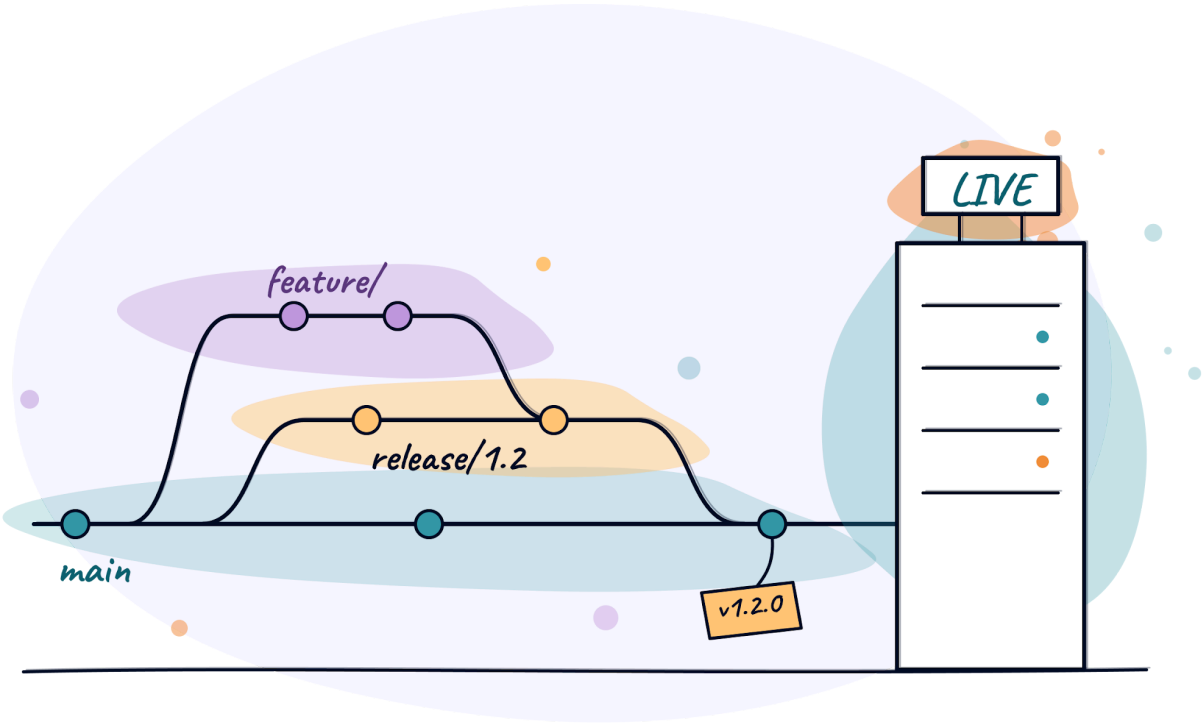
A SiteRocket Labs ebook

Branching Out

8 Git Branching Rules for Boring Deployments



By SiteRocket Labs
www.siterocket.com

SiteRocket® Labs

Introduction: Why Branching Matters

Imagine a shared kitchen where every cook can add ingredients to the same pot, at any time, with no recipe. Maybe dinner turns out fine. More likely, someone adds salt, someone else adds salt, and nobody can work out why the end result tastes terrible.

A codebase without a branching policy is that pot of soup. Here is what a good policy gives you:

- **A clean, readable history.** When something breaks, you can look back and see exactly what changed, when, and why.
- **Safer releases.** You always know what is in production and what isn't.
- **Fewer surprises.** Work in progress doesn't leak into live code by accident.
- **Easier teamwork.** Developers aren't stepping on each other's toes, and nobody has to ask "wait, which branch should I use?"
- **Less stress.** When the process is predictable, releases become boring. Boring is exactly what you want from a release.

The rules in this ebook might feel strict at first. But every one of them exists because somebody, somewhere, learned the hard way what happens without it. (We'll tell you a few of those stories along the way. Names have been changed to protect the embarrassed.)

For teams building healthcare software, this matters even more. Standards like ISO 13485 and IEC 62304 expect you to show exactly what changed in each release, why, and who approved it. A clear branching policy makes that traceability part of everyday work instead of a scramble before an audit.

> ### Before We Begin: This Is Our Way, Not *The* Way
>
> This ebook walks through the policy that we use internally at SiteRocket Labs on client projects, and it works for us. That said, **it describes one way of doing things, not the only way.** So read this as an example of what can work and adapt it to your needs. The most important thing is that you *have* an official code branching policy and that everybody follows it.

Chapter 1: A Quick Git Refresher

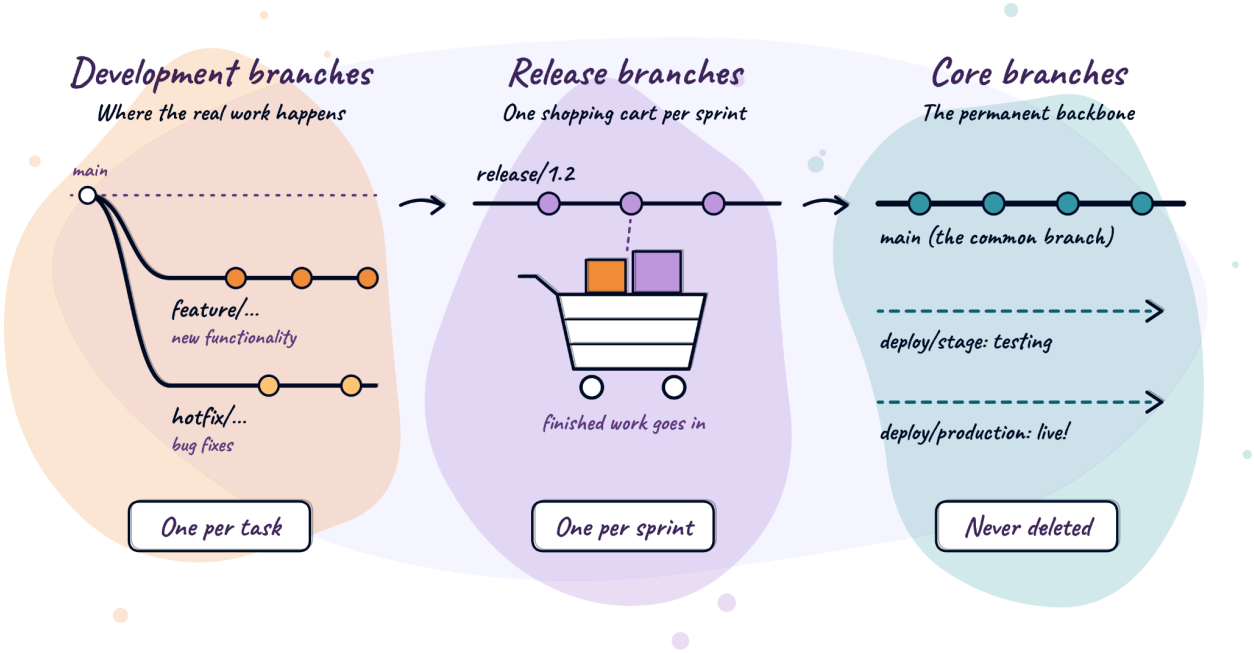
You don't need to be a Git expert to follow along, but a few terms come up a lot.

- **Branch:** A separate version of the codebase. Think of it as a parallel copy of the code where you can make changes without affecting anyone else.
- **Commit:** A saved snapshot of your changes, with a message describing them.
- **Merge:** Combining the changes from one branch into another.
- **Merge commit:** A special commit that records "these two branches were joined here." You create one on purpose by merging with the `--no-ff` flag (short for "no fast-forward"). It leaves a visible marker in the history showing exactly where a branch came in.
- **Fast-forward:** When the branch you're merging into has nothing new, Git simply slides it forward to match. No merge commit is created. You can insist on this with `--ff-only`, which will refuse to merge if a fast-forward isn't possible.
- **Rebase:** Taking the commits from your branch and replaying them on top of a newer starting point, as if you had begun your work from there. It's how you bring in the latest changes without creating a messy merge.
- **Tag:** A permanent, named label stuck onto a specific commit, usually used for version numbers like `v1.2.0`.
- **Push:** Sending your local commits up to the shared repository.

Keep these in your back pocket. We'll use them constantly.

Chapter 2: Meet the Branches

Our policy uses three families of branches: **core**, **development**, and **release**.



Core Branches

Core branches are the permanent backbone of the project and consist of the common branch (main) and the deploy branches. They are never deleted, and only the lead developers on a project are allowed to create them.

The common branch

This is your main line of history, and we always call it `main`, though some companies may name this branch `master`. Regardless of its name, we refer to this branch as “the common branch” because every other branch is created from it and is merged back into it. It holds finished, tested work and should be considered the “single source of truth”.

Deploy branches

A deploy branch is wired up to an environment (a place where your app runs). Deploy branch names always start with `deploy/`, followed by the name of the environment. So `deploy/stage` matches the `stage` environment.

Here's the key idea: **pushing to a deploy branch automatically deploys your code to that environment.** The branch isn't just a branch; it's a trigger.

Because of that, the commits on a deploy branch should be thought of as *ephemeral*. They only represent what is deployed *right now* to that environment, and nothing more. They are not a place to do work, and not a source of truth for history.

There are two kinds of deploy branches:

1. `deploy/production` matches your live environment, the one real users see. It may only contain commits that are already on the common branch. At times it may be several commits *behind* the common branch. This can happen if some finished work is scheduled to go out at a later time, and that's fine. What matters is that nothing ever lands on production that isn't already on the common branch.
2. **Non-production deploy branches** are everything else: `deploy/stage`, `deploy/uat`, and so on. These are testing environments. To make sure testing is realistic, they're periodically aligned with the common branch using a "hard reset," which wipes the branch back to match the common branch exactly. That keeps your test environments looking as much like production as possible, rather than slowly drifting into a strange hybrid of old experiments.

Development Branches

This is where the real work happens. Every change, however small, starts on a development branch. There are two types:

- `hotfix/` for branches that mainly contain bug fixes.
- `feature/` for everything else (new functionality, improvements, and so on). If a branch isn't mainly a bug fix, it's a feature.

Release Branches

A release branch collects all the changes that belong to a particular sprint or batch of work. Each one carries a major and minor version number and is named like this: `release/1.2`.

Think of a release branch as a shopping cart. Over the course of a sprint, development branches get dropped into the cart as they are finished. When the cart is ready, you check out (test it, and ship it).

What if you need to ship an urgent bug fix between sprints? Say version `1.2` is live and `1.3` isn't ready yet. The fix goes into the branch for the most recently released version, `release/1.2`, and gets tagged as a patch version (for example `v1.2.1`). Patch numbers never appear in a release branch's name, so the branch should still be named `release/1.2`.

Chapter 3: Naming Things

Good names save time. Every development branch is named using three parts:

1. A **prefix** for the type (`feature/` or `hotfix/`)
2. A short, **descriptive slug** in lowercase with dashes
3. The **task ID** from your project management system, if there is one

For example, let's say that you're developing a new registration workflow for your app that allows users to configure multi-factor authentication, and the task ID is XY889 in your project management system. In this case, you'd name your development branch something like this:

`feature/register-with-mfa-XY889`.

The **task ID** is especially important if you need to adhere to a quality standard like ISO 13485 or IEC 62304, which mandates traceability.

If one pull request covers more than one task, include all the IDs:

`feature/user-onboarding-workflows-XY889-XY953`

Anyone glancing at a list of branches can tell what each one is for, what kind of change it is, and exactly which task to look at for the full story.

Chapter 4: The Life of a Change

Let's follow a feature from start to finish. This is the whole policy in one story. (Don't worry about memorizing it; the rules in the next chapter explain each step.)

Step 1: Start fresh. Pull down the latest commits, then create your development branch from the newest commit on the common branch. Let's say you're building a new settings page: `feature/add-settings-page-AB210`.

Step 2: Do your work. Commit on your own branch. Never commit directly to the common branch or any deploy branches.

Step 3: Open a pull request. A teammate reviews your work. (More on this in Chapter 6.)

Step 4: Drop it in the cart. When your branch needs testing, merge it into the release branch for the related sprint (say `release/1.2`) using a merge commit (`git merge --no-ff`).

Step 5: Deploy to a test environment. When the release branch is ready to be tested, merge it into the right non-production deploy branch (for example `deploy/stage`), again with `--no-ff`. Tag the commit (say `v1.2.0-alpha.1`) and push. The push triggers the deploy.

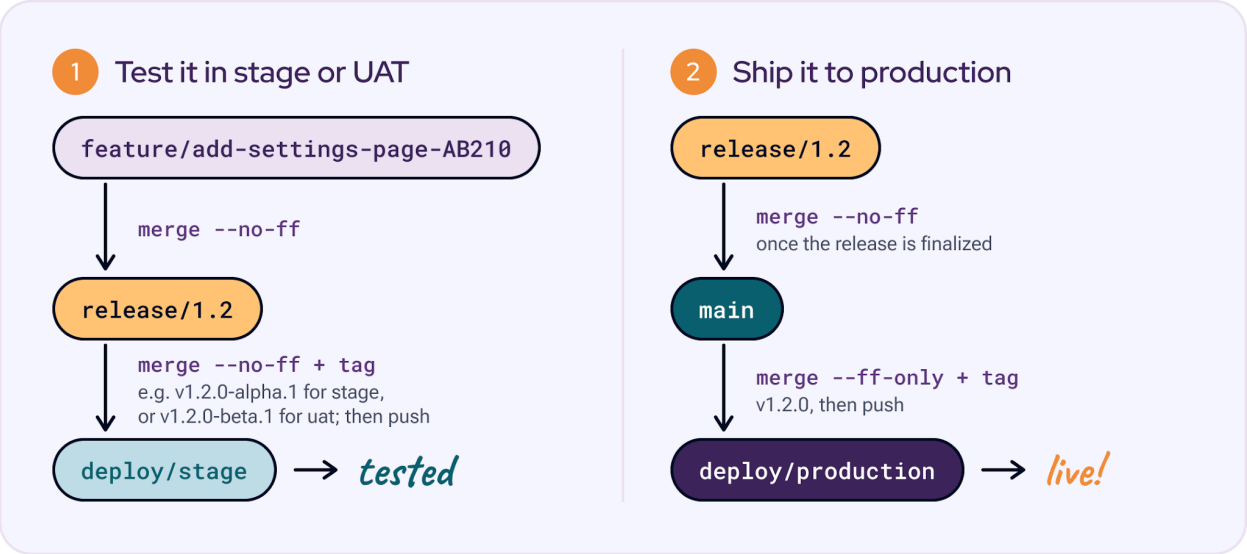
Step 6: Test, fix, repeat. If testers find problems, commit the fixes directly on the related development branch(es), go back to step 4 and merge into the release branch so you can deploy again with a new tag (say `v1.2.0-alpha.2`).

Step 7: Finalize. Once the release branch is tested and approved, merge it into the common branch with `--no-ff`.

Step 8: Go live. Merge the common branch into `deploy/production` using `--ff-only`, tag the commit with the version (`v1.2.0`), and push.

Congrats! Your changes are live.

Here's the flow at a glance:



Notice the pattern: changes only ever flow *forward*, from small branches into bigger ones, and then out to the world. Nothing flows backward.

Chapter 5: The Rules, and the Reasons Behind Them

Our policy has a short list of general rules that we ask everyone to follow strictly. Here they are in plain language, along with the "why" for each. When possible, these should be enforced in your version control software to avoid mistakes being made by new developers learning the rules.

Rule 1: Never delete a core branch

Core branches are the backbone of the project. Deleting one is a bit like removing a load-bearing wall because it was in the way. The consequences range from "annoying" to "everything falls down."

Rule 2: Only lead developers create core branches

Core branches are wired into environments and deployments. Creating one is a structural decision, and we want a lead developer to make that decision on purpose, not by accident.

Rule 3: Always start from the latest common branch

Whenever you create a new development or release branch, first pull down the latest commits, then branch from the newest commit on the common branch. A few related rules go with it:

- **Always create a development branch,** no matter how few commits you expect. "It's just one line" is the most dangerous sentence in software.
- **Always create a release branch,** no matter how few development branches will go into it.
- **Never make commits directly on the common branch or any deploy branch.** Do your work on development branches, always.

Why? Starting from stale code means you're building on a foundation that has already changed. You may accidentally undo someone else's work or run into conflicts that never needed to exist. And committing straight to the common or deploy branches skips review and testing entirely.

Development branches also keep releases flexible: If a feature needs to be removed, you can rebuild the release branch from its starting point and merge back in only the branches you still want. Work committed directly to a release or deploy branch can't be cleanly removed, and it silently disappears when that branch is reset.

Side Story #1

The Friday Typo



It was 4:45 on a Friday afternoon. Dev, a cheerful developer who had been at the company for three weeks, spotted a typo on the checkout page: "Pruchase" instead of "Purchase." It was tiny. It was embarrassing. It would take ten seconds to fix.

"Not worth a whole branch," Dev thought, and committed the fix directly to the `main` branch. While he was at it, he tidied up a nearby line of code that looked a little messy. Then he pushed, closed his laptop, and went home feeling productive.

The tidy-up line, it turned out, wasn't messy at all. It was handling a special case for customers in Canada. By 5:30, every Canadian order was failing at checkout. By 7:00, a client's phone was ringing in the middle of a family dinner. By 7:15, the lead developer was cancelling plans and rolling back changes, and trying to work out which of Dev's changes was the culprit, because nothing had been reviewed, tested, or even described on a pull request.

> *Had Dev made a branch, someone would have looked at that "messy" line and said, "Hey, don't touch that." The branch would have taken two minutes to create. Instead, the Friday night fiasco took about four hours to recover from.*
>
> ***Moral:** There is no change too small for a branch. The branch is the safety net.*

Rule 4: Test through the release branch, and always make a merge commit

When a development branch needs to be tested, merge it into its sprint's release branch using `--no-ff`.

Why? The merge commit is a bookmark in your history. It says, "Here is where *this* feature joined the release." If that feature turns out to be problematic, a merge commit makes it far easier to find and to pull back out cleanly. Without it, a feature's commits blend into the surrounding history and become much harder to untangle.

Rule 5: Test a release by merging it into a non-production deploy branch, and tag it

When a release branch needs testing in an environment, first merge it into the matching non-production deploy branch (with `--no-ff`), tag the latest commit, then push to trigger the deploy.

Tags use the **full version number prefixed with a** `v`, followed by a suffix that tells you the environment and how many times you've deployed:

- `stage` uses the `-alpha.` suffix
- `uat` uses the `-beta.` suffix

For the sprint matching version `1.2`, the first deploys would be tagged:

- `v1.2.0-alpha.1` when deployed to `stage`
- `v1.2.0-beta.1` when deployed to `uat`

Every time you deploy again before version 1.2.0 reaches production, the deploy count goes up: `v1.2.0-alpha.2`, `v1.2.0-alpha.3`, and so on.

Why? Tags let anyone answer "exactly what version is on stage right now?" in a few seconds. They also give you a permanent record of each deploy, which is a huge help when someone says, "It worked on the third deploy yesterday, but not the fourth."

Rule 6: Merge approved (finalized) releases into the common branch

Once a release branch has been tested and finalized, merge it into the common branch using `--no-ff`.

Why? The common branch is where finished, trusted work lives. Merging in a release as a single, clearly marked unit keeps the history readable. You can see at a glance that "release 1.2 landed here."

Rule 7: Go to production only through the common branch, and only by fast-forward

When the common branch has commits ready for production, merge it into `deploy/production` and push. Three details matter here:

- **Never merge any other branch into** `deploy/production`**.** Only the common branch.
- **Always use** `--ff-only`**,** so the two branches always contain exactly the same commits.
- **Always tag the commit with the full version number,** prefixed with a `v`. The first release of the `1.2` sprint is `v1.2.0`.

Why? A fast-forward-only merge guarantees one thing: production contains exactly what's on the common branch, nothing more and nothing less. The review and testing happen earlier, through pull requests and the release branch. Together, they mean that production can never contain a surprise. And if a fast-forward isn't possible, Git refuses, which is a loud warning that something is wrong.

Tip: Let your tools enforce it. In GitHub, add a branch protection rule (or ruleset) to `deploy/production` so that only your lead developers can push to it. You can also turn on "Require linear history," which blocks merge commits, so a fast-forward from the common branch is the only way in. That way, the rule doesn't rely on everyone remembering it, and a new developer can't accidentally take a shortcut to production.

Side Story #2

The Mystery Feature



Priya was a talented developer in a hurry. A client needed a small pricing fix live immediately, and the common branch had a couple of unfinished things in it that weren't ready for prime time. "I'll just merge my fix branch straight into `deploy/production`," she reasoned. "That skips the unfinished stuff. Clever!"

It worked. The pricing fix went live. Everyone relaxed.

Two weeks later, the team merged the common branch into `deploy/production` the normal way, and Git put up a wall of conflicts. Production now had commits that the common branch had never heard of, and the common branch had commits that production had never seen. Nobody could be sure what exactly was running live. Worse, when the team sorted it out, they discovered that Priya's fix contained a half-finished discount calculation she had forgotten about. For two weeks, a handful of customers had been quietly getting 15% off everything.

The client was very understanding. The accountant was less so.

***Moral:** Production should only ever be a reflection of the common branch. If the fix is urgent, put it through the process (fast), rather than around it. That's what hotfix branches are for.*

Rule 8: Never merge the common or deploy branches *into* a development or release branch

This one surprises people, so read it carefully. If you're working on a development or release branch and need the latest changes from the common branch, **don't merge the common branch in. Rebase your branch on top of the common branch instead.**

And deploy branches? They must never, under any circumstances, be merged into a development or release branch.

Why? There are two reasons:

1. **Merging the common branch into your branch makes a tangled history.** Merge commits pile up going in both directions, and soon nobody can tell which changes belong to which feature. A rebase replays your work on top of the latest code, producing a straight, easy-to-read line.
2. **Deploy branches are ephemeral.** Remember, they only represent what's deployed to an environment right now, and non-production ones get hard-reset on a regular basis. If you merge one into your branch, you're pulling in a snapshot of something that is designed to vanish.

What if one development branch depends on another?

Sometimes new work needs code from another branch that isn't finished yet. It's tempting to merge that branch into yours so you can keep going. Don't: never merge one development branch into another.

Why? Once two development branches are merged together, they can no longer be tested, released or removed separately. If the first branch is removed from a release before it is deployed, its changes are still hiding inside the second one, waiting to resurface.

Instead, use one of these approaches:

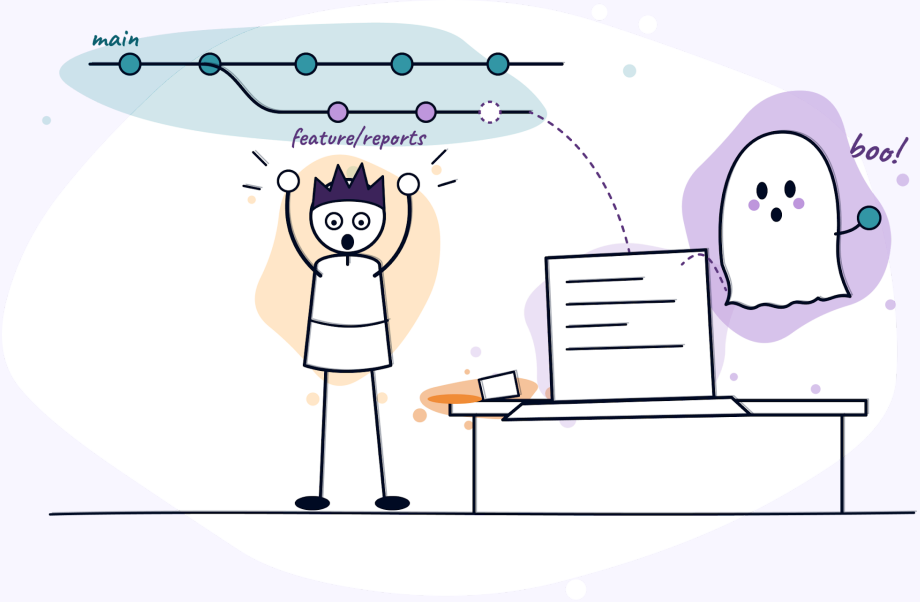
1. **Finish first, then branch.** If the branch you depend on can be tested and approved on its own, finish it first. Once it has reached the common branch (when its release is finalized), create your new branch from the latest common branch, and the changes you need will already be there.

2. **Combine them.** If neither piece of work can be tested or approved without the other, they're really one piece of work. Combine them into a single task with a single development branch.

3. **Take turns.** If several pieces of work all depend on the same unfinished change, don't work on them at the same time. Choose one to finish first and combine it with the shared change as in option 2. Once it reaches the common branch, create the remaining branches from the latest common branch.

Side Story #3

The Haunted Branch



Marcus had been working on a big reporting feature for three weeks. His branch had drifted far from the latest code, so, trying to be responsible, he decided to catch up. He glanced at the branches, saw `deploy/stage` *at the top, thought "that's the most recent, right?" and merged it into his feature branch.*

What Marcus didn't know was that `deploy/stage` *held an experimental change from another team, deployed for a quick test and never meant to survive. A few days later, the team did their regular hard reset of the stage branch and the experiment disappeared from*

it. But it hadn't disappeared from Marcus's branch. It had been quietly carried along inside his reporting feature.

Marcus's pull request showed hundreds of changes that had nothing to do with reports. His reviewer was baffled. The release manager was alarmed. The team spent a full afternoon performing something close to an exorcism, carefully pulling a ghost out of Marcus's branch one commit at a time.

***Moral:** Deploy branches show what's running in an environment right now, and that can be wiped away at any time. Never merge them into your own work. If you need fresh code, rebase on the common branch.*

Chapter 6: The Pull Request Process

Branches are only half of the story. The other half is how changes get reviewed. Here's the process we use while a project is still in development (before production launch).

1. **Assign the PR in GitHub and give it the label "Ready for review."** There's no need to assign it in the project management system as well. Keeping it in one place avoids duplicated effort.
2. **Reviewers aim to review assigned PRs within 2 work days.** Nobody should have to wait a week to find out whether their work is okay.
3. **After approval, the reviewer merges changes onward,** and then into the main line on a schedule that depends on the size of the change:
    - **Hot fixes and small changes** (like fixing a typo) can be merged immediately.
    - **Larger changes** are merged around a week after they land in the testing environment. This gives QA and project management staff time to do an initial round of testing on the new functionality.
4. **Coordinate larger merges with your teammates.** A quick conversation up front prevents conflicts and complications later.
5. **Reviewing developers schedule the larger merges themselves,** putting them on their calendars so they happen at an appropriate time and don't get forgotten.

Why review at all? A second pair of eyes catches mistakes, spreads knowledge around the team, and keeps quality consistent. It's also the perfect moment to ask, "Does this branch name follow the policy?" and "Did you start from the latest code?"

Chapter 7: Making It Your Own

Now that you've seen how we do it, here are some questions to ask when adapting this policy for your own team.

How many environments do you really have? We use several, including `stage` and `uat`. If you only have staging and production, you only need `deploy/stage` and `deploy/production`. Don't build structure that you don't need.

How often do you release? Most of our release branches map to software sprints. If you deploy continuously, you might skip release branches and merge reviewed feature branches straight to the common branch. If you release rarely, you might keep release branches longer.

How big is your team? Rules like "only leads create core branches" matter more when many people have access. On a team of two, you might loosen some of the formality, but keep the habits that protect production.

What tools do you use for task tracking? We use Asana as our project management system, so we include short Asana task IDs in our branch names. If you use Jira, Linear, or something else, swap in those IDs.

Merge or rebase? We use merge commits at some stages and rebases at others. Some teams prefer rebasing everything, and others prefer merging everything. Pick a convention and write it down.

Whatever you change, here are the principles we'd suggest holding onto:

- **Production should only ever contain reviewed, tested, merged code.**
- **All work happens on development branches, never directly on the core branches.**
- **Names and tags should tell a story,** so anyone can understand them at a glance.
- **The policy should be written down,** and everybody should know where to find it.
- **Rules should have reasons.** If nobody can explain why a rule exists, it's time to either explain it or remove it.

Quick Reference Cheat Sheet

Branch names

Type	Format	Example
Feature	`feature/<slug>-<TASK-ID>`	`feature/study-view-optimizations-AB889`
Bug fix	`hotfix/<slug>-<TASK-ID>`	`hotfix/fix-login-error-AB421`
Release	`release/<major>.<minor>`	`release/1.2`
Deploy	`deploy/<environment>`	`deploy/stage`

Tag formats

Where	Format	Example
`stage`	`v<version>-alpha.<count>`	`v1.2.0-alpha.1`
`uat`	`v<version>-beta.<count>`	`v1.2.0-beta.1`
`production`	`v<version>`	`v1.2.0`

How to merge

From → To	How
Development → Release	`--no-ff`
Release → Non-production deploy	`--no-ff` and tag
Release → Common branch	`--no-ff`
Common branch → `deploy/production`	`--ff-only` and tag
Need the latest common branch commits?	**Rebase**, never merge
Development → another development branch	**Never** (see Rule 8)

The Golden Rules

1. Never delete a core branch
2. Only lead developers create core branches
3. Always branch from the latest commit on the common branch
4. Test through the release branch, and always make a merge commit
5. Test a release by merging it into a non-production deploy branch, and tag it
6. Merge finalized releases into the common branch
7. Deploy to production only by merging the common branch, and only by fast-forward
8. Never merge the common or deploy branches into a development or release branch

A Final Word

A branching policy isn't about bureaucracy. It's about trust: trusting that what's in production is what you think it is, trusting that your teammates' work won't collide with yours, and trusting that when something does go wrong, you'll be able to find out why quickly.

If you're part of a team without a policy, we hope this gives you a starting point. If you already have one, we hope it sparks some ideas. And if you'd like help setting up a workflow that suits your team, or you're looking for a development partner who takes this kind of thing seriously, we'd love to chat.

SiteRocket Labs

www.siterocket.com