Git and GitHub Best Practices

Diogo Ribeiro
Data Scientist Lead, Portugal
https://diogoribeiro7.github.io
https://github.com/DiogoRibeiro7

November 13, 2025

Abstract

Pragmatic Git and GitHub usage lets engineering teams ship quickly without gambling on history rewrites or risky deploys. This book distills field-tested workflows, policy templates, and recovery tactics so intermediate users can operate with senior-level discipline.

Contents

Chapter 1

Introduction: Why Version Control Discipline Matters

1.1 Chapter Overview

Discipline in version control keeps fast-moving teams from tripping over their own velocity. This chapter shows why sloppy histories cost real money, which guardrails prevent incidents, and how to tell if your repository health is drifting in the wrong direction.

1.2 From Solo Hacks to Team Delivery

A lone developer can survive with improvised workflows, but once multiple contributors share a branch every shortcut multiplies risk. Agree on branch creation, commit hygiene, pull request narratives, and validation evidence before work begins so nobody has to reverse-engineer intent during a release train.

Listing 1.1: Moving from improvisation to a reviewable workflow

```
$ git switch main
$ git pull --ff-only
$ git switch -c feat/runtime-metrics
$ git add dashboards/runtime.py flag_config.yaml
$ git commit -m "feat: add runtime metrics dashboard"
$ git push -u origin feat/runtime-metrics
$ gh pr create --title "feat: runtime metrics" --body-file .github/pull_request_template.md
```

1.3 Guardrails that Fail Fast

Automation protects the team from its own urgency. Branch protection, required checks, and signed commits block dangerous merges before customers feel the blast radius. Without guardrails, "just this once" force pushes become habit and recovering lost work dominates sprint time.

Listing 1.2: Enforcing protections with the GitHub CLI

```
$ cat > .github/main-protection.json <<'JSON'
{
  "required_status_checks": {
```

```
    "strict": true,
    "contexts": ["ci"]
  },
  "enforce_admins": true,
  "required_pull_request_reviews": {
    "dismiss_stale_reviews": true,
    "required_approving_review_count": 2
  },
  "required_linear_history": true,
  "restrictions": null
}
JSON
$ gh api \
    -X PUT \
    -H "Accept: application/vnd.github+json" \
    repos/org/app/branches/main/protection \
    --input .github/main-protection.json
```

1.4 Scenario: Hotfix Fallout

A payments team force-pushed a hotfix directly to `main`. The patch worked, but the push rewound three approved commits that had already rolled through staging. Customers were double charged, the on-call team spent hours rebuilding history from reflog entries, and finance issued refunds for a preventable incident. The fix was simple: a protected hotfix branch, documented approval steps, and a recovery checklist.

Listing 1.3: Post-incident recovery runbook

```
$ git fetch origin
$ git checkout -b incident/recover-main origin/main@{1}
$ git cherry-pick 4f2c1ab 7ac923e
$ git push origin incident/recover-main
$ gh pr create --title "Restore commits lost to hotfix" --body "Recovered from reflog; includes QA trans
```

Exercises

1. Document the last Git-related incident in your team and identify the missing guardrail.
2. Run `git status -sb`, `git branch -vv`, and `gh run list -limit 5` on your flagship repo; note surprises.
3. Draft a force-push policy covering who may do it, notification steps, and recovery expectations.
4. Export current branch protection rules via `gh api` and diff them against your documented policy.
5. Build a script that highlights pull requests lacking CI results after 30 minutes.
6. Interview product or support partners about how Git discipline affects their release confidence.
7. Propose quantitative signals (lead time, rollback count, branch age) that indicate version-control drift.

Chapter 2

Mental Models for Git

2.1 Chapter Overview

Git is a content-addressable database where immutable commits form a graph and branches are movable pointers. Once you see history as math instead of magic, recovery and collaboration become predictable.

2.2 Commits, References, and the DAG

Commits form a directed acyclic graph (DAG). Branches, tags, and `HEAD` merely name specific nodes. Rebases create new nodes; merges add nodes with multiple parents. This mental model explains why history rewrites must be coordinated and why reflog recovery works.

Listing 2.1: Interrogating the commit graph

```
$ git log --graph --decorate --oneline --all | head -n 12
$ git for-each-ref --format='%(refname:short) %(objectname:short)' refs/heads
$ git merge-base feature/payments release/2024-06
```

2.3 Working Tree, Index, and Object Database

Git tracks three zones: working tree (files on disk), index (staged changes), and object database (commits). Confusing them produces noisy diffs and broken rollbacks. Stage intentionally, inspect cached diffs, and verify what you are about to publish.

Listing 2.2: Keeping zones aligned deliberately

```
$ git status -sb
$ git add -p src/session.py
$ git diff --cached
$ git commit -m "fix: expire sessions on logout"
```

2.4 Scenario: Detached HEAD Without Panic

During a bisect, an engineer checked out a specific commit, forgetting it detaches HEAD. Instead of panicking, they created a branch pointing to that node, committed their findings, and moved on. Detached HEAD is just a pointer state, not a disaster.

Listing 2.3: Salvaging work from a detached HEAD

```
$ git checkout 1f2e8c1   # HEAD detached
$ vim metrics/job.py
$ git switch -c spike/job-metrics
$ git commit -am "spike: prototype job metrics"
$ git push -u origin spike/job-metrics
```

Exercises

1. Draw the DAG for your last merge, labeling branch pointers before and after.
2. Explain `HEAD~2` versus `HEAD^` using real commits from your repo.
3. Practice `git add -p` until you can stage half a file without rereading documentation.
4. Simulate a detached HEAD, create a branch from it, and push the branch for safekeeping.
5. Use `git merge-base` to find where your feature diverged from `main` and note the SHA.
6. Inspect `git reflog -date=iso | head -n 5` and annotate which entries correspond to resets or rebases.
7. Teach this mental model to a teammate and record open questions.

Chapter 3

Personal Git Hygiene

3.1 Chapter Overview

Your laptop is the first place Git fails. Harden it with repeatable dotfiles, safety nets, and bootstrap scripts so crashes, lost terminals, or compliance audits become inconveniences instead of outages.

3.2 Opinionated Dotfiles and Aliases

Treat dotfiles as code. Store `/.gitconfig`, hooks, and helper scripts in a private repository so every machine shares safe defaults. Include commit signing, color hints, and hunk-friendly aliases.

Listing 3.1: Minimal but opinionated /.gitconfig

```
[user]
  name = Jane Developer
  email = jane@example.com
[commit]
  gpgsign = true
[alias]
  st = status --short --branch
  lg = log --graph --decorate --oneline
[color]
  ui = auto
```

3.3 Local Safety Nets

Reflogs, stashes, and backup branches let you experiment without fear. Treat destructive commands (like `git clean -fd`) as safe only when you can quickly rewind via reflog checkpoints.

Listing 3.2: Capturing WIP before risky commands

```
$ git reflog --date=iso | head -n 4
$ git stash push -m "wip: parser refactor"
$ git reset --hard origin/main
$ git stash pop
$ git branch backup/wip HEAD@{1}
```

3.4 Scenario: Laptop Rebuild in Under an Hour

A data scientist's laptop failed mid-sprint. Because they used managed dotfiles, password-vaulted SSH keys, and checklist-driven bootstrap scripts, they were committing again within an hour. Without hygiene, the outage would have stolen days of throughput.

Listing 3.3: Bootstrapping a fresh workstation

```
$ git clone git@github.com:jane/dotfiles.git ~/.dotfiles
$ ~/.dotfiles/bootstrap.sh
$ ssh-keygen -t ed25519 -C "jane@example.com"
$ gh auth login --hostname github.com --web
$ git clone git@github.com:org/app.git
```

Exercises

1. List every location where Git is configured on your machine (system, global, repo) and reconcile inconsistencies.
2. Reinstall your dotfiles on a disposable VM and document anything that still requires manual work.
3. Enable commit signing and verify GitHub labels a pushed commit as "Verified".
4. Practice recovering from an intentional `git reset -hard` using the reflog.
5. Normalize `core.autocrlf` and `core.ignorecase` across your team.
6. Publish a short guide describing the aliases you rely on and why.
7. Schedule quarterly reminders to rotate SSH/GPG keys and test new hardware onboarding.

Chapter 4

Commits: Units of Change

4.1 Chapter Overview

Commits are the smallest units reviewers approve, release scripts package, and incident responders blame. Well-formed commits isolate behaviors, document motivation, and provide evidence of testing.

4.2 Crafting Focused Commits

Group code by behavior, not file extension. If a change touches schema, application logic, and documentation, split it into separate commits so reviewers can reason about one concept at a time.

Listing 4.1: Building a feature with narrative commits

```
$ git add migrations/2024_05_add_feature.sql
$ git commit -m "feat: add raw_score column"
$ git add app/services/score.py
$ git commit -m "feat: compute raw score using new column"
$ git add feature_flags.yaml
$ git commit -m "chore: guard raw score rollout"
```

4.3 Commit Message Quality

Messages explain intent. **Bad:** "fix stuff". **Good:** "fix: clamp retry interval under 5 minutes to meet SLA". Use templates or Conventional Commits to encode motivation, validation, and risk.

Listing 4.2: Applying a commit template

```
$ cat .git/commit_template
Summary (imperative):
Motivation:
Testing:
$ git config commit.template .git/commit_template
$ git commit
```

4.4 Scenario: Repairing History Before Sharing

A teammate stacked five WIP commits with failing tests. Instead of asking reviewers to "ignore the first four", they used fixup commits plus `-autosquash` to rewrite history before publishing, saving reviewers time and preserving bisect-friendly history.

Listing 4.3: Polishing a branch before pushing

```
$ git commit --fixup 9ac3baf
$ git commit --fixup 9ac3baf
$ git rebase -i --autosquash origin/main
$ git push --force-with-lease origin feat/runtime-metrics
```

Exercises

1. Rewrite a recent multi-file commit into focused slices and compare the review conversation.
2. Configure a commit template and enforce its use in your primary repo.
3. Practice creating fixup commits and autosquashing before pushing.
4. Explain to a teammate when `git commit -amend` is safe versus when it becomes destructive.
5. Produce a bad-versus-good commit message pair for a bug you recently fixed.
6. Create a feature flag branch that merges daily even though the feature remains disabled.
7. Add automated lint checks that fail CI when commit messages violate your format.

Chapter 5

Branching Strategies

5.1 Chapter Overview

Branch policy determines merge risk, conflict frequency, and release cadence. Whether you embrace trunk-based development or regulated release trains, the strategy must minimize drift and clarify responsibilities.

5.2 Trunk-Based with Short-Lived Branches

Trunk-based teams keep `main` releasable and rely on feature flags to hide incomplete work. Branches should live for days, not weeks, and CI must stay green or merges stop.

Listing 5.1: Trunk-based flow with feature flags

```
$ git switch main
$ git pull --ff-only
$ git switch -c feat/alerts-toggle
$ git add flag_config.yaml alerts.py
$ git commit -m "feat: add alert suppression flag"
$ git push -u origin feat/alerts-toggle
$ gh pr merge --squash --auto feat/alerts-toggle
```

5.3 Git Flow in Regulated Teams

Highly regulated teams need long-lived release branches for audits. Automate merges between `develop`, `release/x.y`, and `main` so fixes propagate deterministically and no branch becomes the "real" source of truth.

Listing 5.2: Synchronizing develop and release branches

```
$ git switch develop
$ git pull
$ git switch -c release/2024.06
$ git push -u origin release/2024.06
$ git switch release/2024.06
$ git merge --no-ff develop
$ git push origin release/2024.06
$ git switch develop
```

```
$ git merge --no-ff release/2024.06
$ git push origin develop
```

5.4 Environment Branches Are Anti-Patterns

Branches named `staging` or `qa` collect unreviewed fixes and confuse incident responders. Replace them with deployment tags or environment-specific workflows tied to the real source branch.

Listing 5.3: Deploying via tags instead of environment branches

```
$ git tag -a staging-2024-05-18 -m "Deploy build 345 to staging"
$ git push origin staging-2024-05-18
$ gh workflow run deploy.yml -f environment=staging -f ref=staging-2024-05-18
```

Exercises

1. Inventory branch types in your org and document expected lifetime plus naming conventions.
2. List remote branches older than 30 days and decide which to prune or archive.
3. Design a feature-flag rollout plan that keeps `main` releasable even when work is half-done.
4. Diagram how fixes move between `develop`, `release/x.y`, and `main` today; highlight bottlenecks.
5. Replace one environment branch with a tag-driven deployment in a sandbox repo.
6. Compare squash merges and merge commits for trunk-based work and justify your default.
7. Set up monitoring for branch age and open-PR lifetime to spot stagnation early.

Chapter 6

Merging, Rebasing, and Conflict Resolution

6.1 Chapter Overview

Merging is diplomacy plus Git mechanics. Know when to rebase, when to merge, and how to resolve conflicts calmly so history stays readable and teammates keep trust.

6.2 Choosing Merge or Rebase

Use `git rebase` to tidy private work before sharing; use `git merge` when you need a durable record of integration. Communicate before rewriting public branches and always prefer `-force-with-lease` over `-force`.

Listing 6.1: Contrasting merge and rebase flows

```
$ git switch feat/new-router
$ git fetch origin
$ git rebase origin/main
$ git switch main
$ git merge --no-ff feat/new-router
$ git push origin main
```

6.3 Conflict Management Workflow

Conflicts are inevitable; chaos is optional. Fetch everything, reproduce the merge locally, inspect blame/history for context, rerun tests, and narrate tricky decisions in the pull request.

Listing 6.2: Structured conflict resolution checklist

```
$ git fetch --all
$ git rebase origin/main
$ git status
$ git log -S"retry_interval" -- src/jobs/scheduler.py
$ git mergetool
$ pytest jobs/tests/test_scheduler.py
$ git add src/jobs/scheduler.py
$ git rebase --continue
```

6.4 Scenario: Broken History After a Rebase

A shared branch was rebased without warning, then pushed with `-force`. Teammates lost commits until reflog pointers rescued them. The team now announces rewrites in chat, uses `-force-with-lease`, and keeps a backup branch before rewrites.

Listing 6.3: Recovering overwritten commits via reflog

```
$ git fetch origin
$ git checkout feature/shared
$ git reflog show feature/shared | head -n 5
$ git branch backup/rewrite feature/shared@{1}
$ git push origin backup/rewrite
$ git push --force-with-lease origin feature/shared
```

Exercises

1. Define when your team should use merge commits versus rebases and socialize it.
2. Resolve a contrived conflict using both CLI and GUI tools; record what felt faster.
3. Simulate overwriting teammate commits and recover using reflog plus `-force-with-lease`.
4. Write a pre-push hook that warns when you're about to force-push a protected branch.
5. Compare audit trails between merge commits and rebased histories for the same feature.
6. Draft a runbook for handling conflict-heavy rebases in your repo.
7. Track how often conflicts occur and flag hotspots so architecture can address them.

Chapter 7

Working with Remotes and Forks

7.1 Chapter Overview

Remote configuration determines whether forks stay current or turn into merge nightmares. Know which remote is authoritative, how to sync safely, and how to mirror repositories for compliance without risking divergence.

7.2 Upstream Sync

Forks fall behind quickly unless you routinely pull from the canonical repository. Automate syncing so your pull requests only contain intentional work.

Listing 7.1: Keeping a fork aligned with upstream

```
$ git remote add upstream git@github.com:org/app.git
$ git fetch upstream
$ git switch main
$ git merge --ff-only upstream/main
$ git push origin main
```

7.3 Collaborating via Forks

When contributors lack direct write access, forks plus pull requests enable safely audited contributions. Keep branches narrow, mention maintainers early, and reference issues for context.

Listing 7.2: Submitting work from a fork

```
$ git switch -c feat/api-timeouts
$ git commit -am "feat: add per-client timeout config"
$ git push --set-upstream origin feat/api-timeouts
$ gh pr create --title "feat: customizable timeouts" --body "Problem, solution, validation"
```

7.4 Mirrors and Read-Only Remotes

Some enterprises mirror GitHub repositories into internal Git servers for analytics or legal archiving. Treat mirrors as read-only replication targets and push only to the primary remote to avoid split histories.

Listing 7.3: Updating a read-only mirror

```
$ git clone --bare git@github.com:org/app.git
$ cd app.git
$ git remote add mirror ssh://git.corp/mirrors/app.git
$ git push --mirror mirror
```

Exercises

1. List every remote in your flagship repository and identify who owns each.
2. Automate fork syncing with a scheduled GitHub Action or cron job.
3. Practice submitting a pull request from a fork with zero write access.
4. Configure a read-only mirror in a sandbox and verify pushes to it fail.
5. Document when developers may use `-force-with-lease` versus when to open a new branch.
6. Evaluate whether stale forks should be archived and propose a review cadence.
7. Inspect remote URLs for SSH vs HTTPS usage and align with company policy.

Chapter 8

GitHub Repository Hygiene

8.1 Chapter Overview

Repository settings drift as teams grow. Regular audits keep permissions tight, templates consistent, and automation helpful rather than noisy.

8.2 Repository Settings and Permissions

Require reviews, enforce linear history where appropriate, and restrict who can push to protected branches. Document break-glass procedures and log every exception.

Listing 8.1: Auditing and updating repository settings

```
$ gh repo view org/app --json name,defaultBranchRef,visibility
$ gh api repos/org/app -X PATCH \
    -F allow_merge_commit=false \
    -F allow_rebase_merge=false \
    -F allow_squash_merge=true \
    -F delete_branch_on_merge=true
$ gh api repos/org/app/branches/main/protection
```

8.3 Templates, CODEOWNERS, and Defaults

Templates encode expectations for issues, pull requests, and security disclosures. CODEOWNERS ensures the right reviewers see changes immediately, keeping review latency low.

Listing 8.2: Repository hygiene files

```
# CODEOWNERS
src/api/ @backend-core
infra/terraform/ @sre-team
*.md @docs

# .github/pull_request_template.md
## Context
- Problem:
- Solution:
```

```
## Validation
- [ ] Tests
- [ ] Monitoring links
```

8.4 Automations and Bot Etiquette

Bots such as Dependabot, secret scanning, and enforcement bots keep repositories healthy. Tune them carefully: noise is a configuration bug, not an excuse to disable automation.

Listing 8.3: Dependabot configuration excerpt

```
version: 2
updates:
  - package-ecosystem: pip
    directory: /
    schedule:
      interval: weekly
    reviewers:
      - backend-core
  - package-ecosystem: github-actions
    directory: /
    schedule:
      interval: monthly
```

Exercises

1. Export repository settings via the GitHub API and compare them to policy documents.
2. Draft a break-glass procedure for temporarily relaxing protections during incidents.
3. Update CODEOWNERS so every critical path has at least two reviewers.
4. Implement or revise issue templates that collect reproduction steps and rollback plans.
5. Enable Dependabot and secret scanning in a sandbox, then trigger each to verify alert routing.
6. Decide which merge strategies to allow (merge, squash, rebase) and justify the choice.
7. Audit third-party GitHub Apps installed on the repo and remove unused ones.

Chapter 9

Pull Requests and Code Review

9.1 Chapter Overview

Pull requests are change proposals, not raw diffs. They tell the story of a problem, solution, and verification so reviewers can focus on risk instead of archaeology.

9.2 Narrative Pull Requests

Explain the why before the how. Include validation evidence, screenshots, log snippets, and links to dashboards so reviewers can judge risk quickly.

Listing 9.1: Creating a narrative pull request

```
$ gh pr create \
    --title "feat: async ingestion pipeline" \
    --body $'Problem: queue backpressure at 50k msg/min.\nSolution: shard workers by tenant.\nValidatio
```

9.3 Review Checklists and Expectations

Shared checklists align reviewers on architecture, tests, observability, and rollback plans. Without them, reviews devolve into style debates.

Listing 9.2: Reviewer checklist fragment

```
- [ ] Design matches architecture docs
- [ ] Tests cover happy + sad paths
- [ ] Observability updated (logs, metrics, alerts)
- [ ] Rollback or feature flag documented
- [ ] Security/privacy implications addressed
```

9.4 Handling Feedback and Iteration

Treat comments as collaboration. Push incremental commits so reviewers can focus on deltas, and announce rebases before force-pushing.

Listing 9.3: Addressing review feedback with fixup commits

```
$ git add processors/ingest.py
$ git commit --fixup 1f2d3ab
$ git add docs/runbook.md
$ git commit --fixup 1f2d3ab
$ git rebase -i --autosquash origin/main
$ git push --force-with-lease origin feat/async-ingest
```

Exercises

1. Rewrite a past pull request body to emphasize problem, approach, and validation explicitly.
2. Add a reviewer checklist to your PR template and gather feedback after a week.
3. Practice responding to feedback by pushing incremental commits rather than rewriting history.
4. Measure review lead time (creation to merge) and discuss bottlenecks with your team.
5. Role-play a review session with one engineer defending a risky change and another probing for risk.
6. Configure required reviewers for sensitive paths through CODEOWNERS and verify enforcement.
7. Build a notification that pings you when PRs waiting on your review exceed 24 hours.

Chapter 10

Continuous Integration and GitHub Actions

10.1 Chapter Overview

CI catches bugs before customers do. Fast, deterministic pipelines encourage compliance; flaky or slow ones tempt developers to bypass safeguards. Treat workflows as code and enforce budgets.

10.2 Designing Fast, Reliable Pipelines

Parallelize linting, unit, and integration suites. Cache dependencies, fail fast on formatting errors, and set clear time budgets for each job so performance regressions surface early.

Listing 10.1: Lean CI job with caching

```
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v4
        with:
          python-version: '3.11'
      - uses: actions/cache@v4
        with:
          path: ~/.cache/pip
          key: pip-${{ hashFiles('requirements.txt') }}
      - run: pip install -r requirements.txt
      - run: pytest -q
```

10.3 Workflows as Code and Reuse

Store workflows under version control, review them like any other code, and use reusable workflows (`workflow_call`) to keep multiple repositories consistent.

Listing 10.2: Invoking a reusable workflow

```
name: service-ci
```

```
on:
  pull_request:
jobs:
  main:
    uses: org/.github/.github/workflows/python-ci.yml@v2
    with:
      pytest-marker: service
    secrets: inherit
```

10.4 Secrets and Environments

Never hardcode credentials inside workflow files. Use GitHub environments with required reviewers for production deploys and rotate secrets proactively.

Listing 10.3: Managing secrets for workflows

```
$ gh secret set DOCKER_PASSWORD --org org --body "$(pass show docker/password)"
$ gh api -X PUT repos/org/app/environments/prod/protection \
    -F required_reviewers[0][type]=User \
    -F required_reviewers[0][id]=123456 \
    -F wait_timer=15
```

Exercises

1. Measure the longest-running workflow in your repo and propose a runtime budget.
2. Split an existing pipeline into parallel jobs and compare runtimes before and after.
3. Create a reusable workflow in a central repo and call it from at least two services.
4. Rotate a GitHub Actions secret and document how you validated the change.
5. Configure staging and production environments with required reviewers; rehearse approvals.
6. Analyze the last ten CI failures and categorize them (flake, infra, product bug).
7. Pin action versions (e.g., `@v4`) and record the audit for security reviews.

Chapter 11

Release Management and Tags

11.1 Chapter Overview

Releases translate commits into artifacts customers can trust. Consistent tagging, changelog generation, and hotfix procedures keep launches boring and auditable.

11.2 Semantic Versioning in Practice

Agree on what constitutes MAJOR, MINOR, and PATCH changes. Automate version bumps so marketing, product, and SRE speak the same language about risk.

Listing 11.1: Automating semantic version bumps

```
$ npm run release -- --release-as minor
$ git push origin main
$ git push origin v2.4.0
```

11.3 Annotated Tags and Changelogs

Annotated tags carry context for auditors and incident responders. Lightweight tags lack metadata, making it harder to reconstruct what shipped.

Listing 11.2: Creating annotated tags and releases

```
$ git tag -a v2.4.0 -m "Release v2.4.0\n- feat: async ingestion\n- fix: clamp retries"
$ git push origin v2.4.0
$ gh release create v2.4.0 --notes-file CHANGELOG.md --target main
```

11.4 Scenario: Coordinated Hotfix

When production alarms fired, the team branched from the last tag, cherry-picked the fix, ran targeted tests, then merged the hotfix back into both release and main. Because the playbook was documented, the entire event took minutes, not hours.

Listing 11.3: Hotfix branch dance

```
$ git fetch origin
```

```
$ git checkout -b hotfix/2024-05-21 v2.4.0
$ git cherry-pick 4ac9dbe
$ pytest critical/tests
$ git push origin hotfix/2024-05-21
$ gh pr create --base release/2.4 --title "hotfix: clamp retries"
$ gh pr create --base main --title "hotfix: clamp retries (main backport)"
```

Exercises

1. Define MAJOR/MINOR/PATCH criteria for your product and circulate the decision.
2. Create an annotated tag in a sandbox and verify it appears in `git describe` output.
3. Automate changelog generation from commit messages or PR labels.
4. Practice cutting a hotfix branch from the last production tag and merging it back.
5. Evaluate whether your release process supports dry runs; propose improvements if not.
6. Document a rollback plan tied to tags for your primary service.
7. Track lead time from merge to release and set a goal to reduce it.

Chapter 12

Security, Secrets, and Compliance

12.1 Chapter Overview

Repositories often contain business logic, infrastructure code, and credentials. Treat Git as part of your security boundary: enforce commit signing, scan for secrets, and archive audit evidence before regulators ask.

12.2 Signed Commits and Verification

Signed commits prove provenance and deter impersonation. Require signing on protected branches and bake key setup into onboarding.

Listing 12.1: Configuring signed commits end-to-end

```
$ gpg --full-generate-key
$ gpg --list-secret-keys --keyid-format=long
$ git config --global user.signingkey ABCDEF1234567890
$ git config --global commit.gpgsign true
$ gpg --armor --export ABCDEF1234567890 > pubkey.asc
$ gh api user/gpg_keys -X POST -F armored_public_key=@pubkey.asc
```

12.3 Secret Scanning and Rotation

Secrets inevitably leak unless you scan locally, in CI, and via GitHub's native detectors. When a secret appears in history, rotate it immediately and document the response.

Listing 12.2: Integrating gitleaks locally and in CI

```
$ pre-commit install
$ cat .pre-commit-config.yaml
- repo: https://github.com/gitleaks/gitleaks
  rev: v8.18.2
  hooks:
    - id: gitleaks
$ pre-commit run --all-files
```

12.4 Compliance Evidence and Audit Trails

Auditors want immutable records. Export GitHub audit logs, archive CI results, and tag releases tied to regulatory submissions so you can answer "who approved this" without scrambling.

Listing 12.3: Archiving audit logs and release metadata

```
$ gh api orgs/org/audit-log --paginate > audit-log-$(date +%F).json
$ gh release view v2.4.0 --json tagName,name,author,createdAt > release-v2.4.0.json
$ aws s3 cp audit-log-$(date +%F).json s3://compliance-bucket/audit/
```

Exercises

1. Enable commit signing enforcement on your main repo and verify it with a test push.
2. Configure gitleaks (or similar) locally and in CI, then intentionally trigger it with a fake secret.
3. Rotate one credential discovered in history and document the timeline.
4. Export a sample audit log and store it in your compliance archive.
5. Draft a security incident response checklist referencing Git evidence (commits, tags, PRs).
6. Review ownership for every secret used in CI/CD and fill any rotation gaps.
7. Evaluate whether your repositories need GitHub Advanced Security features and budget accordingly.

Chapter 13

Advanced Workflows and Troubleshooting

13.1 Chapter Overview

Large codebases demand advanced Git tactics: sparse checkout for monorepos, automated bisection for regressions, and calm recovery drills for destructive mistakes.

13.2 Monorepo Strategies

Use sparse checkout and partial clones so developers only touch relevant paths. Combine with path-based CI filters to avoid rebuilding the entire repository for every change.

Listing 13.1: Working on part of a monorepo

```
$ git clone --filter=blob:none --sparse git@github.com:org/monorepo.git
$ cd monorepo
$ git sparse-checkout init --cone
$ git sparse-checkout set services/billing shared/libs
$ npm install --workspace services/billing
$ npm test --workspace services/billing
```

13.3 Bisecting Bugs Quickly

Git bisect halves the search space for regressions. Script the test command so every iteration is deterministic and logged for future reference.

Listing 13.2: Automated bisect run

```
$ git bisect start
$ git bisect bad HEAD
$ git bisect good v2.3.0
$ git bisect run ./scripts/smoke-test.sh
$ git bisect reset
```

13.4 Recovering from Mistakes

Wrong branch, wrong remote, destructive reset—these happen. Reflog and backup branches turn panic into procedure. Communicate early when mistakes affect shared history.

Listing 13.3: Undoing an accidental reset

```
$ git reset --hard HEAD~3   # oops
$ git reflog | head -n 5
$ git reset --hard HEAD@{1}
$ git push --force-with-lease origin feature/payment-cleanup
```

Exercises

1. Clone a large repository using `-filter=blob:none` and compare size/time to a full clone.
2. Configure sparse checkout for directories you actively modify and measure build savings.
3. Script a bisect session that runs your smoke tests automatically and logs the culprit commit.
4. Intentionally reset away local commits in a sandbox and recover them via reflog.
5. Design a playbook for responding to a destructive force push on `main`.
6. Evaluate whether your monorepo needs path-based CI filtering and outline the rollout.
7. Practice using `git worktree` to juggle multiple reviews without extra clones.

Appendix A

Checklists, Templates, and Further Reading

A.1 Chapter Overview

Checklists and templates encode hard-won lessons into repeatable process. Keep them in the repository so updates travel with the code they describe.

A.2 Daily Flow Checklist

Give developers a predictable routine that keeps workspaces clean, branches fresh, and CI friendly.

Listing A.1: Daily flow checklist

```
- Pull latest: git fetch --prune && git switch main && git pull --ff-only
- Rebase feature branches before coding
- Run lint + unit tests locally
- Update PR descriptions with validation evidence
- Clean merged branches locally and remotely
```

A.3 Template Snippets

Standardize release notes, PR templates, and incident summaries so responses stay consistent even under stress.

Listing A.2: Release note template sketch

```
## Impact
-

## Changes
-

## Validation
- [ ] Tests
- [ ] Metrics linked

## Rollback
```

```
-
```

A.4 Further Reading

Curate trusted sources so engineers know where to deepen Git and GitHub expertise.

Listing A.3: Curated reading list

```
- Pro Git (Chacon & Straub)
- GitHub Engineering Blog
- Atlassian Git Tutorials
- Internal RFC-102 Branch Protection Policy
- Internal RFC-118 CI/CD Architecture
```

Exercises

1. Create a `docs/checklists/` directory in your repo and add at least two actionable checklists.
2. Customize the daily checklist above for your stack and share it during stand-up.
3. Test the release note template in the next deployment dry run and gather feedback.
4. Assemble the reading list in your company wiki, linking to both external and internal resources.
5. Build a script that validates the presence of required templates and fails CI if they disappear.
6. Assign owners to every checklist/template and schedule quarterly reviews.
7. Add links to these resources in onboarding documentation for new engineers.