Audit & Compliance Practices in Git Projects

* **1. Commit Signing & Attribution**
  + Use GPG/SSH commit signing to verify author identity.
  + Helps in auditing changes by ensuring all commits are authentic.
  + git log --show-signature can verify signed commits.
* **2. Audit Logs (for GitHub Enterprise)**
  + GitHub Enterprise offers detailed Audit Logs.
  + Captures:
    - Sign-in attempts
    - Repository changes
    - Secret access
    - Token usage
  + Available at: https://github.com/organizations/<your-org>/settings/audit-log
* **3. Access & Role Management**
  + Use least privilege principle:
    - Admins
    - Write
    - Read
  + Enforce 2FA for contributors.
  + Use teams and role-based access control in organizations.
* **4. Token & Credential Management**
  + Rotate Personal Access Tokens (PAT) regularly.
  + Revoke unused tokens.
  + Use GitHub Secrets for automated workflows.
  + Avoid hardcoding secrets – enforce .gitignore rules and use secret scanning tools.
* **5. Dependency & Vulnerability Scanning**
  + Use GitHub's built-in:
    - Dependabot Alerts
    - Dependency Graph
  + Keeps track of known vulnerable packages.
  + Automate PRs to patch vulnerabilities.
* **6. Branch Protection & PR Reviews**
  + Enforce:
    - Branch protection rules
    - Required status checks
    - Mandatory code reviews
  + Helps maintain code quality and change traceability.
* **7. Audit Commands in Git (Locally)**
  + git log – Commit history
  + git blame <file> – Who changed each line
  + git show <commit> – Full details of a commit
  + git reflog – Reference log of changes to HEAD
* **8. Logging CI/CD Pipelines**
  + Log all build steps and environment actions.
  + Keep timestamped logs of who triggered what and when.
  + Store logs securely for compliance review.
* **9. Secret Scanning & Compliance Tools**
  + Enable GitHub Advanced Security (if available).
  + Use external tools:
    - TruffleHog
    - GitSecrets
    - Snyk
    - SonarQube
    - Integrate in CI/CD pipelines for automated scanning.
* **10. Policy Documentation**
  + Maintain a Security Policy (SECURITY.md) in your repo.
  + Include:
    - Contact for reporting vulnerabilities
    - Audit process
    - Role definitions
  + Display on GitHub under the "Security" tab.

Optimizing Git Workflows for Team Productivity

* **1. Choose the Right Workflow for Your Team**
  + Pick one and stick with it for consistency.

|  |  |  |
| --- | --- | --- |
| **Workflow** | **Best For** | **Description** |
| **Feature Branch** | Small teams | Each feature/fix in its own branch, merged via PR. |
| **Git Flow** | Large projects | Uses develop, release, hotfix, feature branches. |
| **Trunk-Based** | Fast delivery | Everyone commits to main, feature flags used. |
| **Fork-Based** | Open source | Each contributor works in their own fork; PRs for merging. |

* 2. Branch Naming Conventions
  + Use consistent naming to stay organized:
    - feature/login-page
    - bugfix/payment-crash
    - hotfix/production-db
    - release/v1.2.0
  + Tip: Prefix branches by type (feature, bugfix, etc.).
* 3. Clear Commit Messages
  + Follow a format like:
    - <type>: <short description>
    - Body (optional)
  + Examples:
    - feat: add user registration
    - fix: correct typo in login modal
    - docs: update README for install guide
  + Use tools like Commitizen to standardize this.
* 4. Use Pull Requests Effectively
  + Always use PRs for code review.
  + Assign reviewers, not just ping all.
  + Enforce:
    - Minimum number of approvals
    - Status checks
    - CI/CD green builds
  + Use draft PRs for work-in-progress visibility.
* 5. Enable Branch Protection Rules
  + Protect main or develop branches:
    - Require PRs
    - Require reviews
    - Require passing CI
    - Prevent force pushes
    - Require signed commits (optional)
* 6. Schedule Regular Merges
  + Set weekly or daily syncs:
    - Merge feature branches to develop
    - Merge develop to main when stable
  + Avoid long-lived branches to reduce merge conflicts.
* 7. Automate Repetitive Tasks
  + Use GitHub Actions or GitLab CI:
    - Linting, testing
    - Secret scanning
    - Deployments
  + Automate changelogs and releases using semantic versioning.
* 8. Pair Programming & Code Review Best Practices
  + Encourage pairing on critical changes
  + Do small, frequent PRs — easier to review
  + Review for:
    - Logic errors
    - Security issues
    - Readability
* 9. Clean Up Stale Branches
  + Delete merged branches to keep the repo clean.
    - git branch -d feature/old-branch
  + Or prune remote branches:
    - git fetch -p
* 10. Use Git Hooks for Pre-checks
  + Add pre-commit and pre-push hooks:
    - Lint code
    - Run tests
    - Prevent committing secrets
  + Tools:
    - Husky
    - lint-staged
* 11. Use .gitignore and .gitattributes
  + Prevent unnecessary files from cluttering the repo
  + Normalize line endings, treat binaries correctly
* 12. Educate and Document
  + Create a CONTRIBUTING.md with your Git practices
  + Document your Git flow, commit style, PR process

Commit Message Conventions

* Basic Format
* <type>(optional scope): <short summary>
* (optional body)
* (optional footer)

Commonly Used Commit Types

|  |  |
| --- | --- |
| **Type** | **Meaning** |
| **feat** | A new feature |
| **fix** | A bug fix |
| **docs** | Documentation only changes |
| **style** | Formatting, missing semicolons, etc. (no code logic change) |
| **refactor** | Code change that neither fixes a bug nor adds a feature |
| **perf** | Code change that improves performance |
| **test** | Adding or fixing tests |
| **chore** | Maintenance tasks (build process, tools, etc.) |
| **ci** | Changes to CI configuration files and scripts |
| **build** | Changes that affect the build system or dependencies |
| **revert** | Revert a previous commit |

Examples

* feat: add login functionality for new users
* fix: resolve null pointer exception on dashboard load
* docs: update API section in README
* style: format indentation in app.js
* refactor: extract user validation logic into a utility
* test: add unit tests for auth middleware
* chore: update dependencies and clean up package-lock.json

Why Use Conventions?

* Easy to read commit history
* Improved searching and filtering (git log --grep=fix)
* Helps with automated changelogs (e.g., with standard-version)
* Useful in release automation workflows

Tools That Enforce It

* Commitizen – prompts for standardized commits
  + npx commitizen init cz-conventional-changelog --save-dev --save-exact
* Husky + lint-staged – prevent bad commits via Git hooks

Code Review Best Practices  
For Reviewers

* 1. Understand the Context
  + Read the description and related issue/ticket.
  + Run the code locally if needed to test the change.
* 2. Review Code, Not the Developer
  + Focus on the code’s quality, clarity, and correctness — not the person who wrote it.
* 3. Be Constructive and Respectful
  + Suggest alternatives rather than demanding changes.
  + Example:
    - ❌ “This is wrong.”
    - ✅ “Could we simplify this by using a map instead?”
* 4. Check for:
  + Correctness: Does it work as intended? Any bugs or edge cases?
  + Security: Are there any vulnerabilities or insecure patterns?
  + Clarity: Is the code easy to read and understand?
  + Test Coverage: Are there adequate unit/integration tests?
  + Performance: Is there any unnecessary overhead or inefficiency?
  + Style & Consistency: Does it follow the project’s coding standards?
* 5. Limit Scope Creep
  + Keep reviews focused. Large changes should be broken into smaller PRs.

Code Review Best Practices  
For Authors

* 1. Keep Pull Requests (PRs) Small & Focused
  + Easier to review and less chance of introducing bugs.
* 2. Write a Clear Description
  + Include:
    - What changed?
    - Why it changed?
    - Any side effects or related context?
* 3. Tag Reviewers Appropriately
  + Tag team members who own that part of the code.
* 4. Write Clean Commits
  + Use conventional commit messages.
  + Keep commits logical and atomic.
* 5. Respond Graciously to Feedback
  + Engage in constructive discussion and iterate promptly.

Code Review Best Practices

* Tools That Help
  + GitHub / GitLab / Bitbucket: Inline comments, suggestions, approvals
  + Linting/Formatting Tools: Prettier, ESLint, Checkstyle, etc.
  + CI Pipelines: Ensure all tests and builds pass before review
* Final Tips
  + Say "LGTM" (Looks Good To Me) when approved
  + Use emojis or comments to show positivity
  + Encourage knowledge sharing through thoughtful reviews
  + Set team expectations for review turnaround (e.g., 24–48 hours)

Scaling Git for large teams & projects

* Scaling Git effectively for **large teams and complex projects** requires thoughtful workflows, tooling, and structure.
* 1. Adopt a Scalable Branching Strategy
  + Use a clear, team-wide branching model:
  + Choose what suits your team's velocity and deployment style.

|  |  |
| --- | --- |
| **Strategy** | **Description** |
| **Git Flow** | Good for release-driven teams, uses feature/, develop, release/, hotfix/ branches. |
| **GitHub Flow** | Simpler, based on main or master + feature branches. Ideal for CI/CD and smaller deployments. |
| **Trunk-Based Development** | Single main branch, frequent commits, heavy automation — good for advanced CI/CD. |

* 2. Modularize Your Repository
  + a. Monorepo (Single Repo for Many Projects)
    - Tools: Nx, Lerna, Bazel
    - Benefits: Easier dependency management, atomic commits
    - Challenges: CI/CD complexity, repo bloat
  + b. Multirepo
    - One repo per project or service
    - Easier to scale organizationally, but more coordination needed
* 3. Access Control and Permissions
  + Use GitHub Teams or GitLab groups
  + Restrict access by:
    - Branch protection rules
    - Code owner approvals
    - Reviewers and merge policies
* 4. Review and Merge Best Practices
  + Set up pull request templates with:
    - Description
    - Screenshots
    - Test instructions
  + Enable required reviews and status checks
* 5. Analytics & Monitoring
  + GitHub Insights, GitLab Analytics, or 3rd party tools like:
    - Pluralsight Flow
    - Waydev
    - SonarQube
  + Track:
    - PR cycle times
    - Commit frequency
    - Review quality
    - Code churn
* 6. Performance Optimizations
  + a. Shallow Clones
    - git clone --depth=1 https://github.com/your-org/large-repo.git
  + b. Sparse Checkout
    - Only check out specific directories:
    - git sparse-checkout init --cone
    - git sparse-checkout set path/to/dir
  + c. Git Large File Storage (LFS)
    - Track large assets separately:
    - git lfs install
    - git lfs track "\*.psd"
* 7. Communication & Collaboration
  + Integrate Git with:
    - Slack for PR notifications
    - Jira for linking commits to tickets
    - Conventional Commits for changelog automation
* 8. Training & Documentation
  + Document your Git policies in a CONTRIBUTING.md
  + Train new developers on:
    - Branching strategy
    - Commit style
    - Review process
    - CI/CD tools
* 9. Use Submodules or Subtrees (If Needed)
  + - Submodules: Point to specific commits of other repos. More flexible, but requires more manual effort.
    - Subtrees: Integrate one repo into another with full history.
  + Only use them if shared components are tightly coupled or versioned separately.
* 10. Enforce Standards and Automation
  + Linters and Formatters: ESLint, Prettier, Pylint
  + CI Pipelines: GitHub Actions, GitLab CI, Jenkins, CircleCI
  + Code Coverage: Enforce test coverage minimums