modified+ --defer --  
state ./prod_artifacts/
```

Dev → prod promotion

feature branch	→ local/dev schema	(developer runs and tests changes)
pull request	→ CI schema	(automated build + tests on changed models)
merge to main	→ production	(deployment job runs the same code with prod config)

Git Hooks in Pipelines

Git hooks are scripts that run automatically at certain Git events.

```
# Pre-commit hook – run linting/tests before every commit  
# File: .git/hooks/pre-commit (chmod +x)  
  
#!/usr/bin/env bash  
set -e  
  
echo "Running pre-commit checks..."  
  
# SQL lint check  
if command -v sqlfluff &> /dev/null; then  
    sqlfluff lint transformations/ || { echo "SQL linting failed"; exit 1; }  
fi  
  
# Python linting  
if command -v ruff &> /dev/null; then
```

```

    ruff check . || { echo "Ruff linting failed"; exit 1; }
fi

# Check for secrets
if git diff --cached | grep -E "
(password|secret|api_key)\s*=\s*['\"]" ; then
    echo "Potential secret detected in diff. Aborting."
    exit 1
fi

echo "All checks passed."

```

```

# pre-commit framework – manage hooks as config
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.3.0
    hooks:
      - id: ruff
      - id: ruff-format

  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.5.0
    hooks:
      - id: end-of-file-fixer
      - id: trailing-whitespace
      - id: check-yaml
      - id: detect-private-key
      - id: check-added-large-files
        args: ['--maxkb=500']

# Install
pip install pre-commit
pre-commit install          # installs into .git/hooks/pre-
                             commit
pre-commit run --all-files # run manually

```

Useful Aliases & Tips

```

# Add to ~/.gitconfig or ~/.bashrc

# Git aliases
git config --global alias.st "status"
git config --global alias.co "checkout"
git config --global alias.br "branch"
git config --global alias.lg "log --oneline --graph --all
--decorate"
git config --global alias.last "log -1 HEAD"
git config --global alias.undo "reset HEAD~1 --mixed"
git config --global alias.unstage "restore --staged"

# Bash aliases
alias gs="git status"
alias ga="git add"
alias gc="git commit -m"
alias gp="git push"
alias gl="git log --oneline --graph --all"
alias gco="git checkout"
alias gcb="git checkout -b"

# Find which branch introduced a bug
git bisect start
git bisect bad # current commit is broken
git bisect good abc1234 # this commit was good
# Git checks out midpoint – test and mark good/bad until
bug commit found
git bisect reset # end bisect

# Search commit history for a string
git log -S "function_name" --oneline # commits
that added/removed string
git log --all --grep="fix orders" # commits
with message matching

# Show file history
git log --follow -p models/fct_orders.sql # full diff
history of a file
git log --follow --oneline models/fct_orders.sql

# Recover a deleted branch (if you know the commit hash)
git reflog # shows all recent HEAD

```

movements

```
git checkout -b recovered abc1234 # create branch from that commit
```

Common Pitfalls

Pitfall	Symptom	Fix
Committing secrets (<code>.env</code> , connection profiles, keys)	Credentials in history — deleting the file later doesn't remove them	Rotate the secret immediately, then purge with <code>git filter-repo</code> ; add <code>detect-private-key</code> / <code>gitleaks</code> to pre-commit
Committing data files or <code>target/</code>	Repo balloons to GBs; clones take minutes	<code>.gitignore</code> data and build artifacts; use <code>check-added-large-files</code>
<code>git push --force</code> on a shared branch	Teammates' commits disappear	Use <code>--force-with-lease</code> , and only on your own feature branch
Rebasing a branch others have pulled	Duplicate commits and confusing conflicts for everyone	Only rebase local/private branches; merge shared ones
Long-lived feature branches	Huge conflict-ridden PRs that nobody reviews properly	Keep branches under a day or two; merge small, often
<code>git reset --hard</code> with uncommitted work	Work is gone (it was never committed, so <code>reflog</code> can't help)	<code>git stash</code> first; commit WIP early
Vague commit messages ("fix", "wip")	History is useless for finding when/why a model changed	Conventional commits: <code>fix: handle NULL customer_id in stg_orders</code>
Editing production directly, then committing	Git and prod drift apart; the next deploy undoes the hotfix	All changes go through Git; deploy from <code>main</code> only

Cheat Sheet

Task	Command
New branch from latest main	<code>git switch main && git pull && git switch -c feature/DE-123-x</code>
See what changed	<code>git status · git diff · git diff --staged</code>
Stage part of a file	<code>git add -p</code>
Commit	<code>git commit -m "feat: ..."</code>
Fix last commit (not pushed)	<code>git commit --amend --no-edit</code>
Update branch with main	<code>git fetch && git rebase origin/main</code>
Push new branch	<code>git push -u origin HEAD</code>
Undo last commit, keep changes	<code>git reset --soft HEAD~1</code>
Undo a pushed commit safely	<code>git revert <sha></code>
Discard local edits to a file	<code>git restore <file></code>
Park work in progress	<code>git stash / git stash pop</code>
Who changed this line	<code>git blame <file></code>
When was this string added/removed	<code>git log -S "text" --oneline</code>
Find the commit that broke it	<code>git bisect start → bad → good <sha></code>
Recover a "lost" commit	<code>git reflog → git switch -c rescue <sha></code>
Pretty history	<code>git log --oneline --graph --all</code>

Merge vs rebase in one line: rebase your own branch onto `main` to keep history linear; never rebase something others have pulled.

Interview Questions

Q: What is the difference between `git merge` and `git rebase`? A: Merge combines two branches by creating a merge commit that has both as parents — history is preserved exactly but becomes non-linear. Rebase replays your commits on top of another branch, rewriting them with new hashes, which gives a clean linear history. Because rebase rewrites commits, you should only rebase branches nobody else has pulled; for shared branches, merge. A common team pattern is: rebase your feature branch on `main` before opening a PR, then squash-merge.

Q: What's the difference between `git reset` and `git revert`? A: `reset` moves the branch pointer backwards, effectively removing commits from the branch (`--soft` keeps changes staged, `--mixed` keeps them unstaged, `--hard` discards them). It rewrites history, so it's for local, unpushed work. `revert` creates a new commit that undoes an earlier one — history stays intact, so it's the safe way to undo something already on a shared branch like `main`.

Q: What is `git fetch` vs `git pull`? A: `fetch` downloads new commits from the remote and updates remote-tracking branches (`origin/main`) but doesn't touch your working branch. `pull` is `fetch` followed by a merge (or rebase, if configured) into your current branch. Fetching first lets you inspect what changed before integrating it.

Q: A teammate accidentally committed an AWS key and pushed it. What do you do? A: First, rotate/revoke the key immediately — assume it's compromised the moment it's pushed, because the history is already cloned and possibly scraped. Then remove it from history with `git filter-repo` (or BFG) and force-push, and ask everyone to re-clone. Finally, prevent a repeat: add the file pattern to `.gitignore` and a secret scanner (`gitleaks`, `detect-private-key`) to pre-commit and CI.

Q: How would you set up CI/CD for a SQL transformation project with Git? A: Feature branches with pull requests into `main`. On every pull request, CI lints SQL and Python, runs unit tests, and builds only the changed models and their downstream dependents into an isolated CI schema, then runs data tests against them — reading unchanged upstream tables from production to keep runs fast and cheap (dbt calls this "slim CI" with `state:modified+ --defer`). Merging to `main` triggers the production deployment. Pre-commit hooks catch style and secret issues before review.

Q: What is trunk-based development and why do data teams like it? A: Everyone works on short-lived branches (hours to a day or two) that merge into a single `main` that's always deployable. It avoids the painful merges of long-lived `develop` / `release` branches. For data teams it pairs well with incremental CI builds and environment-based deploys: small changes are easy to review, easy to test against production data, and easy to revert.

Further Reading

- [Pro Git book](https://git-scm.com/book/en/v2) (https://git-scm.com/book/en/v2) — free, the definitive reference
- [Conventional Commits](https://www.conventionalcommits.org/) (https://www.conventionalcommits.org/)
- [pre-commit](https://pre-commit.com/) (https://pre-commit.com/) — hook framework used above
- [SQLFluff](https://docs.sqlfluff.com/) (https://docs.sqlfluff.com/) — SQL linter for CI and pre-commit
- [dbt: Defer](https://docs.getdbt.com/reference/node-selection/defer) (https://docs.getdbt.com/reference/node-selection/defer) — one tool's implementation of CI that builds only changed models
- [git-filter-repo](https://github.com/newren/git-filter-repo) (https://github.com/newren/git-filter-repo) — removing files/secrets from history

Previous: [Linux & Bash](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/linux-bash/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/linux-bash/) • **Next:** [Cloud Storage](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 29, 2026