Git Version Control for FDA and IEC 62304 Compliance

By IntuitionLabs • 8/26/2025 • 50 min read

git | fda compliance | version control | iec 62304 | 21 cfr part 11 | medical device software | regulatory submission | traceability

Versioning Your Way to Compliance: Git Workflows for FDA Submissions

Introduction

Organizations developing **FDA-regulated products** – whether medical devices, pharmaceuticals, or digital health software – face stringent documentation and traceability requirements during regulatory submissions (e.g. 510(k), PMA for devices; IND, NDA for drugs). Compliance standards such as FDA's Quality System Regulation (QSR) and ** 21 CFR Part 11** mandate robust controls on electronic records, including **version control, audit trails, and traceability** for all changes. In parallel, international software lifecycle standards like ** IEC 62304** (medical device software) emphasize configuration management and documentation. This report explores how modern software teams can leverage **Git**, a distributed version control system, to meet these compliance demands. We will discuss FDA expectations for software development documentation, map Git's capabilities to regulatory requirements, and recommend **Git workflow best practices** (branching strategies, release tagging, commit conventions, integration with issue trackers and CI/CD) that enhance traceability and auditability. We also address techniques for preserving **immutable records** and **electronic signatures** in Git-based workflows, and provide case studies illustrating successful FDA submissions underpinned by Git versioning. The goal is to equip regulatory professionals and software engineers in regulated environments with a comprehensive guide to using Git to **streamline compliance** without sacrificing agility.

Regulatory Context: FDA Submissions and Compliance Standards

FDA regulatory submissions require extensive evidence that a product's development process was controlled and documented. For medical devices, a premarket submission (whether a **510(k)** for clearance of a class II device or a **PMA** for approval of class III) must include a **Design History File (DHF)** documenting the design and development of the device ketryx.com ketryx.com. This includes all design outputs, changes, and verification/validation activities, demonstrating compliance with FDA's design control requirements (21 CFR 820.30) ketryx.com. Notably, FDA's design control regulation explicitly requires procedures for **identifying, documenting, validating (or verifying), reviewing, and approving design changes before implementation** ecfr.gov. In practice, this means every change to the device design (including software) must be traceable and cannot be applied without proper review and approval.

For drug development, submissions like **INDs** and **NDAs** similarly demand that any software used (for example, in clinical trials or in production of a combination product) adheres to Good Clinical/Manufacturing Practice. Regulatory guidance emphasizes ** software validation** and traceability to ensure data integrity. All FDA-regulated sectors must also comply with **21 CFR Part 11** when using electronic records/signatures. Part 11 requires controls such as ** secure, computer-generated, time-stamped audit trails** that record who made changes and when, without obscuring prior entries nsflow.com. In other words, electronic records (including code and documents managed electronically) must have an audit trail so that regulators can reconstruct the history of modifications. Additionally, **access controls and unique user identities** are mandated to ensure only authorized individuals can make changes or sign records.

International standards complement these regulations. **IEC 62304:2006+A1:2015**, recognized by FDA fda.gov, defines **software life cycle processes** for medical devices, including requirements for **software configuration management** (Clause 8) and **problem resolution**. IEC 62304 calls for maintaining **configuration identification, change control, and status accounting** for all software items greenlight.guru. In plain terms, developers must **identify each software configuration item and version, control changes to them, and record the status/history of those changes**, which aligns perfectly with the capabilities of a version control system. FDA's 2023 guidance on software in device submissions explicitly allows companies to submit a **Declaration of Conformity to IEC 62304** in lieu of detailed documentation of their processes fda.gov fda.gov – underscoring that following this standard (and by extension, having solid configuration management) is key to regulatory compliance.

Moreover, FDA guidance now expects a **Software Version History** document in submissions, which is essentially a **line-item log of software versions, dates, and changes** from the point design controls were applied up to the final release fda.gov. Each entry should describe changes relative to the prior version, and the final entry corresponds to the version in the marketed device, noting any last-minute fixes and their impact fda.gov. This regulatory expectation directly speaks to the need for **comprehensive version control**: sponsors must be able to produce a complete history of software changes – something that a well-managed Git repository can readily provide.

Finally, beyond design controls and Part 11, quality standards like **ISO 13485:2016** (for medical device QMS) and guidance such as FDA's *General Principles of Software Validation* emphasize **traceability** (linking requirements to design, implementation, and testing) and **risk management** of software changes. For instance, FDA expects processes to **link user needs, system requirements, software requirements, design specifications, tests, and risk control measures** – establishing end-to-end traceability fda.gov. Any version control approach in a regulated setting should facilitate or at least not impede this traceability (for example, by enabling tagging or referencing of requirement IDs in commits). In summary, the compliance landscape sets forth clear demands: **every change to software must be controlled,**

documented, and traceable, with evidence of review/approval and no possibility of surreptitious alteration. This is the context in which Git will be evaluated as a tool.

Leveraging Git for Regulatory Compliance

Git, as a distributed version control system, offers innate features that support these compliance needs. At its core, Git tracks changes to files over time, recording each change as a **commit** with a unique cryptographic hash (SHA-1/SHA-256), the author's identity, timestamp, and a message. This forms an **immutable history** of the project's evolution: once committed and pushed to a central repository, past revisions cannot be altered without detection (due to the hash linkage). This property aligns strongly with Part 11's requirement that record changes **not obscure previously recorded information** nsflow.com. Git's history is essentially an ** audit trail** of source code development. Each commit is a timestamped record of "who did what, when," providing the kind of accountability regulators seek. As long as rewrite commands (like history editing or force pushes) are disallowed in the compliance-critical repositories, Git's append-only log of commits functions as a secure audit trail. In fact, industry guidelines for Part 11 compliance recommend using "append-only" record systems or credentials s3.amazonaws.com – a properly managed Git repository can be configured in exactly this manner, so that nothing is truly deleted or overwritten, only new commits appended.

Another advantage of Git is its support for **distributed collaboration with controlled access**. In an FDA-regulated team, developers can each have their local clone of the repository (providing redundancy and backup of the full history), but an authoritative central repository (self-hosted or cloud-based with proper validation) can be designated as the source of truth for submissions and production releases. Access controls (authentication, user permissions) can be enforced such that only authorized contributors can commit or merge changes, satisfying Part 11's requirement to **limit system access to authorized individuals** (e.g., via unique accounts and role-based privileges). Modern Git hosting platforms (GitLab, GitHub Enterprise, Bitbucket) support integration with corporate authentication and allow fine-grained permissions and **audit logs** of who pushed commits or merged pull requests. These logs, combined with Git's internal history, help demonstrate compliance with **electronic record integrity** and access control rules.

Git also inherently supports **traceability** when used with good practices. Each commit can (and should) include a message describing the change; teams can adopt **commit message conventions** to reference requirements, bug IDs, or risk items. For example, including a requirement ID in the commit message ties the code change to a specific design input or user need. This practice implements traceability linking design inputs to implementation and further to testing (if test cases also reference the requirement or commit). FDA's expectation that you link requirements to design and verification fda.gov can be facilitated by such Git conventions in combination with an issue tracking system. In fact, many organizations use an issue tracker (like Jira or Azure DevOps) alongside Git, where each issue represents a requirement, change request, or bug fix; developers mention the issue ID in commit messages (e.g., "Fix defect #123,

addressing requirement R1.2"). This creates a cross-reference between the code repository and the requirements management system. The result is that one can generate a **traceability matrix** or change impact analysis by traversing these links – a powerful demonstration of control for auditors.

From a **software lifecycle documentation** perspective, Git provides a backbone for maintaining all sorts of artifacts under version control: not just source code, but also documents (requirements specs, design documents, test protocols, etc.), or at least links to them. Storing documentation in a Git repository (written in Markdown, reStructuredText, etc.) allows the same version history and branching to apply to documents, ensuring that your procedures and specifications evolve under control. Some companies have even implemented their entire Quality Management System (QMS) documentation as Markdown files in a Git repo, using pull requests for document review and approvals. This can work, though care must be taken for **electronic signatures** on documents – a simple Git commit or PR approval may not fully meet Part 11 signature requirements without additional controls openregulatory.com openregulatory.com. Nonetheless, having documents versioned in Git ensures **old versions are retained** openregulatory.com and changes are logged – fundamental for document control in ISO 13485 and FDA QSR.

One concern in regulated use of Git is tool validation. FDA's QSR (21 CFR 820) states that **any software tool used in production or the quality system must be validated for its intended use** greenlight.guru. In plain terms, if you use Git (or an entire DevOps platform) as part of your design control process, you should demonstrate that it functions correctly and reliably in that context. This typically involves documenting the intended use (e.g., "manage source code versions and record change history for Project X"), risk-assessing the tool (Git is low risk to product quality, but a failure could lose history or allow unauthorized changes), and performing some qualification tests – for instance, verifying that the repo can only be accessed by authorized users, that history cannot be modified without trace, that backups work, etc. Many companies handle this via a **standard operating procedure (SOP)** on configuration management that includes using Git, plus vendor certification or internal testing of the Git hosting solution. FDA's recent *Computer Software Assurance (CSA)* approach encourages focusing validation effort on high-risk tools greenlight.guru greenlight.guru; a version control system might be considered medium risk, but robust configuration and standard use can mitigate most risks. In any case, showing auditors that you have **procedures for using Git** and have tested its key functions goes a long way. Git itself is a mature, widely-used tool (including in safety-critical industries), which usually gives confidence as long as you manage it properly.

In summary, Git's distributed, history-preserving, and collaborative nature can strongly support regulatory compliance by providing an **audit-trailed repository of truth** for all software development artifacts. The next sections will delve into specific **workflow practices** to maximize these benefits in an FDA-regulated environment.

Git Workflow Best Practices in Regulated Environments

Not all version control workflows are equal in the eyes of compliance. Certain Git patterns and policies can significantly enhance **traceability, auditability, and control**. Here we outline best practices for branching, merging, and releasing that align with regulatory expectations:

- **Use a Structured Branching Strategy:** A common approach in regulated projects is to adopt a variation of **GitFlow** or a **trunk-based workflow** with clearly defined branch policies. For example, a GitFlow-style model might have a long-lived `main` (or `master`) branch representing the current released codebase and a `develop` branch for integration of ongoing development. Feature branches branch off `develop` for new features or bug fixes, and release branches are created from `develop` when preparing a formal release. This structure cleanly separates **development work from released (or soon-to-be-released) code**, which is useful because **released versions must be locked down and reproducible** for submission and future auditing. In a regulated context, one might treat the `main` branch as sacred – it contains only code that has passed verification and is approved for use. All merges into `main` occur via controlled pull requests after testing. Meanwhile, **trunk-based development** (frequent small merges into main) can also be used, but it requires strong discipline: automated tests and possibly feature toggles to ensure that `main` is always in a releasable state. Trunk-based development has the advantage of simplicity and continuous integration, but in a safety-critical scenario, teams often still implement gating controls (e.g. requiring approval before a merge to main).

- **Branch Protections and Two-Person Review:** Regardless of specific branching model, a **key best practice is to protect the primary branches** (main, develop, release branches) so that no one can push changes directly to them. Instead, changes must come via pull requests (PRs) or merge requests from feature branches. This enforces the **four-eyes principle (two-person integrity)** – another person must review and approve the code change before it is integrated. In fact, an industry white paper on Part 11 compliance recommends a strict "**Git Fork & Pull Request workflow**" where developers work in isolated forked repositories or branches and create PRs to the main repo, with **no direct commits to main allowed** s3.amazonaws.com. The same source stresses that each PR undergo a senior developer's review and **digital sign-off**, implementing a formal approval step s3.amazonaws.com. This process mirrors the requirement for independent review and approval of design changes before implementation ecfr.gov – the Git platform's code review feature becomes the vehicle for that required design change review. The record of the code review (who approved and when) along with the merge commit provides evidence of compliance. Many Git hosting tools allow requiring *at least N approvers* for a PR and can even enforce that certain roles (e.g. a QA engineer or lead) must approve, which can be mapped to the idea of an electronic signature on the change. Ensure that these settings are documented in your SOP and that audit logs of PR approvals are retained.

- **Granular Feature Branches Tied to Work Items:** It's wise to create feature or bug-fix branches for each discrete change, and name the branches clearly (e.g., `feature/123-new-login-screen` or `bugfix/456-null-pointer-exception`). Often the branch name or initial commit message will include an ID referencing the requirement or issue being addressed. This practice aids traceability because one can later map a branch (and its merged commits) back to a specific change request or requirement. It also prevents unrelated changes from mingling, which simplifies impact analysis. In regulated environments, avoid giant all-encompassing commits; instead, **commit frequently** with logical units of change and descriptive messages. Each commit should ideally address one issue or requirement. Smaller commits with good messages make it easier to generate the **"brief description of all changes"** needed for the software version history documentation fda.gov. They also make code reviews more effective. Regulators don't mandate how small a commit should be, but they will appreciate the ability to follow the thread of development for each feature/fix.

- **Tagging and Releasing:** Use Git **tags** to mark all significant releases, especially those that go into testing, regulatory submissions, or production. A tag (e.g., `v1.0`, `v2.1.3-release`) is a human-friendly alias to a specific commit (which might be the merge commit on main corresponding to that version). By tagging releases, you create an immutable reference to the exact code that was submitted to FDA or deployed in the field. This is crucial for configuration management: if an issue arises or if FDA asks for an audit, you can easily retrieve the exact code version that was approved. In submissions, when providing the **software version history**, you can list the Git tag or commit ID for each version tested and released fda.gov. This gives additional confidence in the integrity of the submission: a regulator could, if desired, ask to inspect the code at that commit and verify the changes described. Some companies even include the Git commit hash in their submission documents or labeling (for instance, in release notes or the "about" info of the software) as an unambiguous identifier of the software configuration. Ensure that once a version is tagged for submission, that tag is never moved or reused – it must remain a permanent pointer to that commit to serve as an audit reference.

- **Release Branches and Patches:** In medical device software, after an initial submission and clearance, you may need to issue updates (bug fixes or minor feature changes). It's common to maintain a **release branch** for each major version (e.g., a `1.x` branch) so that urgent fixes can be made based off the released code, while new development continues on the main or develop branch. This way, a patch (say v1.0.1) can be prepared on the release branch, tested, and submitted without pulling in all the ongoing new features intended for v2.0. Git makes branching and parallel development easy, so you can maintain multiple versions. Just be sure to also tag patch releases. Maintaining separate branches for supported releases aligns with **configuration management best practices** to clearly identify which version is in which state (development vs released vs retired) greenlight.guru.

- **Avoid Rebasing or History Rewrites on Public Branches:** In non-regulated open-source projects, developers sometimes rebase or squash commits to keep history tidy. In a regulated context, however, the **history is sacrosanct** – it's your audit trail. Thus, do *not* rewrite history on any branch that has been shared or used for compliance evidence. Configure the Git server to reject force pushes on protected branches. Squashing commits into one may simplify a merge, but you lose the granular history of individual changes (and their authorship and timestamps), which could be valuable evidence. It's better to keep the full sequence of commits as-is. If a messy history is a concern, it can be addressed by planning commits better or using pull request squash *only* on feature branches before merge (and even that should be carefully considered). Remember, a regulator might question any anomalies in record keeping – a rewritten history could be viewed as tampering with records. **Immutability of records is paramount**; as one guide notes, using append-only methods for records and avoiding deletion or alteration is essential for audit trails s3.amazonaws.com.

- **Commit Signing and Developer Identity:** Git allows developers to **digitally sign commits and tags** using GPG or X.509 certificates. In a regulated environment, this can add an extra layer of assurance that commits truly came from the person indicated (and weren't altered). A signed commit includes a cryptographic signature that can be verified against the developer's public key. While not explicitly required by FDA, using signed commits or tags is a **best practice for authenticity**, akin to an electronic signature on each commit. Part 11 stipulates that electronic signatures must be unique to an individual and verifiable dev.to dev.to. A GPG signature meets the verifiability condition (it mathematically ties the commit to the signer's identity). However, to fully comply as an *FDA-recognized* e-signature, you also need policies ensuring that key ownership is unique and keys are securely managed (and typically a mapping of the key ID to the person's identity in a controlled document). Many teams simply use repository access controls and user accounts as the identity mechanism (i.e., only your account can commit under your name, and sharing accounts is prohibited), which FDA inspectors generally accept along with audit logs. But if you want to be extra cautious, commit signing provides an immutable link between identity and action – something that could satisfy even a strict Part 11 auditor. Additionally, consider signing release tags by the responsible engineer or manager; this is like countersigning the "released" version.

By implementing these workflow practices, an organization creates a controlled environment where **every code change is reviewed and traceable**, and the history cannot be inadvertently or maliciously changed. This aligns with the intent of FDA's design control and Part 11 requirements, essentially **using Git as the backbone of configuration management** and design change control.

Documentation Practices and Tool Integration for Compliance

Process alone isn't enough – how you document and integrate your use of Git with other tools greatly influences compliance. We now turn to concrete practices for documentation, commit conventions, and integrating Git with CI/CD, issue trackers, and repositories for a seamless compliance ecosystem:

- **Commit Message Conventions:** Writing clear, descriptive commit messages is always good practice, but in a regulated context it becomes critical. Each commit message should provide enough detail to understand the nature of the change and *why* it was made. A recommended convention is to include references to related artifacts: for example, **reference the requirement ID, risk ID, or test case ID** that the commit addresses. A commit might read: "Implement checksum verification on data export (Req#REQ-9); mitigates risk R-15; tested by TC-22." This makes the commit self-documenting in terms of traceability: an auditor can see this and know which **requirement** it ties to and that there is a corresponding test case. It also helps you maintain a trace matrix automatically, since the links (REQ-9, R-15, TC-22) can be cross-referenced with your requirements and risk management records. Indeed, FDA expects traceability across these items fda.gov, and a disciplined commit messaging practice is a lightweight way to achieve it. Additionally, using **imperative mood and present tense** in commit messages (as per many Git guidelines) is recommended ("Fix bug…" "Add check…"), making them consistent and easy to compile into release notes.

- **Changelog and Version History Documentation:** Maintain a **human-readable changelog** or "release notes" document that is updated for each release. This can often be generated or at least started by tooling that reads commit messages (especially if you follow a structured format like *Conventional Commits*). The changelog should summarize new features, fixes, and changes in each version – effectively an expansion of the "brief description of changes" that FDA wants in the submission's software version history fda.gov. By comparing successive entries, one can see what changed between versions. You might keep this changelog in the repository (e.g., a `CHANGELOG.md` file) and tag each section with the version. During submission, the contents of this can populate the Software Version History table. Because it's under Git, every update to the changelog is itself tracked, preserving the documentation evolution. Ensure the final version of the changelog for a release is reviewed for accuracy and approved (this could be part of the release PR). This practice not only helps regulators; it also helps your internal teams and customers understand changes.

- **Linking Git to Issue Trackers:** Integrate your Git workflow with an **issue tracking system** (such as Jira, GitLab issues, GitHub issues, Azure Boards). Each change in a regulated project typically originates from a formal change request, requirement, or defect report. By tracking those in an issue system and then linking commits or merge requests to the issue (through mentions or IDs), you maintain a rich web of traceability. For example, your procedure might require that "each commit must mention an approved change request ID," and your CI pipeline could even enforce this by rejecting commits that don't follow the pattern. Many modern tools will automatically close or comment on an issue when a linked commit is merged – providing an **audit trail that the code change corresponding to a requirement has been implemented and merged**. The issue can then be updated to reflect verification status (e.g., testers attach evidence). This way, when preparing submission documentation or an audit, you can easily pull reports: show me all commits linked to requirement X and all test results linked to requirement X. Some compliance software platforms (like Ketryx, Jama, etc.) offer out-of-the-box **traceability reports that aggregate data from Git and issue trackers** ketryx.com. The key is consistency in linking them. The benefit is twofold: developers work naturally (fix issue -> commit code -> close issue), and traceability is generated in the background. It reduces duplicate effort compared to manually maintaining a separate trace matrix in Excel.

- **Continuous Integration / Continuous Delivery (CI/CD):** A common misconception is that CI/CD can't be used in regulated environments – it absolutely can, but you must manage it under configuration control. A CI system (Jenkins, GitLab CI, GitHub Actions, etc.) should be treated as a *production tool* and thus **validated for its intended use** (ensuring the pipeline builds and tests correctly, and that any risk of it failing is mitigated) greenlight.guru. Once validated, CI can dramatically enhance compliance by automatically running tests and analyses on each commit. This provides evidence of software verification: for example, each commit or PR triggers unit tests, integration tests, static code analysis, maybe even generation of documents like requirements traceability matrices or test result summaries. The results can be archived (for instance, store the test reports or a link to the CI job output for each build). Regulators appreciate seeing objective evidence of testing. CI pipelines can also automate **compliance checks** – e.g., verifying that commit messages follow the convention, that every requirement has at least one test, that release builds include proper version metadata, etc. When it comes time to do a formal release, the pipeline can produce an **artifact package** containing the compiled software, a PDF of release notes, an SBOM (Software Bill of Materials), and even the **DHF documents** updated to that version. Such artifacts can be attached to the Git release tag or stored in a controlled repository. Automating these tasks reduces human error and ensures repeatability (a principle FDA likes in processes). If using CI for deployment (DevOps), ensure that deployment scripts are also under version control and validated. The pipeline definitions themselves (like a `.gitlab-ci.yml` or Jenkinsfile) should be under Git control, so changes to the build process are tracked and approved.

- **Documentation Repositories and MDVC (Modern Document Version Control):** Consider using Git not just for code but for maintaining **living documentation**. For instance, **SRS (Software Requirements Specifications)**, design descriptions, and test protocols can be written in a lightweight markup (Markdown, AsciiDoc) and versioned in the repo. This way, any change to requirements or design is tracked by Git with a commit history. Collaboration on docs can use the same pull request/review workflow as code. If preferred to use traditional tools (like Word or Excel), you can still version those binary files in Git – though diffing them is not easy, the version history is maintained. Alternatively, store those documents in an **electronic document management system (eQMS)** that can integrate with your development tools. Some organizations sync document statuses with Git (for example, using an ID in the document to link to a commit). The emerging practice is to treat "documentation as code," which means applying the same rigor of peer review, versioning, and automation to documents as to source code. The OpenRegulatory initiative, for example, demonstrates how a QMS can be implemented in GitHub/GitLab using Markdown and pull request approvals in lieu of wet-ink signatures openregulatory.com openregulatory.com. They caution, however, that **electronic signatures** in such a system need careful consideration to truly meet FDA requirements – simply typing a name in a Markdown file is not a secure, unique signature openregulatory.com. A mitigation is to use the Git platform's **merge approval as a signature**, combined with audit logs and perhaps GPG signing of the merge commit by the approver. If that approach is taken, document it in your SOP: e.g., "Approval of a pull request by the Quality Manager constitutes an electronic signature for document approval in compliance with 21 CFR Part 11." Also ensure that your Git system requires unique logins and password policies as per Part 11 (many Part 11 requirements for closed systems, like unique user ID and password controls, are handled by the IT infrastructure of your source control system) dev.to nsflow.com.

- **Generating Traceability and Design History File (DHF) Outputs:** Ultimately, during an FDA submission or inspection, you will need to present your design control evidence in a readable format. Git can be leveraged to generate some of this content automatically. For instance, you can create scripts or use tools that **extract commit logs between two tagged versions** to produce a **version history table** for that interval. If commit messages are well-structured, you can populate columns like "Change Description" and "Rationale" directly from them. Some teams maintain a "DHF index" document in the repo that lists all required items (SRS, risk analysis, test plan, etc.) and the current version or commit for each. With integrations (like those provided by compliance platforms such as Ketryx), the system can pull data from **Jira (requirements, test results) and GitHub (commits, code reviews)** to automatically build a DHF or trace matrix ketryx.com. Even without specialized tools, a combination of Git log, issue tracker queries, and perhaps a little programming can produce these documents, which you then review and include in submissions. The advantage is consistency – the documents come straight from the source of truth (code and requirements), reducing manual transcription errors. Regulators are increasingly receptive to digital records; the FDA's own guidance acknowledges modern practices and encourages sponsors to use current **standards and best practices** fda.gov. Showing that your submission documentation is directly tied to your version control and requirements systems demonstrates a high level of control.

- **Linking Test Results and Validation Evidence:** Store or link **test results** to the relevant commit or build. For example, after a successful CI test run on a release candidate, you might commit a test summary or have the CI system post the results to an issue. Ensuring that for every release tag, you have captured the verification status (pass/fail of all tests, code coverage, etc.) is essential for the submission's **Verification and Validation** section. Also, if you perform formal **tool validation** of Git or other tools, keep those records accessible (perhaps in a separate quality repository or eQMS, but refer to them in your plan). FDA may not ask to see tool validation for something like Git, but if it's part of your procedure, it should be available. Using Git to manage the **software lifecycle data** end-to-end – from requirements to code to test to release – enables a high degree of **automation and data integrity**, but it must be backed by clear documentation of the process (in SOPs) and evidence that the process is followed (in records and logs).

In implementing these practices, organizations often find that **regulatory compliance and developer productivity are not at odds** – a well-integrated toolchain can achieve both. By automatically generating documentation and linking data, developers spend less time on bureaucratic tasks while auditors get a richer view into the project. The next section will discuss how to preserve these records immutably and address the nuances of electronic signatures in such a Git-centric workflow.

Preserving Immutability, Audit Trails, and Electronic Signatures

One of the core concerns of regulators is ensuring that records – whether design documents or source code – are **unalterable or at least change-tracked** once finalized. Git provides a strong foundation for immutability through its hashed history, but it is not foolproof out of the box. Here

we cover techniques to **harden Git's immutability and audit features**, and how to handle electronic signatures for compliance:

- **Immutable Repositories and Append-Only Access:** As mentioned, configure your central Git repository to be effectively **append-only** for critical branches. This means disabling non-fast-forward updates (no force pushes, no deleting tags or branches without admin approval). Some teams even adopt a workflow where developers don't push directly to the main repository at all – instead, all changes come via pull requests and are merged by a continuous integration bot or release manager. This creates a checkpoint where integrity can be verified before merging. Additionally, consider using a **code hosting solution that supports append-only mirroring** or immutable logs. For instance, some regulated firms periodically export the Git repository (or specific important branches) to a secure, timestamped archive or an external write-once medium. This might be done at release milestones – effectively taking a "snapshot" of the repository state and storing it in a **design control vault**. In any dispute or audit, you can show that snapshot and the live repo and demonstrate they match (or identify discrepancies, which in itself is evidence of tampering if it ever occurred). The Inductive Automation Part 11 guidance specifically highlights using append-only mechanisms for record generation and audit trails to meet regulatory expectations s3.amazonaws.com, which aligns with treating the version control history as append-only.

- **Audit Trail Completeness:** Git's commit history is a form of audit trail, but there are additional audit considerations. For example, if someone attempts an unauthorized action (like trying to force push or access the repo), your system should log that too (this would be at the server/IT level, not in Git itself). Ensure that server logs or Git management logs are kept as part of your records retention. Also, note that Git by default tracks changes to file content but not *who viewed or who approved* something. That's where the integration with code reviews and issue trackers is important: those systems should maintain **records of reviews, approvals, and sign-offs**. Export or archive those records (for instance, store the merged pull request with its discussion and approval list, perhaps as a PDF or in an audit system). Some tools have an API to pull this info. The goal is that if FDA asks, "Show us the approval of changes for version 2.0," you can produce a report of all PRs that went into v2.0, each showing who reviewed/approved and when – essentially an **audit trail of the design change control process**. In regulated industries, this is often referred to as an **electronic Design History File log**.

- **Electronic Signatures on Commits or Merges:** Part 11 Subpart C deals with electronic signatures, requiring uniqueness and verification of identity, as well as linking signatures to their records. In a Git workflow, there are a few ways to implement what could be considered an e-signature for a software change or document:

- The **commit author/committer identity** (name, email) plus the commit hash itself can serve as a sort of signature for code changes. It's not a signature in the legal sense, but it links the change to a person. If commit signing (GPG) is used, it strengthens this link by cryptographically certifying it.

- A **pull request approval** can be seen as an electronic signature on the set of changes in that pull request. For this to be Part 11 compliant, you need to ensure that the act of approval is firmly attributed to a specific individual (unique login) and includes a meaning of signature (e.g., the UI might show "Approved by Jane Doe on 2025-08-08" – this is a signature manifestation). It should also be irrevocably tied to the record (once merged, the approver and timestamp are logged and can't be easily altered without trace). GitLab, for example, has been exploring features for Part 11 compliant approvals with additional confirmation steps (entering a password at approval, etc., to mirror a 21 CFR 11 signature) gitlab.com. If your platform doesn't have that, your SOP can require that approvers re-authenticate when logging in and note that as satisfying the intent.

- **Tag signing by responsible parties:** Another approach is to have responsible individuals sign the Git tag of a release. For instance, after all approvals, the QA manager uses `git tag -s` to create a signed tag `v1.0` on the release commit, including a message like "Approved for release by QA Manager on date". The signed tag is then pushed. This provides a clear, unforgeable marker in the repo that this version was approved. The signature can be verified independently. To an FDA inspector, this could be presented as the electronic signature of the release. However, be prepared to explain your process (how you ensure only authorized folks have the signing keys, how you archive keys, etc.).

- **Records Retention and Retrieval:** FDA requires that electronic records and their audit trails be retained as long as the record itself (often for years, e.g., the life of the device plus some time). A Git repository must therefore be **preserved and accessible** for potentially a long period. This means robust backup strategies – regular backups of the repository to a secure, off-site location – and migration plans if technologies change. Using open formats (Git's data can always be retrieved even if you change hosting providers) is helpful. If using a cloud service, ensure you can export the full repo easily. Also, consider how you will provide records to an inspector if asked. You likely wouldn't give them raw Git repo access, but you might generate specific reports (commit histories, diff listings, etc.). Having **documented procedures on how to retrieve historical versions** is part of being compliant. For example, "Procedure XYZ: Retrieving Historical Software Versions" could outline how to check out an old tag and rebuild the software (with archived build environment if needed). Showing this capability proves the integrity and maintainability of records.

- **Traceability of Changes to Requirements and Risks:** Immutability isn't just for code; you should also maintain an audit trail of changes to specifications and requirements. If those are stored in Git, the same rules apply (no deleting history, etc.). If they are in an external system, ensure that system has versioning or you export version snapshots to Git. For example, some teams export a requirements document to PDF and commit it to Git at each baseline release, so that the exact version of requirements corresponding to a software version is saved. This can go in an **Appendix of the DHF**. Part 11 doesn't explicitly mention requirements docs, but the principle of **keeping an audit trail of all design decisions and changes** certainly applies under design controls.

- **Handling of Deviations and Anomalies:** FDA submissions also require listing **unresolved anomalies (bugs)** in the released software. If you track bugs in an issue tracker linked with Git, you can generate this list from open issues at release time. The tie-in with version control is that each bug fix is a commit, and unresolved ones presumably have no commit yet. Maintaining the state of these in a controlled way (don't quietly close issues without rationale, etc.) is part of the record integrity. It's worth mentioning in your configuration management plan how you use the issue tracker and Git to handle bug tracking and ensure none are lost.

- **Training and User Policies:** A subtle but important aspect of Part 11 compliance is user training and policies. All users of the Git system should be trained in the SOP (so they know not to rewrite history, to write proper commit messages, to secure their credentials, etc.). Also, policies like **no shared accounts** (each person has their own login) are critical for accountability dev.to. Implement technical controls to enforce password complexity and inactivity timeouts on your Git platforms dev.to – these are standard IT security but map to Part 11 rules for ensuring only authorized use. In case of personnel leaving, promptly remove their access (and maybe revoke their signing key if used). These measures, while not unique to Git, ensure that the chain of trust in your version control records is maintained.

In essence, preserving immutability and audit trails in Git requires a mix of **technical settings, procedural controls, and possibly cryptographic signing**. When done correctly, you can confidently show that your software repository is an **unalterable ledger of the development process**, and that every change was authorized and signed off. This satisfies regulatory expectations for electronic record integrity and signature accountability. As a result, the Git repository (and its associated tool logs) becomes a central piece of your **compliance evidence**.

Case Studies: Git Workflows in FDA-Regulated Projects

To illustrate these concepts, we consider a couple of **example scenarios** (based on real-world practices) where Git-based workflows were successfully used to manage software versioning and documentation in FDA submissions:

Case Study 1: Medical Device Software 510(k) Submission
Acme HealthTech is a startup developing a Class II medical device – a wearable cardiac monitor with accompanying analysis software (a Software as Medical Device component). In preparation for a **510(k) submission**, Acme needed to document their software development lifecycle (per FDA's guidance) and ensure traceability of requirements through design, implementation, and test. They adopted **GitLab** as their central code repository and configured a workflow aligned with GitFlow. Key elements included:

- A `main` branch representing the latest released software and a `develop` branch for ongoing development. For version 1.0 (the initial release to be submitted), they created a `release-1.0` branch from `develop` once feature-complete.

- All developers worked on feature branches named after the requirement or issue (e.g., `feat/REQ-5-user-authentication`). Each branch was linked to a requirement in their Jira project. Jira and GitLab were integrated such that mentioning "REQ-5" in a commit message automatically updated the Jira ticket with a reference to that commit. This provided instantaneous **traceability** from requirement to code. Additionally, a **traceability matrix** could be exported from Jira showing for each requirement: the implementing commit(s) and the test case results (because they also linked automated test results to Jira).

- When developers considered a feature done, they opened a **Merge Request (MR)** on GitLab to merge their feature branch into `develop`. The MR required at least one **peer review approval** (for code quality) and one **QA representative approval** (for regulatory impact assessment). GitLab's approval rules ensured no MR could be merged without these signatures. Each approval recorded the user's name and timestamp – effectively an **electronic signature of review and assent** on that change s3.amazonaws.com.

- Once all MRs intended for version 1.0 were merged into `develop` and tests passed, the team performed a **release candidate build** from the `release-1.0` branch. They ran full verification testing (including integration tests on actual devices). The results were archived. They then tagged the commit as `v1.0.0-rc1`. Some bugs were found during validation, which led to new commits on the release branch to fix them (each fix was also done via MR with approval). This incremental approach is compliant with FDA's expectation that you assess and test all changes even late in the cycle fda.gov.

- After resolving test issues, the final commit on `release-1.0` was tagged `v1.0.0`. The **QA Manager digitally signed the tag** with her GPG key and pushed it. This tag sign-off indicated that this exact code is approved for release.

- For the submission, Acme compiled the **Software Development and Configuration Management documentation** largely from their GitLab records. They provided a summary of their processes (development model, branching strategy, tools) referencing conformity to **IEC 62304 clauses 5.1 and 8** fda.gov. They included the **Software Version History table**: it listed version 0.5 (the version when formal design control started) through 1.0.0, with dates and brief descriptions. These descriptions were pulled from the Git commit messages of the release merges, giving a high-level overview of what changed in each version fda.gov. For instance, "v0.7.0 – Added user authentication module (Req-5), improved ECG filtering (Req-8); internal testing only" and so on, up to "v1.0.0 – Final release version; includes all features, performance verified; no outstanding anomalies." Because the team tracked issues, they could confidently state "no outstanding anomalies" or list the few minor ones known, directly referencing their issue tracker for details.

- Acme also submitted an **extract of their Git commit log** (filtered for significant changes) as an appendix, and a mapping of commit IDs to requirement IDs to test cases (as their trace matrix). During the FDA review, there were questions about a particular algorithm change. Acme was able to, on the spot, retrieve the specific commit in question (using the hash from the trace matrix) and show the code diff and commit message explaining the change and its rationale. This level of transparency impressed the reviewers and satisfied their questions quickly, reinforcing the credibility of Acme's submission.

- Importantly, Acme's quality team had validated the GitLab system and its use. They had a **procedure for configuration management** which FDA inspected. It described how branches are managed, how approvals happen, how records (like MRs and issues) are retained. The fact that GitLab kept an **audit trail of all merges and user actions** was noted. Acme's use of **role-based access control** (only senior engineers could merge to `develop` or `main`) and unique user logins satisfied Part 11 requirements about system security and signature uniqueness.

Result: Acme HealthTech's submission sailed through the software review with minimal queries. The FDA reviewer even commented that the software documentation was well organized and

traceable, likely thanks to the rigorous use of version control and linked tools. Acme continues to use the same workflow for post-market changes, maintaining the audit trail for any future audits or submissions (like an eventual PMA).

Case Study 2: Pharmaceutical Data Pipeline – IND Application
PharmaCo is a pharmaceutical company that, as part of an **IND submission for a new drug**, needed to include information about a custom software tool used to analyze clinical trial data (a complex statistical script and application). Although the software itself was not patient-facing, it fell under FDA scrutiny because its outputs supported safety and efficacy claims. PharmaCo used **GitHub Enterprise** to manage the code and collaborated with an external biostatistics firm. Key compliance steps included:

- All code (written in R and Python) was stored in a GitHub repo. A **branching strategy** wasn't complex (they mainly worked on a main branch with feature branches for big changes), but they tagged a version for each clinical study dataset analysis (e.g., `analysis-v1`, `analysis-v2` corresponding to interim and final analyses).

- They established a **commit signing policy**: each statistician and programmer had to GPG-sign their commits. The repo was configured to reject unsigned commits. This ensured that every change to the analysis script could be traced to a specific, authenticated individual – addressing any concerns of data manipulation. The commit messages also had to reference a **validation ticket ID** which corresponded to a verification task (they treated the analysis scripts like software requiring verification).

- Each version tag of the analysis tool was linked to validation documents. For example, when tagging `analysis-v2` for the final analysis, they attached an export (as a PDF) of the automated test results and code review checklist to the GitHub Release associated with that tag. Thus, if someone browsed that release in GitHub, they would find the exact code and the evidence of its validation.

- Part 11 compliance was addressed by controlling access (only a small team had access to the repo, all via individual corporate accounts) and by using GitHub's **protected branches** and **required reviews**. For instance, even a one-line change in the analysis script required a pull request reviewed by a second statistician and a QA person who ensured the change was documented in the validation plan. The approval was recorded in GitHub's pull request record. This two-person rule is analogous to what paper-based systems would do with two signatures on a document change request.

- When the FDA asked in a meeting how PharmaCo ensured the integrity of their analysis, PharmaCo demonstrated their use of Git. They explained that **every modification to the analysis code was logged and reviewable**, and that they could reproduce any prior analysis by checking out the tagged version. To prove it, they actually regenerated a key analysis figure live with a past version of the code from Git, showing the result matched the one in the interim report. This level of reproducibility is a direct benefit of strict version control: it convinced the FDA that the data analysis was reliable and that no "data dredging" or untracked changes had occurred.

- In the IND submission documents, PharmaCo included a section on "Software Controls for Clinical Data Analysis" where they cited **21 CFR Part 11** and explained how their use of GitHub Enterprise provided audit trails and access control. They referenced FDA's guidance on electronic records nsflow.com to assert that their approach met the requirements (time-stamped audit trail, etc.). They also cited internal SOPs aligning with **GAMP5 guidelines** for computerized systems in GxP use. Essentially, they treated the analysis software and its repository as a validated system.

Result: The IND was accepted, and FDA did not request additional audits of PharmaCo's software procedures. The confidence was partly due to the proactive transparency PharmaCo showed. Later, when PharmaCo moved to an NDA (New Drug Application), they continued this approach and even expanded automation (using CI to automatically run all statistical tests when code changed, ensuring no errors were introduced unknowingly). Git versioning became a cornerstone of their data integrity assurance.

Case Study 3: Digital Health Software under IEC 62304 (simulated example combining multiple real scenarios)
MediSoft Inc. develops a mobile app that qualifies as a **Software as a Medical Device (SaMD)**. They must comply with IEC 62304 and FDA's digital health documentation guidelines. They integrate **Git**, **Jira**, and an **eQMS** system:

- Requirements and risk analysis are maintained in Jira. Each requirement is tagged with a risk level and linked to risk mitigations.

- The software code is in Git (Bitbucket server) with a branching model: `main` for releases, feature branches for development. For each **software item** identified in their configuration management plan (e.g., UI module, Algorithm module), they maintain subfolders in the repo and sometimes separate component branches to isolate changes.

- They generate a **Configuration Item Index** (part of config identification in IEC 62304) which is basically a list of all software items and their current version (Git commit hash or tag). A simple script pulls the latest commit ID for each relevant subdirectory and outputs a table. This index is included in the DHF to show traceability of **software items to version** greenlight.guru.

- When preparing for an FDA submission (they used the De Novo pathway for a novel device), they leveraged Git's history to fill in the **"Software Version History"** table. The first version under design control was 0.1 (when they completed requirements). Through Git log queries, they listed each major milestone version, date, and a summary of changes fda.gov. They also indicated which versions were used in verification testing (some versions were internal only, some went to a clinical study). Because Git tags were used for each test release, they had exact references.

- MediSoft's **change control procedure** required that any change to a requirement or risk assessment must go through a change request in their eQMS, but thanks to integration, the developers could trigger this via Git/Jira: if code was changed without an update to a linked requirement when needed, the CI pipeline would flag it. This prevented gaps in documentation. All changes ultimately got reflected in the **DHF documents**, which were versioned and signed in the eQMS, but the eQMS records would include the Git commit IDs for cross-reference.

- During audit, MediSoft provided the auditor with a read-only login to their Bitbucket server to browse the repo history, showing confidence in their process. The auditor examined the commit log around a particular bug fix and saw the peer review comments and approvals attached to that commit. The auditor noted this as an example of a good **audit trail** for software changes, fulfilling regulatory expectations that changes are documented and reviewed prior to implementation ecfr.gov.

These case studies demonstrate that **Git is flexible enough to handle various regulated scenarios** – from device firmware to clinical analysis code to mobile health apps. The common thread is a disciplined workflow, integration with complementary tools, and a mindset that treats the repository as a **regulated record store**. By aligning Git usage with compliance requirements (traceability, change control, audit trails, signatures), companies have successfully navigated FDA submissions while maintaining modern DevOps practices.

Conclusion

Git has emerged not only as a dominant tool for software development, but also as an enabler of regulatory compliance when used thoughtfully. In FDA-regulated environments – spanning medical devices, pharma, biotech, and digital health – the demands for **complete, traceable, and tamper-evident documentation** of the software lifecycle are non-negotiable. As we have detailed, Git's capabilities map closely to these needs: its distributed yet secure change tracking provides the **traceability and audit trail** envisioned by standards like FDA 21 CFR Part 11 and IEC 62304. By implementing robust Git workflows (branch protections, code reviews, release tagging) and integrating with issue tracking and CI/CD, organizations can ensure that **every change is captured, justified, reviewed, tested, and approved** – exactly the assurance regulators seek.

We explored how specific practices – from commit message conventions referencing requirements, to GPG-signing commits and tags, to generating version history tables from Git logs – can turn a Git repository into a living Design History File. The **best practices** outlined, such as using pull requests to enforce independent review and maintaining an immutable history, directly support compliance with FDA's design control (820.30) requirements for design change control and verification ecfr.gov. Moreover, treating Git as part of the quality system (with appropriate validation and SOPs) addresses the expectations of Part 11 for trustworthy electronic records nsflow.com and signatures. The case studies provided give a glimpse into how real teams have **successfully harnessed Git for FDA submissions**, not only satisfying regulatory requirements but often exceeding them, thereby reducing the friction of audits and reviews.

In conclusion, a well-managed Git workflow can be **"your way to compliance"** by embedding compliance activities into the everyday development process. Rather than maintain separate, duplicative documentation to satisfy regulators, teams can lean on their version control system as a single source of truth. This approach improves developer efficiency (less manual paperwork) and yields more reliable, consistent documentation for regulators. However, success

requires careful planning: writing clear procedures, training staff on compliance-conscious use of Git, and selecting tool configurations that maximize integrity (e.g., access control, audit logging, backup). When these elements are in place, Git not only streamlines development but also enhances regulatory **peace of mind**, ensuring that when it's time to submit that 510(k) or NDA, the **traceability matrix, audit trail, and version history** are just a few clicks away in the repository.

By versioning your way to compliance, you build quality and accountability into the product from day one – fulfilling the letter and spirit of FDA regulations while leveraging the best of modern software engineering practices.

References (Selected)

1. FDA, *Content of Premarket Submissions for Device Software Functions – Guidance for Industry and FDA Staff*, June 2023. (Recommends documentation of software development, configuration management, and provides for IEC 62304 conformity) fda.gov fda.gov.
2. FDA, *21 CFR 820.30 Design Controls*. (Regulation requiring design change control and Design History File) ecfr.gov ketryx.com.
3. FDA, *21 CFR Part 11, Electronic Records; Electronic Signatures*. (Criteria for trustworthy electronic records and signatures, including audit trail requirements) nsflow.com dev.to.
4. Johner Institute, "21 CFR Part 11: An over-interpreted law?" August 2024. (Discussion of Part 11 requirements like audit trails) blog.johner-institute.com.
5. IEC 62304:2006+A1:2015, *Medical Device Software – Software Life Cycle Processes*. (International standard mandating configuration management, change control, and traceability in software development) greenlight.guru.
6. Inductive Automation, *21 CFR Part 11 and Pharmaceutical Best Practices with Ignition*, 2022. (White paper recommending use of Git for change control and two-person code reviews in compliance workflows) s3.amazonaws.com s3.amazonaws.com.
7. Greenlight Guru, "SaMD Software as a Medical Device – The Ultimate Guide." (Explains IEC 62304 practices like configuration management and tool validation in SaMD development) greenlight.guru greenlight.guru.
8. Ketryx Blog, "How to Create a Design History File (DHF) for Medical Devices," Jan 2025. (Discusses traceability and integration of tools like Jira and GitHub to auto-generate DHF contents) ketryx.com.
9. OpenRegulatory, "Setting Up a QMS in GitHub/GitLab." (Article illustrating use of Git for document control and discussing electronic signature limitations in such setups) openregulatory.com openregulatory.com.

(Additional citations are embedded in text as † for specific assertions.)

IntuitionLabs - Industry Leadership & Services

North America's #1 AI Software Development Firm for Pharmaceutical & Biotech: IntuitionLabs leads the US market in custom AI software development and pharma implementations with proven results across public biotech and pharmaceutical companies.

Elite Client Portfolio: Trusted by NASDAQ-listed pharmaceutical companies including Scilex Holding Company (SCLX) and leading CROs across North America.

Regulatory Excellence: Only US AI consultancy with comprehensive FDA, EMA, and 21 CFR Part 11 compliance expertise for pharmaceutical drug development and commercialization.

Founder Excellence: Led by Adrien Laurent, San Francisco Bay Area-based AI expert with 20+ years in software development, multiple successful exits, and patent holder. Recognized as one of the top AI experts in the USA.

Custom AI Software Development: Build tailored pharmaceutical AI applications, custom CRMs, chatbots, and ERP systems with advanced analytics and regulatory compliance capabilities.

Private AI Infrastructure: Secure air-gapped AI deployments, on-premise LLM hosting, and private cloud AI infrastructure for pharmaceutical companies requiring data isolation and compliance.

Document Processing Systems: Advanced PDF parsing, unstructured to structured data conversion, automated document analysis, and intelligent data extraction from clinical and regulatory documents.

Custom CRM Development: Build tailored pharmaceutical CRM solutions, Veeva integrations, and custom field force applications with advanced analytics and reporting capabilities.

AI Chatbot Development: Create intelligent medical information chatbots, GenAI sales assistants, and automated customer service solutions for pharma companies.

Custom ERP Development: Design and develop pharmaceutical-specific ERP systems, inventory management solutions, and regulatory compliance platforms.

Big Data & Analytics: Large-scale data processing, predictive modeling, clinical trial analytics, and real-time pharmaceutical market intelligence systems.

Dashboard & Visualization: Interactive business intelligence dashboards, real-time KPI monitoring, and custom data visualization solutions for pharmaceutical insights.

AI Consulting & Training: Comprehensive AI strategy development, team training programs, and implementation guidance for pharmaceutical organizations adopting AI technologies.

Contact founder Adrien Laurent and team at https://intuitionlabs.ai/contact for a consultation.

DISCLAIMER

The information contained in this document is provided for educational and informational purposes only. We make no representations or warranties of any kind, express or implied, about the completeness, accuracy, reliability, suitability, or availability of the information contained herein.

Any reliance you place on such information is strictly at your own risk. In no event will IntuitionLabs.ai or its representatives be liable for any loss or damage including without limitation, indirect or consequential loss or damage, or any loss or damage whatsoever arising from the use of information presented in this document.

This document may contain content generated with the assistance of artificial intelligence technologies. AI-generated content may contain errors, omissions, or inaccuracies. Readers are advised to independently verify any critical information before acting upon it.

All product names, logos, brands, trademarks, and registered trademarks mentioned in this document are the property of their respective owners. All company, product, and service names used in this document are for identification purposes only. Use of these names, logos, trademarks, and brands does not imply endorsement by the respective trademark holders.

IntuitionLabs.ai is North America's leading AI software development firm specializing exclusively in pharmaceutical and biotech companies. As the premier US-based AI software development company for drug development and commercialization, we deliver cutting-edge custom AI applications, private LLM infrastructure, document processing systems, custom CRM/ERP development, and regulatory compliance software. Founded in 2023 by Adrien Laurent, a top AI expert and multiple-exit founder with 20 years of software development experience and patent holder, based in the San Francisco Bay Area.

This document does not constitute professional or legal advice. For specific guidance related to your business needs, please consult with appropriate qualified professionals.