GrowthSchool

GIT & GITHUB

COMPLETE NOTES

TRACK - BRANCH - SHIP

Version control, branching, PRs, CI/CD and team workflow

growthschool.cc | @growthschool.cc

Learning Roadmap

GrowthSchool

Follow this order. Each page has explanation, real commands, a diagram or example, common mistakes, a student lab and a revision snapshot. We build one ecommerce "Shop API" throughout.

HOW TO USE THESE NOTES

Read top to bottom the first time, then use pages 26-30 for fast revision before interviews. Type every command yourself in a throwaway repo. Reading is not remembering.

1. Why Version Control & Git

GrowthSchool

IN SIMPLE TERMS
Version control is a tool that saves the full history of your project so you can see every change, undo mistakes, and work with others without overwriting each other. Git is the most popular version-control tool.

* The Problem

- Copying folders like project-final-FINAL-v2 does not scale and loses history.
- Teams need to work in parallel, review changes, and undo mistakes without fear.

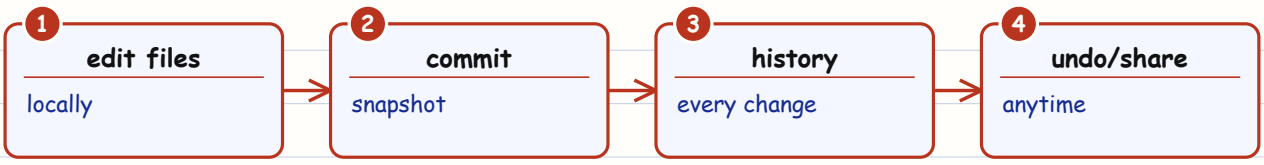
* What Git Is

- Git is a distributed version control system: every clone has the full history.
- It records snapshots (commits), not diffs, and links them into a directed history.

* Git vs GitHub

- Git is the tool that runs locally. GitHub is a hosting platform for Git repositories.
- GitHub adds pull requests, reviews, issues, Actions (CI/CD) and access control on top of Git.

Why version control



FIRST-TIME SETUP

```
# check your version and set identity
git --version
git config --global user.name  "Asha Dev"
git config --global user.email "asha@shop.example"
```

COMMON MISTAKE
Committing as the wrong identity. Set user.name and user.email before your first commit, or history will show the wrong author on every commit you make.

REVISION SNAPSHOT

ASK YOURSELF What makes Git "distributed", and how is it different from GitHub?

DO THIS Explain snapshot vs diff, and name three things GitHub adds on top of Git.

2. The Git Mental Model

GrowthSchool

IN SIMPLE TERMS
The Git mental model is understanding the four places your work lives: your files, the staging area, your local history, and the remote copy on a server.

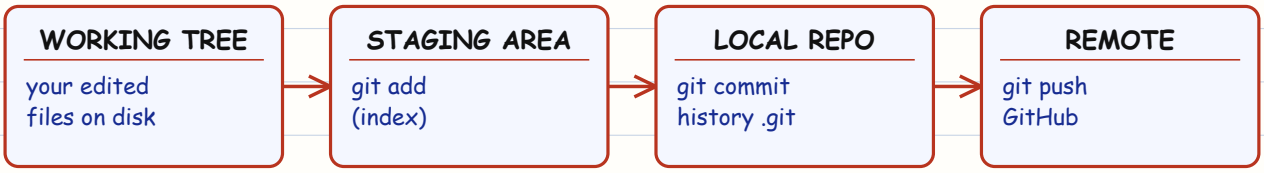
* Four Places Your Code Lives

- Working tree: the real files you edit. Staging area (index): what will go into the next commit.
- Local repository: committed history in the hidden .git folder. Remote: the shared copy (GitHub).

* Why Staging Exists

- Staging lets you craft a clean, logical commit instead of dumping every change at once.
- You can stage part of your work now and leave the rest for a separate, meaningful commit.

The Four Areas of Git



add -> commit -> push moves work right; fetch/checkout brings it back left.

REAL-WORLD EXAMPLE

```
git status            # what changed, what is staged
git add app/checkout.py   # stage one file
git add -p            # stage selected chunks interactively
git commit -m "add checkout total calculation"
```

STUDENT LAB
In a new repo, edit two files. Stage only one with git add, run git status, and confirm the other file stays "not staged". Commit, then observe how status changes.

REVISION SNAPSHOT

ASK YOURSELF	Name the four areas and the command that moves work between each.
DO THIS	Draw the model from memory: working -> staging -> local -> remote.

3. Start a Repository

GrowthSchool

IN SIMPLE TERMS
A repository (repo) is a project folder that Git is tracking. You either create a new one (init) or copy an existing one from a server (clone).

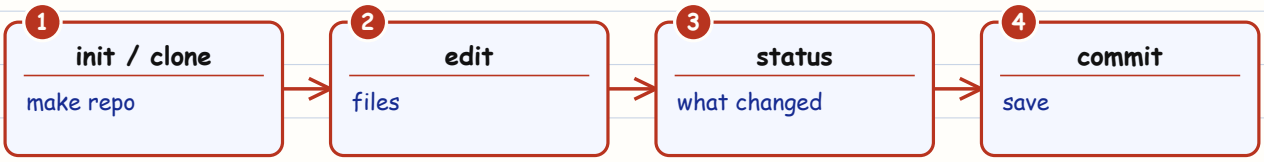
* Two Ways to Begin

- git init turns an existing folder into a repo. git clone <url> copies an existing remote repo.
- Cloning also sets up a remote named origin pointing back to where you cloned from.

* Inspect Before You Act

- git status is the command you will run most: it shows branch, staged, and unstaged changes.
- git log shows history; add --oneline --graph --all for a compact visual of all branches.

Start a repo



REAL-WORLD EXAMPLE

```
git init shop-api            # new local repo
git clone git@github.com:org/shop-api.git
git status
git log --oneline --graph --all --decorate
```

COMMON MISTAKE
Running git init inside an existing repo (nested repos) or in your home folder by accident. Check with git status / git rev-parse --show-toplevel before creating a new repo.

REVISION SNAPSHOT

ASK YOURSELF	When do you use init vs clone, and what does clone set up automatically?
DO THIS	Clone any public repo and read its history with git log --oneline --graph.

4. Staging, Commits, Log & Diff

GrowthSchool

IN SIMPLE TERMS
Committing is saving a snapshot of your staged changes into history, each with a message; diff and log let you see what changed and when.

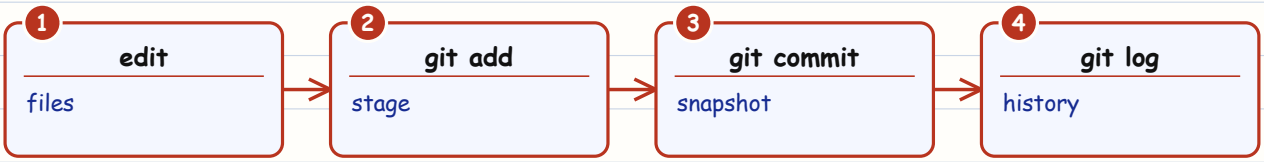
* Make Good Commits

- A commit is a snapshot plus author, timestamp and message. Keep each commit one logical change.
- Write messages in the imperative: "add", "fix", "refactor" - describing what the change does.

* See What Changed

- git diff shows unstaged changes; git diff --staged shows what is about to be committed.
- git log -p shows the patch for each commit; git log --stat summarises files changed.

Commit a change



REAL-WORLD EXAMPLE

```
git add .
git commit -m "fix: prevent negative cart quantity"
git diff              # working tree vs staged
git diff --staged     # staged vs last commit
git log --oneline -5
```

COMMON MISTAKE
Giant commits like "misc fixes" that mix features, formatting and bug fixes together. They make review, rollback and git bisect painful. Commit small and often.

REVISION SNAPSHOT

ASK YOURSELF What is the difference between git diff and git diff --staged?

DO THIS Make three small, well-described commits instead of one large one.

5. Exploring History

GrowthSchool

IN SIMPLE TERMS
Exploring history means reading the story of a project - who changed which line, when, and why - using log, show, and blame.

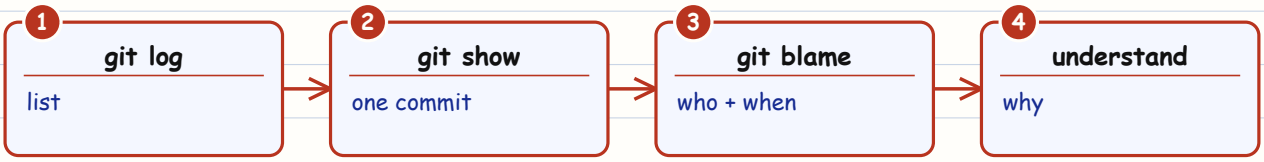
* Read the Story of the Code

- git show <sha> displays one commit in full. git log --oneline gives a scannable list.
- git blame <file> shows who last changed each line and in which commit - great for context.

* Find Things Fast

- git log --grep "checkout" searches commit messages. git log -S"functionName" finds when code appeared.
- git log -- path/to/file limits history to a single file or folder.

Read the history



REAL-WORLD EXAMPLE

```
git show a1b2c3d
git blame app/checkout.py
git log --oneline --grep "refund"
git log -S"calculateTax" --oneline
```

STUDENT LAB
Pick any file in a repo, run git blame, and trace one line back to its introducing commit with git show. Write one sentence explaining why that line exists.

REVISION SNAPSHOT

ASK YOURSELF	Which command tells you WHEN a specific function first appeared?
DO THIS	Use blame + show to explain the origin of one confusing line of code.

6. Branches

GrowthSchool

IN SIMPLE TERMS
A branch is a separate line of work, like a parallel copy of the project, so you can build a feature without disturbing the main version.

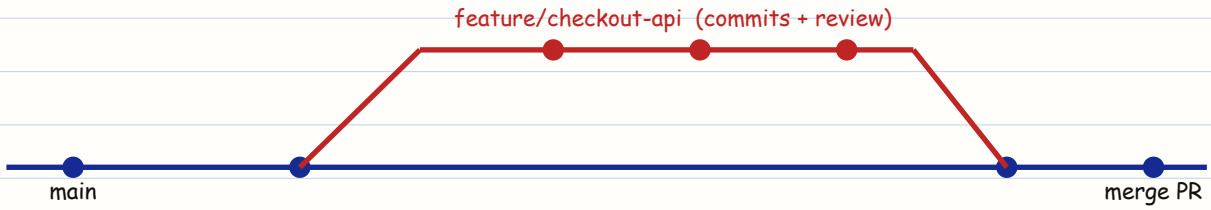
* What a Branch Really Is

- A branch is just a movable pointer to a commit. Creating one is instant and cheap.
- HEAD points to your current branch. Switching branches moves your working tree to that snapshot.

* Everyday Branching

- Create a branch per feature or fix so main always stays releasable.
- git switch is the modern, clearer command; git checkout still works for older habits.

Branch & Pull-Request Flow



REAL-WORLD EXAMPLE

```
git switch -c feature/checkout-api   # create + switch
git branch                # list local branches
git switch main
git merge feature/checkout-api
```

COMMON MISTAKE
Doing all work directly on main. One bad commit then blocks everyone and every release. Branch per change; keep main deployable at all times.

REVISION SNAPSHOT

ASK YOURSELF	If a branch is "just a pointer", what actually moves when you commit on it?
DO THIS	Create feature/x, commit twice, merge to main, then delete the branch.

7. Merge vs Rebase

GrowthSchool

IN SIMPLE TERMS
Merging and rebasing are two ways to combine branches: merge joins them keeping both histories, rebase re-writes your commits on top for a straight line.

* Merge

- git merge joins two branches and, when histories diverged, creates a merge commit.
- It preserves the true history but can produce a busy, non-linear graph.

* Rebase

- git rebase replays your commits on top of another branch, producing a clean, linear history.
- It rewrites commit SHAs, so never rebase commits that others have already pulled.

* Golden Rule

- Rebase local, private work to tidy it. Merge (or PR-merge) shared branches to preserve history.

REAL-WORLD EXAMPLE

```
# tidy your feature branch on top of latest main
git switch feature/checkout-api
git fetch origin
git rebase origin/main
# resolve conflicts, then: git rebase --continue
```

COMMON MISTAKE
Rebasing a branch that teammates already based work on - it rewrites shared history and forces everyone into painful conflicts. Rebase only private, unpushed commits.

REVISION SNAPSHOT

ASK YOURSELF	When is rebase safe and when is it dangerous?
DO THIS	Rebase a local branch onto main, then compare the graph to a merge result.

8. Merge Conflicts (Full Example)

GrowthSchool

IN SIMPLE TERMS
A merge conflict happens when two branches changed the same lines, so Git asks you to choose the correct final result by hand.

* Why Conflicts Happen

- Two branches changed the same lines. Git cannot decide which is correct, so it asks you.
- Conflicts are normal - not errors. Stay calm and resolve them deliberately.

* The Resolution Steps

- 1) Run the merge/rebase. 2) Open conflicted files. 3) Edit to the correct final result.
- 4) Remove the <<<< ==== >>>> markers. 5) git add the file. 6) commit or continue.

REAL-WORLD EXAMPLE

```
<<<<<<< HEAD
    total = price * qty
=======
    total = price * qty * (1 - discount)
>>>>>>> feature/checkout-api
# keep the correct line, delete markers, then:
git add app/checkout.py && git commit
```

COMMON MISTAKE
Committing the conflict markers (<<<<, ====, >>>>) still in the file. Always search the project for these markers before you finish, and re-run tests after resolving.

REVISION SNAPSHOT

ASK YOURSELF	What do the three marker sections HEAD / ==== / branch mean?
DO THIS	Force a conflict on purpose in a test repo and resolve it end to end.

9. cherry-pick

GrowthSchool

IN SIMPLE TERMS
cherry-pick copies one specific commit from one branch onto another, without merging everything else.

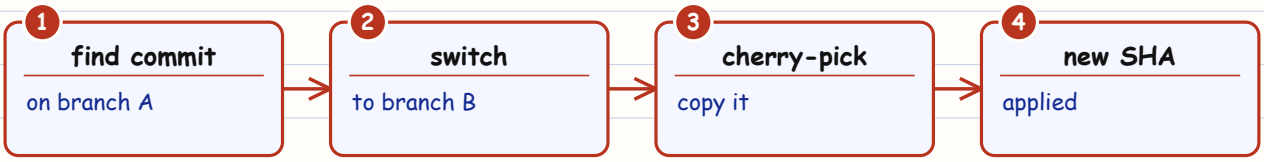
* What It Does

- git cherry-pick <sha> copies a single commit from one branch onto your current branch.
- Useful to bring one urgent fix to a release branch without merging everything else.

* Use Sparingly

- Cherry-picking duplicates the change as a new commit with a new SHA.
- Overuse creates divergent, hard-to-track history. Prefer proper merges when possible.

cherry-pick



REAL-WORLD EXAMPLE

```
git switch release/1.4
git cherry-pick 9f8e7d6      # bring hotfix commit only
git cherry-pick A^..B        # a range of commits
git cherry-pick --abort      # bail out on conflict
```

STUDENT LAB
Create a fix on main, then cherry-pick just that commit onto a release/1.0 branch. Confirm with git log that only that single change was applied.

REVISION SNAPSHOT

ASK YOURSELF	Why does a cherry-picked commit get a new SHA?
DO THIS	Cherry-pick one hotfix commit to a release branch cleanly.

10. reset, revert, restore

GrowthSchool

IN SIMPLE TERMS
reset, revert, and restore are the three ways to undo work in Git - each undoes a different thing in a different, safe or unsafe, way.

* git restore

- Discards changes in the working tree or unstages files. It does not touch commit history.
- git restore <file> reverts edits; git restore --staged <file> unstages.

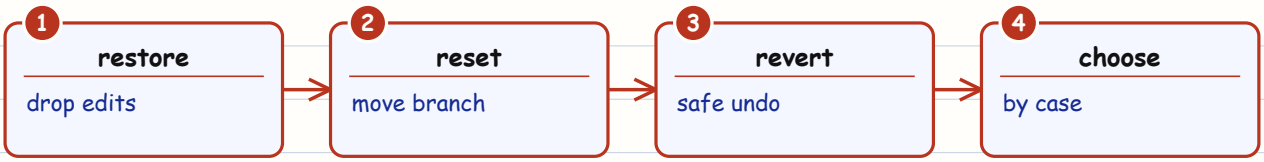
* git reset

- --soft keeps changes staged; --mixed (default) unstages; --hard discards everything.
- reset moves the branch pointer. --hard is destructive - be certain before using it.

* git revert

- Creates a NEW commit that undoes a previous one. Safe for shared/public history.

Pick the right undo



REAL-WORLD EXAMPLE

```
git restore app.py            # drop working edits
git reset --soft HEAD~1       # undo commit, keep changes staged
git reset --hard HEAD~1       # DANGER: discard commit + changes
git revert a1b2c3d            # safe undo on shared branch
```

COMMON MISTAKE
Using git reset --hard on shared branches or before saving needed work. Lost commits may still be recoverable via reflog, but treat --hard as a loaded weapon.

REVISION SNAPSHOT

ASK YOURSELF	Which undo command is safe on a branch others have pulled?
DO THIS	Practice soft vs hard reset in a scratch repo and observe the difference.

11. reflog & Recovery

GrowthSchool

IN SIMPLE TERMS
The reflog is Git's safety net: a private log of everywhere you have been, so you can recover commits that seem lost.

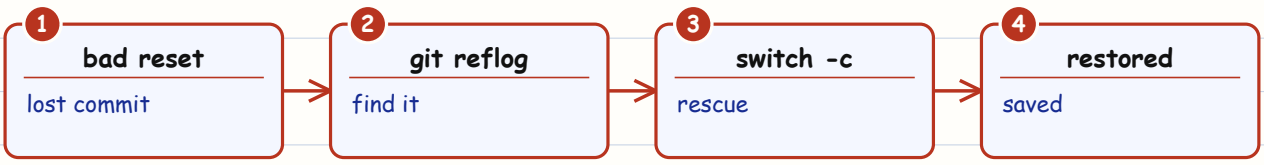
* Your Safety Net

- git reflog records where HEAD has been - even after resets, rebases and branch deletes.
- Most "lost" commits are not gone; they are just no longer referenced by a branch.

* Recover a Commit

- Find the commit SHA in reflog, then create a branch or reset back to it.
- Git eventually garbage-collects truly unreferenced commits, so recover promptly.

Recover with reflog



REAL-WORLD EXAMPLE

```
git reflog
# c0ffee HEAD@{2}: commit: add refund flow
git switch -c recovered c0ffee   # rescue lost work
# or: git reset --hard HEAD@{2}
```

STUDENT LAB
Commit something, run git reset --hard HEAD~1 to "lose" it, then recover it using git reflog. This single skill will save you one day.

REVISION SNAPSHOT

ASK YOURSELF	After an accidental hard reset, what is the first command you run?
DO THIS	Delete a commit with reset, then bring it back from the reflog.

12. git stash

GrowthSchool

IN SIMPLE TERMS
git stash temporarily parks your unfinished changes and gives you a clean workspace, so you can switch tasks and come back later.

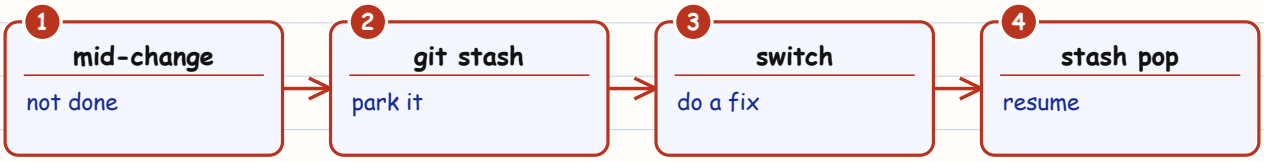
* Park Work Safely

- git stash saves uncommitted changes and gives you a clean working tree instantly.
- Perfect when an urgent fix arrives while you are mid-change on something else.

* Manage the Stash

- git stash list shows saved stashes; git stash pop re-applies and removes the latest.
- Use git stash push -m "msg" to label stashes and avoid a confusing pile of them.

Park work with stash



REAL-WORLD EXAMPLE

```
git stash push -m "wip: checkout ui"
git switch hotfix/login
# ...fix and commit...
git switch feature/checkout-ui
git stash pop
```

COMMON MISTAKE
Treating stash as long-term storage. Stashes are easy to forget and lose. For anything important, make a commit on a branch instead.

REVISION SNAPSHOT

ASK YOURSELF	When is stash better than a commit, and when is it worse?
DO THIS	Stash mid-work, switch branches to do a fix, then pop and continue.

13. Tags, Releases & SemVer

GrowthSchool

IN SIMPLE TERMS
A tag marks an important commit (usually a release); semantic versioning is the MAJOR.MINOR.PATCH numbering that tells users how big a change is.

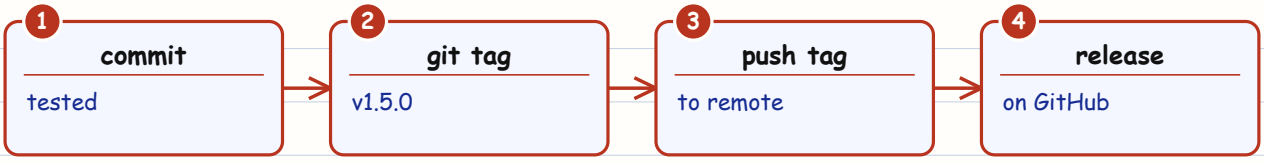
* Tags

- A tag marks a specific commit, usually a release point. Annotated tags store author + message.
- Push tags explicitly: git push origin v1.2.0 (or git push --tags).

* Semantic Versioning

- MAJOR.MINOR.PATCH: MAJOR = breaking change, MINOR = new feature, PATCH = backward-safe fix.
- Example: 1.4.2 -> 1.5.0 adds a feature; 1.4.2 -> 2.0.0 breaks the public API.

Tag a release



REAL-WORLD EXAMPLE

```
git tag -a v1.5.0 -m "Shop API: coupon support"
git push origin v1.5.0
# GitHub Releases can attach notes + build artifacts to a tag
```

COMMON MISTAKE
Bumping only PATCH for a breaking change. Consumers pin versions expecting SemVer rules; a wrong bump silently breaks their builds.

REVISION SNAPSHOT

ASK YOURSELF	What version bump does a breaking API change require?
DO THIS	Tag a commit as v1.0.0 and create a GitHub Release from it.

14. .gitignore & Cleaning

GrowthSchool

IN SIMPLE TERMS
A .gitignore file lists files Git should never track, like build output and secrets, keeping your repo clean and safe.

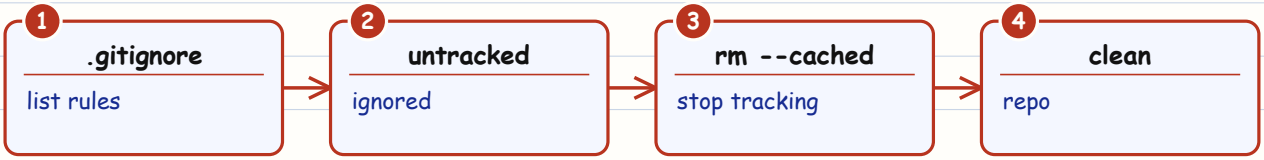
* Ignore the Right Things

- Add build output, dependencies, secrets and local config to .gitignore so they never commit.
- Common entries: node_modules/, __pycache__/, .env, dist/, *.log, .DS_Store.

* Already Tracked?

- .gitignore only affects untracked files. To stop tracking something already committed,
- run git rm --cached <file> then commit the removal.

Ignore the right files



REAL-WORLD EXAMPLE

```
# .gitignore
node_modules/
.env
*.log
dist/
# stop tracking a committed secret file:
git rm --cached .env && git commit -m "chore: stop tracking .env"
```

COMMON MISTAKE
Committing a .env or credentials file, then just adding it to .gitignore. The secret is still in history - you must rotate the secret and scrub history (git filter-repo).

REVISION SNAPSHOT

ASK YOURSELF	Why does adding a tracked file to .gitignore not remove it from the repo?
DO THIS	Add a .gitignore, then untrack an accidentally committed log file.

15. Remotes: fetch, pull, push

GrowthSchool

IN SIMPLE TERMS
A remote is a shared copy of your repo on a server (like GitHub); fetch, pull, and push move commits between your machine and it.

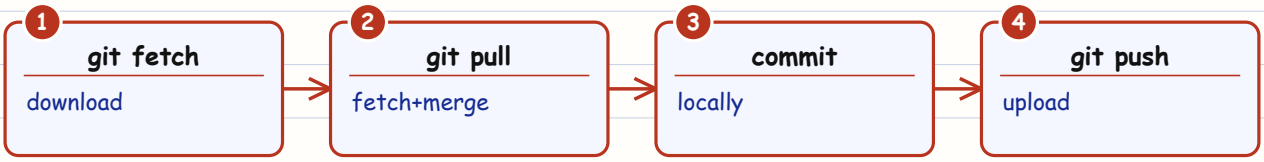
* Remotes

- A remote is a named URL to a shared repo. origin is the default created by clone.
- git remote -v lists remotes; git remote add upstream <url> adds another (common in forks).

* Sync Direction

- git fetch downloads remote changes without touching your files. git pull = fetch + merge.
- git push uploads your commits. Set upstream once with git push -u origin <branch>.

Sync with remote



REAL-WORLD EXAMPLE

```
git remote -v
git fetch origin
git pull --rebase origin main    # linear history on pull
git push -u origin feature/checkout-api
```

COMMON MISTAKE
Blindly git pull creating surprise merge commits. Prefer git pull --rebase for a clean linear history, and always fetch before you assume you are up to date.

REVISION SNAPSHOT

ASK YOURSELF	What is the difference between git fetch and git pull?
DO THIS	Add a second remote (upstream) and fetch from it without merging.

16. GitHub & Collaboration

GrowthSchool

IN SIMPLE TERMS
GitHub is a website that hosts Git repositories and adds teamwork features like issues, reviews, and access control on top of Git.

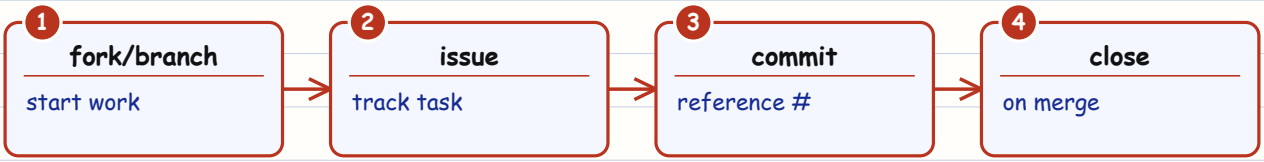
* What GitHub Adds

- Hosting plus pull requests, code review, issues, projects, Actions (CI/CD) and fine-grained access.
- Issues track work; labels, milestones and projects organise it; discussions handle Q&A.

* Fork vs Branch

- Team members with write access branch inside the same repo.
- Outside contributors fork the repo, push to their fork, then open a PR back to the original.

Collaborate on GitHub



REAL-WORLD EXAMPLE

```
# authenticate and work with GitHub from the CLI
gh auth login
gh repo clone org/shop-api
gh issue create --title "Cart total wrong with coupons"
```

STUDENT LAB
Create a GitHub repo for a small Shop API, add a README and an issue describing one feature, then reference that issue number in a commit message (e.g. "fixes #1").

REVISION SNAPSHOT

ASK YOURSELF	When do you fork instead of branching directly?
DO THIS	Open an issue, then close it automatically via a commit that says "closes #N".

17. Pull Requests & Code Review

GrowthSchool

IN SIMPLE TERMS
A pull request (PR) proposes merging your branch into another and opens it for teammates to review before it is accepted.

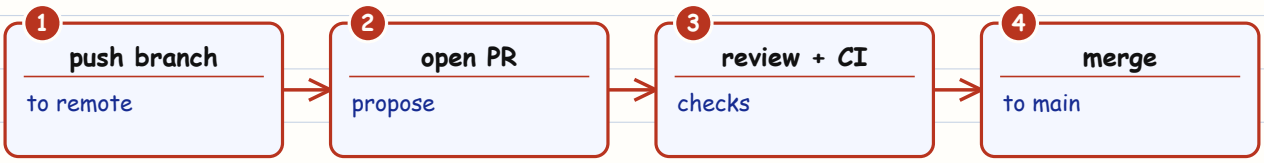
* The PR

- A pull request proposes merging your branch into a target branch and opens it for review.
- Keep PRs small and focused; add a clear description of what changed and why, plus how to test.

* Good Review Culture

- Reviewers check correctness, readability, tests and security - not personal style preferences.
- Respond to comments, push follow-up commits, and let CI pass before requesting final approval.

Open a pull request



REAL-WORLD EXAMPLE

```
git push -u origin feature/checkout-api
gh pr create --base main --title "feat: coupon support" \
  --body "Adds coupon discount. Tests added. Closes #12."
gh pr status
```

COMMON MISTAKE
Massive PRs touching 50 files. They get rubber-stamped, hide bugs, and are impossible to review well. Split work into small, reviewable PRs.

REVISION SNAPSHOT

ASK YOURSELF	What makes a pull request easy (or hard) to review?
DO THIS	Open a PR from a feature branch and request review from a teammate.

18. Branch Protection & CODEOWNER

GrowthSchool

IN SIMPLE TERMS
Branch protection and CODEOWNERS are GitHub rules that stop unreviewed or broken code from reaching your main branch.

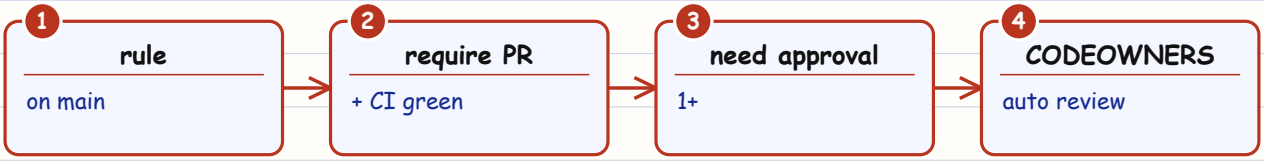
* Protect main

- Branch protection rules block direct pushes and require PRs, passing checks and approvals.
- This keeps the release branch healthy and enforces the review process automatically.

* CODEOWNERS

- A CODEOWNERS file auto-requests specific reviewers when matching paths change.
- Example: /infra/ requires the platform team; /payments/ requires the payments team.

Protect main



WHY IT MATTERS
Rules > good intentions. Automated protection ensures no unreviewed or failing code reaches main,
even under deadline pressure when humans cut corners.

REAL-WORLD EXAMPLE

```
# .github/CODEOWNERS
*                @shop/maintainers
/payments/       @shop/payments-team
/infra/          @shop/platform-team
```

REVISION SNAPSHOT

ASK YOURSELF	What three checks would you require before allowing a merge to main?
DO THIS	Enable "require PR + passing CI + 1 approval" on a test repo.

19. Conventional Commits

GrowthSchool

IN SIMPLE TERMS
Conventional Commits is a simple standard for writing commit messages (like "feat:" or "fix:") that machines can read to automate versioning.

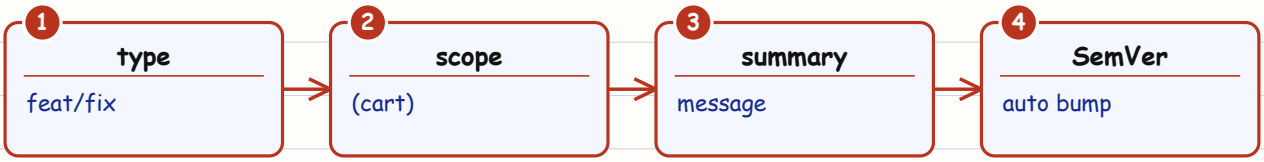
* A Commit Message Standard

- Format: type(scope): summary. Types include feat, fix, docs, refactor, test, chore, ci.
- Example: feat(checkout): apply coupon discount to cart total.

* Why Bother

- Machine-readable messages enable automatic changelogs and SemVer version bumps.
- feat -> MINOR bump, fix -> PATCH bump, and a "BREAKING CHANGE:" footer -> MAJOR bump.

Conventional commits



REAL-WORLD EXAMPLE

```
feat(cart): support multiple coupons
fix(auth): reject expired session tokens
refactor(api): extract price calculator
feat(api)!: rename /v1/checkout to /v2/checkout

BREAKING CHANGE: clients must use /v2/checkout
```

STUDENT LAB
Rewrite five of your recent commit messages in Conventional Commit format and identify which SemVer bump each one implies.

REVISION SNAPSHOT

ASK YOURSELF	Which commit type triggers a MINOR vs a PATCH release?
DO THIS	Write one feat, one fix and one breaking-change commit correctly.

20. Branching Strategies

GrowthSchool

IN SIMPLE TERMS
A branching strategy is the agreed team pattern for how branches are created, reviewed, and merged - such as feature branches or trunk-based.

* Feature Branch (recommended start)

- Short-lived branch per change -> PR -> review -> merge to main. Simple and effective for most teams.

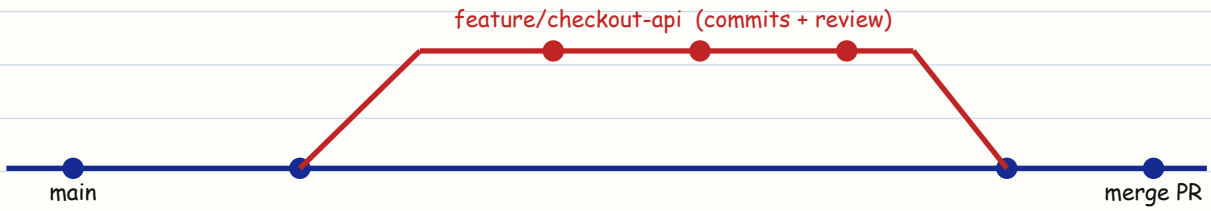
* GitFlow

- Long-lived develop + release + hotfix branches. Powerful but heavy; suits scheduled releases.

* Trunk-Based

- Everyone commits small changes to main behind feature flags with strong CI. Enables fast delivery.

Branch & Pull-Request Flow



COMMON MISTAKE
Adopting heavy GitFlow for a tiny team that ships continuously. Match the strategy to your release cadence and team size - complexity is not maturity.

REVISION SNAPSHOT

ASK YOURSELF	Which strategy fits a team that deploys many times per day?
DO THIS	Describe your current project's branching model and one way to simplify it.

21. GitHub Actions Fundamentals

GrowthSchool

IN SIMPLE TERMS
GitHub Actions is GitHub's built-in automation: it runs your build and tests automatically whenever you push code or open a PR.

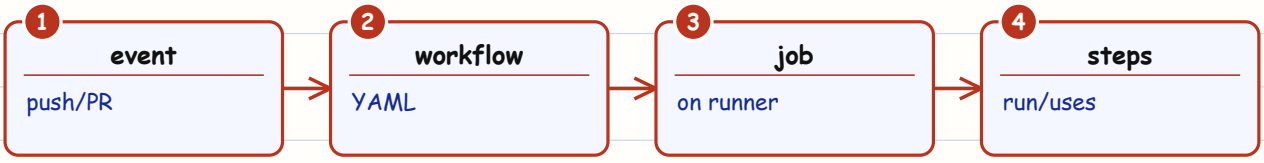
* The Building Blocks

- Workflow: a YAML file in .github/workflows. Event: what triggers it (push, pull_request, schedule).
- Job: a set of steps on a runner. Step: a shell command or a reusable action (uses:).

* Why CI

- Continuous Integration runs build + tests on every push/PR so bugs are caught in minutes, not days.
- A green check on a PR gives reviewers confidence the change actually works.

GitHub Actions parts



REAL-WORLD EXAMPLE

```
name: ci
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: echo "run build and tests here"
```

REVISION SNAPSHOT

ASK YOURSELF	Name the four core concepts: workflow, event, job, step.
DO THIS	Add a workflow that just prints "hello" on every push and watch it run.

22. CI Workflow for Shop API

GrowthSchool

IN SIMPLE TERMS
Continuous Integration (CI) means automatically building and testing every change, so bugs are caught in minutes instead of days.

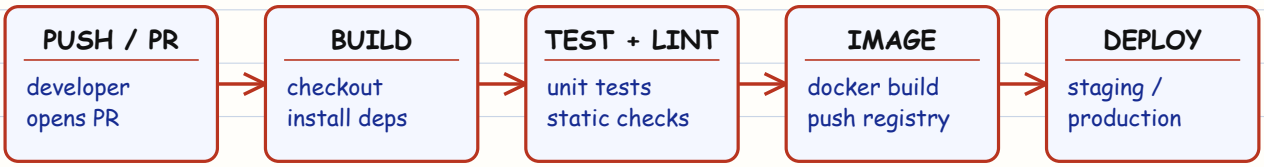
* A Realistic Pipeline

- On every PR: check out code, set up runtime, install dependencies, run lint, run tests.
- Cache dependencies to keep the pipeline fast, and fail the build if any step fails.

* Read the YAML Carefully

- Pin action versions (e.g. @v4) for reproducibility. Keep secrets in GitHub Secrets, never in YAML.

GitHub Actions CI/CD Pipeline (Shop API)



REAL-WORLD EXAMPLE

```
jobs:
  build-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with: { node-version: "20", cache: "npm" }
      - run: npm ci
      - run: npm run lint
      - run: npm test
```

REVISION SNAPSHOT

ASK YOURSELF	Why cache dependencies and pin action versions?
DO THIS	Add lint + test steps to a workflow and make a PR fail on a bad test.

23. CD & Release Automation

GrowthSchool

IN SIMPLE TERMS
Continuous Delivery (CD) means automatically shipping tested changes toward production, often with an approval step for safety.

* From Green Check to Deploy

- After tests pass on main, a deploy job can build a Docker image and push it to a registry.
- Use environments and required approvals for production; deploy staging automatically.

* Secrets & Environments

- Store credentials in GitHub Secrets or an OIDC-based cloud role - never hard-code them.
- Gate production behind a protected environment that requires a manual approval.

REAL-WORLD EXAMPLE

```
on:
  push: { branches: [main] }
jobs:
  deploy:
    environment: production   # requires approval
    steps:
      - run: docker build -t registry/shop-api:$GITHUB_SHA .
      - run: docker push registry/shop-api:$GITHUB_SHA
```

COMMON MISTAKE
Printing secrets with echo in a workflow, or committing a token to test CI. Logs are visible; use masked GitHub Secrets and rotate anything ever exposed.

REVISION SNAPSHOT

ASK YOURSELF	How do you require a human approval before a production deploy?
DO THIS	Add a deploy job that only runs on pushes to main.

24. Git Internals

GrowthSchool

IN SIMPLE TERMS
Git internals are the simple building blocks under the hood - objects and pointers - that explain why Git behaves the way it does.

* Objects

- Git stores four object types: blob (file contents), tree (directory), commit, and tag.
- Each object is content-addressed by a SHA hash, so identical content is stored once.

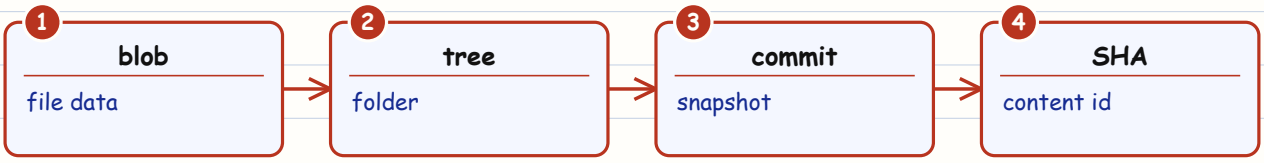
* Refs & HEAD

- A branch is a ref: a file containing a commit SHA. HEAD points to the current branch/commit.
- This is why branching is instant - it just writes a small pointer, not a copy of files.

* Why This Matters

- Understanding objects explains why commits are immutable and why history rewrites make new SHAs.
- It also explains git gc (garbage collection) removing unreachable objects after some time.

Git objects



REAL-WORLD EXAMPLE

```
git cat-file -t a1b2c3d     # type: commit
git cat-file -p a1b2c3d     # show commit contents
cat .git/HEAD               # ref: refs/heads/main
git rev-parse HEAD          # current commit SHA
git count-objects -v        # repo object stats
```

STUDENT LAB
Run git cat-file -p on a commit, then on the tree SHA it points to, then on a blob SHA.
You have now walked the object graph from commit down to file contents by hand.

REVISION SNAPSHOT

ASK YOURSELF	What are the four Git object types, and why is each commit immutable?
DO THIS	Inspect a commit object with cat-file and follow it down to a blob.

25. Undo Decision Guide

GrowthSchool

IN SIMPLE TERMS
The undo decision guide helps you pick the right, safe way to undo something based on whether your work has been shared yet.

* Ask One Question First

- Has this commit been pushed/shared? If NO, you may safely rewrite history (reset, amend, rebase).
- If YES, do not rewrite shared history - use git revert to add a safe, inverse commit.

* Pick the Right Tool

- Discard file edits -> restore. Undo last commit, keep work -> reset --soft. Undo public commit -> revert.

LOCAL / NOT PUSHED

git restore <file> : discard working change
git reset --soft HEAD~1 : keep changes staged
git commit --amend : fix last message/content
git reflog : find any lost commit

ALREADY PUSHED / SHARED

git revert <sha> : safe inverse commit
never rewrite public history casually
communicate before force operations
push --force-with-lease if truly needed

COMMON MISTAKE
Rewriting pushed history to "clean up", forcing everyone to re-sync and often losing work. Once shared, prefer revert. Reserve force-push for branches only you use.

REVISION SNAPSHOT

ASK YOURSELF	First question before any undo: has it been shared?
DO THIS	For three scenarios, name the exact command you would run.

26. Troubleshooting Playbook

GrowthSchool

IN SIMPLE TERMS
Troubleshooting is the calm, step-by-step way to fix common Git problems like rejected pushes or a detached HEAD.

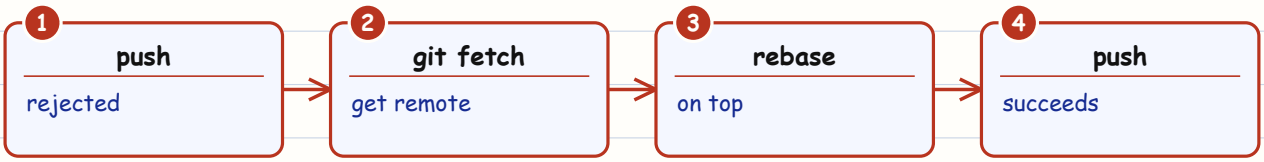
* Common Situations

- detached HEAD: you checked out a commit, not a branch. Create a branch to keep new work.
- "rejected - non-fast-forward" on push: remote has commits you lack; fetch + rebase, then push.

* More Fixes

- Accidentally committed to main: create a branch there, then reset main back with reset (if local).
- Wrong author on last commit: git commit --amend --author="Name <email>".

Fix a rejected push



REAL-WORLD EXAMPLE

```
# push rejected? sync first
git fetch origin
git rebase origin/main
git push
# detached HEAD? rescue your work:
git switch -c fix/rescue
```

COMMON MISTAKE
Force-pushing to "fix" a rejected push without looking at what is on the remote. You can overwrite a teammate's commits. Always fetch and inspect first.

REVISION SNAPSHOT

ASK YOURSELF	What causes a non-fast-forward push rejection, and how do you fix it safely?
DO THIS	Reproduce a detached HEAD and recover by creating a branch.

27. End-to-End Team Workflow

GrowthSchool

IN SIMPLE TERMS
The team workflow is the full loop real teams use: branch, commit, push, open a PR, review, merge, and release.

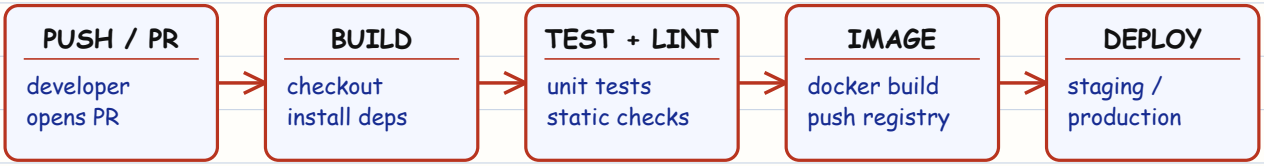
* The Shop API Flow

- Pull latest main -> create feature branch -> commit small -> push -> open PR -> CI runs.
- Review + approve -> merge to main -> tag release -> CD deploys -> monitor -> repeat.

* Keeping main Healthy

- Protected main, required checks, small PRs and fast reviews keep the team unblocked and shipping.

GitHub Actions CI/CD Pipeline (Shop API)



STUDENT LAB
Run the full loop once: branch, commit, push, PR, self-review, merge, and tag a v0.1.0 release on a practice Shop API repo. Note how long each step took.

REVISION SNAPSHOT

ASK YOURSELF	List the workflow from "pull main" to "deployed" in order.
DO THIS	Complete one full branch-to-release cycle end to end.

28. Git & GitHub Cheat Sheet

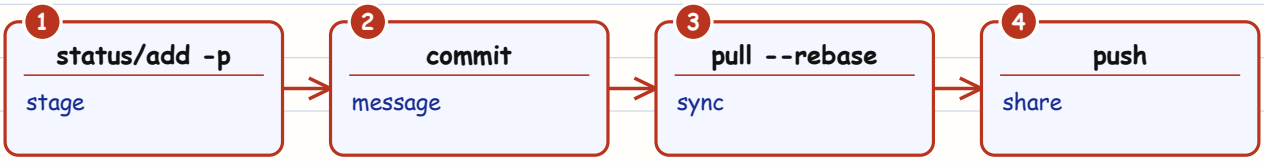
GrowthSchool

IN SIMPLE TERMS
This is a quick-reference list of the Git and GitHub commands you will use every day.

* Daily Drivers

- status, add -p, commit -m, switch -c, pull --rebase, push -u, log --oneline --graph.
- These cover 90% of daily work. Master them before reaching for anything exotic.

Everyday Git



DEBUG LOOP
status -> diff -> log --oneline --graph -> reflog. These four commands answer "what is going on?" in almost every confusing Git situation.

ONE-PAGE COMMAND REFERENCE

```
git switch -c feature/x        git restore <file>
git add -p                     git reset --soft HEAD~1
git commit -m "feat: ..."      git revert <sha>
git pull --rebase origin main  git reflog
git push -u origin feature/x   git stash push -m "wip"
git merge <branch>             git tag -a v1.0.0 -m "..."
git rebase origin/main         git cherry-pick <sha>
gh pr create                   git log --oneline --graph --all
```

COMMON MISTAKE
Memorising rare commands you will Google anyway, while fumbling the daily ones. Drill the top row until switch/add -p/commit/pull --rebase/push are pure muscle memory.

STUDENT LAB
Time yourself: branch, stage a hunk with add -p, commit with a Conventional message, rebase onto main and push - all from memory. Repeat until it takes under a minute.

REVISION SNAPSHOT

ASK YOURSELF	Which four commands orient you when Git is confusing?
DO THIS	Pin this page. Recall five commands from memory without looking.

29. Capstone Challenge

GrowthSchool

IN SIMPLE TERMS
This capstone ties everything together by shipping the Shop API using a complete Git and GitHub workflow.

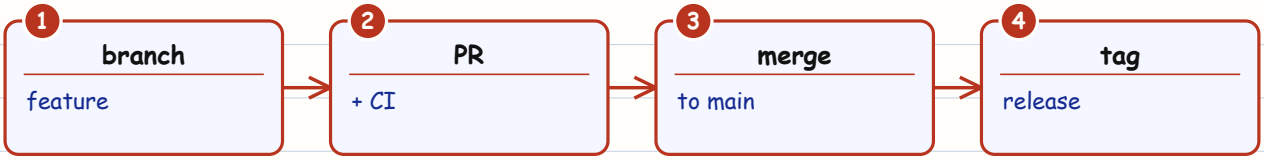
* Ship the Shop API with Git

- Init the repo, add a README and .gitignore, and make three well-scoped Conventional commits.
- Create feature/coupons, implement a change, push and open a PR with a clear description.

* Automate & Release

- Add a GitHub Actions CI workflow that runs lint + tests on every PR.
- Merge via PR, tag v1.0.0, and create a GitHub Release with notes.
- Stretch: add branch protection (CI + 1 approval) and a CODEOWNERS file.

Capstone flow



REAL-WORLD EXAMPLE

```
git switch -c feature/coupons
git commit -m "feat(cart): apply coupon discount"
git push -u origin feature/coupons
gh pr create --base main --fill
git tag -a v1.0.0 -m "Shop API first release" && git push origin v1.0.0
```

STUDENT LAB
Deliverables: a repo with protected main, a green CI check on a merged PR, a v1.0.0 tag/release, and a history that reads cleanly with git log --oneline --graph.

REVISION SNAPSHOT

ASK YOURSELF	Can your commit history be understood by a new teammate in 2 minutes?
DO THIS	Complete every capstone deliverable, then review your own history critically.

30. Interview & Revision Notes

GrowthSchool

IN SIMPLE TERMS
This page gathers the key ideas and common interview questions so you can revise quickly.

* Explain Clearly

- merge vs rebase, reset vs revert, fetch vs pull, and what a branch/HEAD actually is.
- Describe how you resolve a conflict and how you recover a lost commit with reflog.

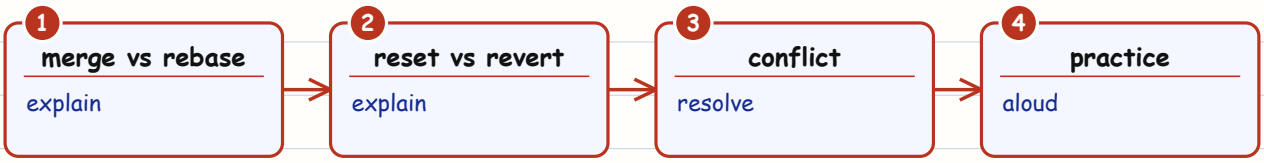
* Common Questions

- How do you undo a pushed commit? (revert) How do you tidy local history? (rebase -i).
- What is in a commit object? (tree, parent, author, message). Why is Git distributed?

* Your Next Step

- Keep a scratch repo. Every time you learn a new command, break something and fix it there.

Revise out loud



REAL-WORLD EXAMPLE

```
# rapid-fire self test - answer before running
git reset --soft HEAD~1   # vs --mixed vs --hard?
git revert <sha>          # why safe on shared branches?
git rebase -i HEAD~3      # squash/reword - local only!
git reflog                # recover after a bad reset
```

STUDENT LAB
Whiteboard the four Git areas and the commands between them, then explain merge vs rebase to a friend. If they understand it, you understand it.

REVISION SNAPSHOT

ASK YOURSELF	Can you explain reset vs revert to a beginner in 30 seconds?
DO THIS	Answer all "Common Questions" above out loud without notes.