Commit Often, Perfect Later, Publish Once: Git Best Practices

Best Practices vary from environment to environment, and there is no One True Answer, but still, this represents a consensus from #git and in some cases helps you frame the discussion for the generation of your very own best practices.

Table of Contents

Do read about git

Knowing where to look is half the battle. I strongly urge everyone to read (and support) the Pro Git book. The other resources are highly recommended by various people as well.

- Pro Git
- Git for Computer Scientists and a different/updated version
- Git from the Bottom Up
- The Git Parable
- Other resources
- Git wiki

Do commit early and often

Git only takes full responsibility for your data when you commit. If you fail to commit and then do something poorly thought out, you can run into trouble. Additionally, having periodic checkpoints means that you can understand how you broke something.

People resist this out of some sense that this is ugly, limits `git-bisect`ion functionality, is confusing to observers, and might lead to accusations of stupidity. Well, I'm here to tell you that resisting this is ignorant. *Commit Early And Often*. If, after you are done, you want to pretend to the outside world that your work sprung complete from your mind into the repository in utter perfection with each concept fully thought out and divided into individual concept-commits, well git supports that: see Sausage Making below. However, don't let tomorrow's beauty stop you from performing continuous commits today.

Personally, I commit early and often and then let the sausage making be seen by all except in the most formal of circumstances (public projects with large numbers of users, developers, or high developer turnover). For a less formal usage, like say this document I let people see what really happened.

Don't panic

As long as you have committed your work (or in many cases even added it with `git add`) your work will not be lost for at least two weeks unless you really work at it (run commands that manually purge it).

See on undoing, fixing, or removing commits in git if you want to fix a particular problematic commit or commits, as opposed to attempting to locate lost data.

When attempting to find your lost commits, first make *sure* you will not lose any current work. You should commit or stash your current work before performing any recovery efforts that might destroy your current work and perhaps take backups of it (see Backups below). After finding the commits you can `reset`, `rebase`, `cherry-pick`, `merge`, or otherwise do what is necessary to get the commit history and work tree you desire.

There are three places where "lost" changes can be hiding. They might be in the reflog (`git log -g`), they might be in lost&found (`git fsck --unreachable`), or they might have been stashed (`git stash list`).

- reflog

  The reflog is where you should look first and by default. It shows you each commit that modified the git repository. You can use it to find the commit name (SHA-1) of the state of the repository before (and after) you typed that command. While you are free to go through the reflog manually (`git log -g`) or searching it (`git log -Sfoo -g`), you can also visualize the repository using the following command (Look for dots without children and without green labels):

  ```
  gitk --all --date-order $(git log -g --pretty=%H)
  ```

- Lost and found

  Commits or other git data that are no longer reachable through any reference name (branch, tag, etc) are called "dangling" and may be found using `fsck`. There are legitimate reasons why objects may be dangling through standard actions and normally over 99% of them are entirely uninteresting for this reason.

  - Dangling Commit

    These are the most likely candidates for finding lost data. A dangling commit is a commit no longer reachable by any branch or tag. This can happen due to resets and rebases and are normal. `git show SHA-1` will let you inspect them.

    The following command helps you visualize these dangling commits. Look for dots without children and without green labels.

    ```
    gitk --all --date-order $(git fsck | grep "dangling commit" | awk '{print $3;}')
    ```

  - Dangling Blob

    A dangling blob is a file that was not attached to a commit. This is often caused by `git add`s that were superceded before commit or merge conflicts. Inspect these files with

    ```
    git show SHA-1
    ```

  - Dangling Tree

    A dangling tree is a directory tree of files that was not attached to a commit. These are rarely interesting, and often caused by merge conflicts. Inspect these files with `git ls-tree -r SHA-1`

- Stashes

  Finally, you may have stashed the data instead of committing it and then forgotten about it. You can use the `git stash list` command or inspect them visually using:

  ```
  gitk --all --date-order $(git stash list | awk -F: '{print $1};')
  ```

- Misplaced

  Another option is that your commit is not lost. Perhaps the commit was just made on a different branch from what you remember. Using `git log -Sfoo --all` and `gitk --all --date-order` to try and hunt for your commits on known branches.

- Look elsewhere

  Finally, you should check your backups, testing copies, ask the other people who have a copy of the repo, and look in other repos.

Do backups

Everyone always recommends taking backups as best practice, and I am going to do the same. However, you already may have a highly redundant distributed ad-hoc backup system in place! This is because essentially every clone is a backup. In many cases, you may want to use a clone for git experiments to perfect your method before trying it for real (this is most useful for `git filter-branch` and similar commands where your goal is to permanently destroy history without recourse—if you mess it up you may not have recourse). Still, probably you need a more formal system as well.

Traditional backups are still appropriate, and clones do not save git configurations, the working directory and index, non-standard refs, or dangling objects anyway. A normal tarball, `cp`, `rsync`, `zip`, `rar` or similar backup copy will be a perfectly fine backup. As long as the underlying filesystem doesn't reorder git I/O dramatically and there is not a long time delay between the scanning of the directory and the retrieval of the files, the resulting copy of `.git` should be consistent under almost all circumstances including if taken while git operations are in progress, though you might have to do some recovery--the data will all be present. See also discussions about custom backup techniques to ensure git consistency--though it does not mention `git bundle create ... --all` which appears to be the only atomic native git command that can create a backup. When performing git experiments involving items other than normally reachable commits, a copy instead of a clone may be more appropriate.

However, if you want a "pure git" solution that clones everything in a directory of repos, something like this may be what you need:

```
cd /src/backupgit
ls -F . | grep / > /tmp/.gitmissing1
ssh -n git.example.com ls -F /src/git/. | grep / > /tmp/.gitmissing2
diff /tmp/.gitmissing1 /tmp/.gitmissing2 | egrep '^>' |
  while read x f; do
    git clone --bare --mirror ssh://git.example.com/src/git/$$f $$f
  done
rm -f /tmp/.gitmissing1 /tmp/.gitmissing2
for f in */.; do (cd $$f; echo $$f; git fetch); done
```

Don't change published history

Once you `git push` (or in theory someone pulls from your repo, but people who pull from a working repo often deserve what they get) your changes to the authoritative upstream repository or otherwise make the commits or tags publicly visible, you should ideally consider those commits etched in diamond for all eternity. If you later find out that you messed up, make new commits that fix the problems (possibly by revert, possibly by patching, etc).

Yes, of course git allows you to rewrite public history, but it is problematic for everyone and thus it is just not best practice to do so.

I've said it and I believe it, but…on occasion…if well managed…there are times when changing published history is perhaps a normal course of business. You can plan for particular branches (integration branches especially) or (better) special alternate repositories to be continually rewritten as a matter of course. You see this in git.git with the "pu" branch, for example. Obviously this process must be well controlled and ideally done by one of the most experienced and well trusted engineers (because auditing merges (and worse, non-merge commits) you have essentially seen before is extremely tedious, boring, and error prone and you have lost the protection of git's cryptographic history validation).

Do choose a workflow

Some people have called git a tool to create a SCM workflow instead of an SCM tool. There is some truth to this. I am not going to specifically espouse one specific workflow as the best practice for using git since it depends heavily on the size and type of project and the skill of users, developers, and release engineers; however both reflexive avoidance of branches due to stupidity of other SCM systems and reflexive overuse of branches (since branches are actually easy with git) is most likely ignorance. Pick the style that best suits your project and don't complain about user's tactical uses of private branches.

I also implore managers who may be thinking of making specific workflow rules by fiat to remember that not all projects are identical, and rules that work on one project may not work on another. People who blather on about continuous integration, rolling deployment, and entirely independent feature changes that you can pick and choose between independently are absolutely correct, *for their project!* However, there are many projects and features which are much more complicated and may take longer than a particular sprint/unit-of-time and require multiple people to complete and have complex interdependencies with other features. It is not a sign of stupidity but rather of complexity and, just perhaps, brilliant developers, who can keep it all straight. It can also lead to a market advantage since you can deploy a differentiating feature which your competitors cannot in a short timeframe.

Branch workflows

Answering the following questions helps you choose a branch workflow:

- Where do important phases of development occur?
- How can you identify (and backport) groups of related change?
- Do you have work which often needs to be updated in multiple distinct long-lived branches?
- What happens when emergency patches are required?
- What should a branch for a particular purpose (including user-tactical) be named?
- What is the lifecycle of a branch?

See the following references for more information on branch workflows.

- Pro Git branching models
- Git-flow branching model (with the associated gitflow tool)
- Gitworkflows man page
- A Git Workflow for Agile Teams
- What git branching models actually work
- Our New Git Branching Model
- Branch-per-Feature
- Who Needs Process

However, also understand that everyone already has an implicit private branch due to their cloned repository: they can do work locally, do a `git pull --rebase` when they are done, perform final testing, and then push their work out. If you run into a situation where you might need the benefits of a feature branch before you are done, you can even retroactively commit&branch then optionally reset your primary branch back to `@{u}`. Once you push you lose that ability.

Some people have been very successful with just master and $RELEASE branches ($RELEASE branch for QA and polishing, master for features, specific to each released version.) Other people have been very successful with many feature branches, integration branches, QA, and release branches. The faster the release cycle and the more experimental the changes, the more branches will be useful—continuous releases or large refactoring project seem to suggest larger numbers of branches (note the number of branches is the tail, not the dog: more branches will not make you release faster).

The importance of some of the questions I asked may not be immediately obvious. For example, how does having work which needs to be updated in multiple distinct long-lived branches affect branch workflow? Well, you may want to try to have a "core" branch which these other branches diverge from, and then have your feature/bugfix branches involving these multiple branches come off of the lowest-common-merge-base (LCMB) for these long-lived branches. This way, you make your change (potentially merge your feature branch back into the "core" branch), and then merge the "core" branch back into all of the other long-lived branches. This avoids the dreaded cherry-pick workflow.

Branch naming conventions are also often overlooked. You must have conventions for naming release branches, integration branches, QA branches, feature branches (if applicable), tactical branches, team branches, user branches, etc. Also, if you use share repositories with other projects/groups, you probably will need a way to disambiguate your branches from their branches. Don't be afraid of "/" in the branch name when appropriate (but do be afraid of using a remote's name as a directory component of a branch name, or correspondingly naming a remote after a branch name or directory component).

Distributed workflows

Answering the following questions helps you choose a distributed workflow:

- Who is allowed to publish to the master repository?
- What is the process between a developer finishing coding and the code being released to the end-user?
- Are there distinct groups that work on distinct sections of the codebase and only integrate at epochs? (Outsourcing)
- Is everyone inside the same administrative domain?

See the following references for more information on distributed workflows.

- Pro Git distributed models
- Gitworkflows man page

Cathedrals (traditional corporate development models) often want to have (or to pretend to have) the one true centralized repository. Bazaars (linux, and the Github-promoted workflow) often want to have many repositories with some method to notify a higher authority that you have work to integrate (pull requests).

However, even if you go for, say, a traditional corporate centralized development model, don't forbid self-organized teams to create their own repositories for their own tactical reasons. Even having to fill out a justification form is probably too cumbersome.

Release tagging

Choosing your release workflow (how to get the code to the customer) is another important decision. You should have already considered most of the issues when going over the branching and distributed workflow above, but less obviously, it may affect how and when you perform tagging, and specifically the name of the tag you use.

At first glance, it is a no-brainer. When you release something you tag something, and of course I *highly* recommend this. However, tags should be treated as immutable once you push. Well, that only makes sense, you might think to yourself, but consider this: five minutes after everyone has signed off on the 2.0 release, it has been tagged `Frobber_Release_2.0` and pushed, but before any customer has seen the resulting product someone comes running in "OMFG, the foobar is broken when you frobnoz the baz." What do you do? Do you skip release `2.0` and tag `2.0.1`? Do you do a take-back and go to every repo of every developer and delete the `2.0` tag?

Two ideas for your consideration. Instead of a release tag, use a release branch with the marketing name (and then stop committing to that branch after release, disabling write access to it in gitolite or something). Another idea, use an internal tag name that is not directly derived from the version number that marketing wishes to declare to the outside world. The problem with the branch idea is that if you cannot (or forget to) disable write access then someone might accidentally commit to that branch, leading to confusion about what was actually released to the customer. The problem with the tag idea is that you need to remember what the final shipped tag name is, independent from the release name. However, if you use both techniques, they cover for each other's disadvantages. In any case, using either technique will be better than using marketing-version tags (as I know from experience).

Security model

You might ask why security is not a top level item and is near the end of the workflow section. Well that is because in an ideal world your security should support your workflow not be an impediment to it.

For instance, did you decide certain branches should only have certain people being allowed to access it? Did you decide that certain repositories should only have certain people able to access/write to them?

While git allows users to set up many different types of access control, access methods, and the like; the best for most deployments might be to set up a centralized git master repository with a gitolite manager to provide fine grained access control with ssh based authentication and encryption.

Of course, security is more than access control. It is also assurance that what you release is what was written by the people it should be written by, and what was tested. Git provides you this for free, but certain formal users may wish to use signed tags. Watch for signed pushes in a future version of git.

Do divide work into repositories

Repositories sometimes get used to store things that they should not, simply because they were there. Try to avoid doing so.

- One conceptual group per repository.

  Does this mean one per product, program, library, class? Only you can say. However, dividing stuff up later is annoying and leads to rewriting public history or duplicative or missing history. Dividing it up correctly beforehand is much better.

- Read access control is at the repo level

  If someone has access to a repository, they have access to the entire repo, all branches, all history, everything. If you need to compartmentalize read access, separate the compartments into different repositories.

- Separate repositories for files that might be needed by multiple projects

  This promotes sharing and code reuse, and is highly recommended.

- Separate repositories for large binary files

  Git doesn't handle large binary files ideally yet and large repositories can be slow. If you must commit them, separating them out into their own repository can make things more efficient.

- Separate repositories for planned continual history rewrites

  You will note that I have already recommended against rewriting public history. Well, there are times when doing that just makes sense. One example might be a cache of pre-built binaries so that most people don't need to rebuild them. Yet older versions of this cache (or at least older versions not at tag boundaries) may be entirely useless and so you want to pretend they never happened to save space. You can rebase, filter, or squash these unwanted commits away, but this is rewriting history and can cause problem. So if you really must do so, isolate these files into a repository so that at least everything else will not be affected.

- Group concepts into a superproject

  Once you have divided, now you need to conquer. You can assemble multiple individual repositories into a superproject to group all of the concepts together to create your unified work.

  There are two main methods of doing this.

  - `git-submodules`

    Git submodules is the native git approach, which provides a strong binding between the superproject repository and the subproject repositories for every commit. This leads to a baroque and annoying process for updating the subproject. However, if you do not control the subproject (solvable by "forking") or like to perform blame-based history archeology where you want to find out the absolute correspondence between the different projects at every commit, it is very useful.

  - `gitslave`

    gitslave is a useful tool to add a subsidiary git repositories to a git superproject when you control and develop on the subprojects at more or less the same time as the superproject, and furthermore when you typically want to tag, branch, push, pull, etc all repositories at the same time. There is no strict correspondence between superproject and subproject repositories except at tag boundaries (though if you need to look back into history you can usually guess pretty well and in any case this is rarely needed).

Do make useful commit messages

Creating insightful and descriptive commit messages is one of the best things you can do for others who use the repository. It lets people quickly understand changes without having to read code. When doing history archeology to answer some question, good commit messages likewise become very important.

The normal git rule of using the first line to provide a short (50-72 character) summary of the change is also very good. Looking at the output of `gitk` or `git log --oneline` might help you understand why.

Also see A Note About Git Commit Messages for even more good ideas.

While this relates to the later topic of integration with external tools, including bug/issue/request tracking numbers in your commit messages provides a great deal of associated information to people trying to understand what is going on. You should also enforce your standards on commit messages, when possible, through hooks. See Enforcing standards below.

On Sausage Making

Some people like to hide the sausage making [1], or in other words pretend to the outside world that their commits sprung full-formed in utter perfection into their git repository. Certain large public projects demand this, others demand smushing all work into one large commit, and still others do not care.

A good reason to hide the sausage making is if you feel you may be cherry-picking commits a lot (though this too is often a sign of bad workflow). Having one or a small number of commits to pick is much easier than having to find one commit here, one there, and half of this other one. The latter approach makes your problem much much harder and typically will lead to merge conflicts when the donor branch is finally merged in.

Another good reason is to ensure each commit compiles and/or passes regression tests, and represents a different easily understood concepts. The former allows `git-bisect` to choose any commit and have a good chance of that commit doing something useful, and the latter allows for easy change/commit/code review, understanding, archeology, and cherry-picking. When reviewing commits, for example the reviewer might see something suspicious in a commit and then have to spend time tracking down their suspicions and write them up, only to discover five commits later that the original developer subsequently found and fixed the problem, wasting the reviewer's time (reviewing the entire patch series as a diff fixes this problem but greatly adds complexity as multiple concepts get muddled). By cleaning up patches into single, logical changes that build on one another, and which don't individually regress (i.e., they are always moving towards some desirable common endpoint), the author is writing a chronological story not of what happened, but what *should* happen, with the intent that the audience (i.e., reviewers) are convinced that the change is the right thing to do. Proponents claim it is all about leaving a history others can later use to understand *why* the code became the way it is now, to make it less likely for others to break it.

The downside to *hiding the sausage* making is the added time it takes to perfect the administrative parts of the developers job. It is time taken away from getting code working; time solely dedicated to either administrative beauty or enhancing the ability to perform the blame-based (or ego-full) development methodology.

If you think about it, movies are made this way. Scenes are shot out of temporal order, multiple times, and different bits are picked from this camera and that camera. Without examining the analogy too closely, this is similar to how different git commits might be viewed. Once you have everything in the "can" (repository) you go back and in post-production, you edit and splice everything together to form individual cuts and scenes, sometimes perhaps even doing some digital editing of the resulting product.

`git rebase -i`, `git add -p`, and `git reset -p` can fix commits up in post-production by splitting different concepts, merging fixes to older commits, etc. See Post-Production Editing using Git also TopGit and StGit.

Be sure you do all of this work *before* doing any non-squashed merges (not rebases: merges) and *before* pushing. Your work becomes much more complex and/or impossible afterwards.

[1] The process of developing software, similar to the process of making sausage, is a messy messy business[2]; all sorts of stuff happens in the process of developing software. Bugs are inserted into the code, uncovered, patched over. The end result may be a tasty program, but anyone looking at the process of how it was created (through inspection of the commits) may end up with an sour taste in their mouth. If you hide the sausage making, you can create a beautiful looking history where each step looks as delicious as the end-product. Back to footnote reference.

[2] If you do not understand why someone would want to hide the sausage making, and you enjoy eating sausage, never, ever, watch sausages being made, read "The Jungle", or otherwise try to expose yourself to any part of the sausage making process. You will lead a much tastier (and perhaps shorter) life in your blissful ignorance.

Do keep up to date

This section has some overlap with workflow. Exactly how and when you update your branches and repositories is very much associated with the desired workflow. Also I will note that not everyone agrees with these ideas (but they should!)

- Pulling with `--rebase`

  Whenever I pull, under most circumstances I `git pull --rebase`. This is because I like to see a linear history (my commit came after all commits that were pushed before it, instead of being developed in parallel). It makes history visualization much simpler and `git bisect` easier to see and understand.

  A specific circumstance in which you should avoid using `git pull --rebase` is if you merged since your last push. You might want to `git fetch; git rebase -p @{u}` (and check to make sure the merge was recreated properly) or do a normal merge in that circumstance.

  Another specific circumstance is if you are pulling from a non-authoritative repository which is not fully up to date with respect to your authoritative upstream. A rebase in this circumstance could cause the published history to be rewritten, which would be bad.

  Some people argue against this because the non-final commits may lose whatever testing those non-final commits might have had since the deltas would be applied to a new base. This in turn might make git-bisect's job harder since some commits might refer to broken trees, but really this is only relevant to people who want to hide the sausage making. Of course to *really* hide the sausage making you should still rebase and then test each intermediate commit to ensure it compiles and passes your regression tests (you do have regression tests, don't you?) so that a future bisector will have some strong hope that the commit will be usable. After all, that future bisector might be you.

  Other people argue against this (especially in highly decentralized environments) because doing a merge explicitly records who performed the merge, which provides someone to blame for inadequate testing if two histories were not combined properly (as opposed to the hidden history with implicit blame of rebase).

  Still others argue that you are unable to automatically discover when someone else has rewritten public history if you use `git pull --rebase` normally, so someone might have hidden something malicious in an older (presumably already reviewed) commit. If this is of concern, you can still use rebase, but you would have to `git fetch` first and look for "forced update" in that output or in the reflog for the remote branches.

  You can make this the default with the "branch.<name>.rebase" configuration option (and more practically, by the "branch.autosetuprebase" configuration option). See man git-config.

- Rebasing (when possible)

  Whenever I have a private branch that I want to update, I use rebase (for the same reasons as above). History is clean and simple. However, if you share this branch with other people, rebasing is rewriting public history and should/must be avoided. You may only rebase commits that no-one else has seen (which is why `git pull --rebase` is safe).

- Merging without speeding

  `git merge` has the concept of fast-forwarding, or realizing that the code you are trying to merge in is identical to the result of the code after the merge. Thus instead of doing work, creating new commits, etc, git simply changes the branch pointers (fast forwards them) and calls it good.

  This is good when doing `git pull` but not so good when doing `git merge` with a non-@{u} (upstream) branch. The reason this is not good is because it loses information. Specifically it loses track of which branch is the first parent and which is not. If you don't ever want to look back into history, then it does not matter. However, if you want to know the branch on which a commit was originally made, using fast-forward makes that question impossible to answer. If you try, git will pick one branch or the other (the first parent or second parent) as the one on which both branches' activities were performed and the other (original) parent's branch will be anonymous. There are typically worse things in the world, but you lose information that is not recoverable in any other way by a repository observer and in my book that is bad. Use `git merge --no-ff` instead.

Do periodic maintenance

The first two items should be run on your server repositories as well as your user repositories.

- Validate your repo is sane (`git fsck`)

  You need not check dangling objects unless you are missing something

- Compact your repo (`git gc` and `git gc --aggressive`)

  This will removed outdated dangling objects (after the two+ week grace period). It will also compress any loose objects git has added since your last gc. git will run a minimal gc automatically after certain commands, but doing a manual gc often (and "--aggressive" every few hundred changesets) will save space and speed git operations.

- Prune your remote tracking branches (`git remote update --prune`)

  This will get rid of any branches that were deleted upstream since you cloned/pruned. It normally isn't a major problem one way or another, but it might lead to confusion.

- Check your stash for forgotten work (`git stash list`)

  If you don't do it very often, the context for the stashed work will be forgotten when you finally do stumble on it, creating confusion.

Do enforce standards

Having standards is a best practice and will improve the quality of your commits, code-base, and probably enhance git-bisect and archeology functionality, but what is the use of a standard if people ignore them? Checks could involve regression tests, compilation tests, syntax/lint checkers, commit message analysis, etc. Of course, there are times when standards get in the way of doing work, so provide some method to temporarily disable the checks when appropriate.

Traditionally, and in some people's views ideally, you would enforce the checks on the client side in a pre-commit hook (perhaps have a directory of standard hooks in your repo and might ask users to install them) but since users will often not install said hooks, you also need to enforce the standards on the server side. Additionally, if you follow the commit-early-and-often-and-perfect-it-later philosophy that is promoted in this document, initial commits may *not* satisfy the hooks.

Enforcing standards in a update hook on the server allows you to reject commits that don't follow the standards. You can also chide the user for not using the standard client-side hook to begin with (if you recommend that approach).

See Puppet Version Control for an example for a "Git Update Hook" and "Git Pre-Commit Hook" that enforces certain standards. Note that the update hook is examining files individually instead of providing whole-repository testing. Whether individual files can be tested in isolation for your standards or whether you need the whole repository (for instance, any language where one file can reference or include another might need whole repository checks) is of course a personal choice. The referenced examples are useful for ideas, anyway.

Do use useful tools

More than useful, use of these tools may help you form a best practice!

- gitolite

  We already mentioned gitolite above, but it forms a great git server intermediary for access control.

- gitslave

  We already mentioned gitslave above, but it forms a great alternative to git-submodules when forming superprojects out of repositories you control.

- gerrit

  To quote the website: Gerrit is a web based code review system, facilitating online code reviews for projects using the Git version control system.

Do integrate with external tools

Increasing communication and decreasing friction and roadblocks to your developer's work will have many advantages. If you make something easy, convenient, and useful to do, people might just well do it.

- Web views

  This is pretty standard stuff, but still a best practice. Setting up a tool like gitweb (or cgit or whatever) to allow URL reference to commits (among other visualization interfaces it provides) gives people a great way to refer to commits in email and conversations. If someone can click on a link vs having to fire up git and pull down the latest changes and start up some visualization tool they are much more likely to help you.

- Bug tracking

  Industry best practice suggests that you should have a bug tracking system. Hopefully you do. Well, I'm here to tell you that integrating your bug tracking system with git makes the two systems one thousand times more effective. Specifically, come up with a standard for tagging commits with bug numbers (eg. "Bug 1234: Adjust the frobnoz down by .5") and then have a receive hook on the upstream repo that automatically appends that commit information to the ticket. If you really love your developers, develop syntax that lets them close the ticket as part of the commit message (eg. "Bug 1235r: Adjust the frobnoz up by .25").

  While it probably will be a side-effect of the git integration, ensuring that your ticketing system has an email interface for ticket creation and so that replies to ticket email get stored in the ticket are all very important for making a ticketing system useful and convenient to use.

The easier a system is for people to use, the more likely they will use it. Being able to see the context in which a commit was made (or conversely, being able to find the commit that solved a problem) is incredibly useful. When you send out your commit announcements, make sure to hyperlink the bug tracker in the commit message, and likewise in the tracker message, hyperlink to the web view of the commit.

Notes: some commits can apply to multiple bugs. Generate a standard and code to handle this standard. Also, if you do hours tracking, you may want a syntax to handle that. (eg. "Bug 12346w/5: Bug 12347rw/3: Adjust the frobnoz up by .3")

- IRC/chat rooms/bots

  Having an IRC server with some standard channels to discuss issues and problems provides a great benefit both tactically and strategically (it helps teach both the questioner and the answerer). Adding a robot in those chat room to provide assistance adds significant value. When someone talks about Bug 1234, the bot can provide a hyperlink to that ticket. When someone pushes some commits or adds a bug, it could announce those facts. All sorts of things are possible (RFC lookups, MAC vendor lookups, Eliza psychoanalysis, etc) but there is a fine line between usefulness and overwhelming noise.

  If you use github, github provides an "IRC" "Service Hook" that lets you get git announcements for free. Said robot will not provide any additional value added services you might want. CIA is another packaged commit announcement system and of course many IRC robot frameworks can be found with simple web searches.

- Wikis

  Having convenient syntax to link to bugs in your bug tracking system and branches/tags/commits in your gitweb again makes all three systems more useful. Increased synergy through integration!

- Other services

  While I have your attention, there are a few other services which you should probably bring up as a best practice for your organization (but which have only limited applicability to git):

  - Pastebin-like private text paste service

    The free internet pastebin services are great and very useful, but you cannot paste proprietary information into them. So bring one of those services up yourself. Ensure that you have a portable application interface to this service to reduce friction (`git status | mypastebin`).

  - Imagebin-like private image paste service

    Likewise, bringing up a image paste service and associated client has the same justification and the same benefit. While the obvious use is for screenshots, other uses will become apparent. Just do it.

  - URL shortener

    The justification for a URL shortener is a little weaker than text/image paste services, but it still exists. Since it is a trivial service, you might as well bring it up for proprietary URLs.

  - Search

    Providing a search service for both the web services you provide, but also for git repositories and any other collections of documents or data you might have lying around will save your users hours of searching. Don't forget to enable locate (for Unix/Linux systems) while you are at it.

  - Mailing lists

    Creating mailing lists with searchable archives for users to communicate improves transparency and information flow. Ideally in many cases the mailing lists should be self-service and have digest options. Of course git commits, and bug announcements, and bug updates should all go to such mailing lists.

  - Role aliases

    Instead of saying, send mail to Bob for your sales questions and Alice for your IT problems, you should have well defined and findable role aliases so that vacations, personnel changes, and related issues do not have to affect people's communication behaviors. Of course, in many cases, you may want to have these roles feed into your ticketing system for tracking instead of depending on human management of private mail queues.

  - VNC-sharing of server consoles

    When you have servers (machines without humans in front of them all of the time), make sure that the console is available virtually. Getting a bog-standard KVM to mediate access to the consoles is good, but what you really need to do is get a VNC enabled KVM or passthrough device (like the AdderLink IPEPS and friends) attached to the KVM so that you can have remote access to the servers. Why VNC and not remote desktop or some java console? Portability and flexibility. If you need multiple users having simultaneous access to the servers, you can get a multi-console multi-server KVM. Remember, if you have to get up out of your seat, you have failed.

    Having a VNC console access allows users to more easily consult with each other on problems.

  - VMs (with VNC sharing) for most services

    Instead of having dedicated hardware, create VMs for your critical services. The VM images then can be more easily backed up, moved around, and have less wasted resources. Of course, don't forget the admonition to have a portable multiuser console system like VNC for console access. Personally, I use KVM-QEMU for my virtualization needs. It is free and works great.

    Having a VNC console access allows users to more easily consult with each other on problems.

  - Audio conference

    An audio conferencing service provides another low-friction method of increasing communication, which speeds development. My sources tell me you can get a plugin for asterisk that will give you free conference services. Since you are of course using VOIP for your communication, you then should have essentially free internal conferencing.

Miscellaneous "Do"s

These are random best practices that are too minor or disconnected to go in any other section.

- Copy/move a file in a different commit from any changes to it

  If you care about git properly displaying the fact that you moved a file, you should copy or move the file in a different commit from any changes you need to immediately make to that file. This is because git does not record `git mv` any different from a delete and an add, and because `git cp` doesn't even exist. Git's output commands are the ones that interpret the data as a move or copy. See the -C and -M options to `git log` (and similar commands).

- (Almost) Always name your stashes

  If you don't provide a name when stashing, git generates one automatically based on the previous commit. While this tells you the branch where a stash was made, it gives you no idea what is in it. Unless you plan to pop a stash in the next few minutes, you should always give it a name with `git stash save XXX` rather than the shorter default `git stash` (which should be reserved for very temporary uses). This way you'll have some idea what a stash is about when you are looking at it months later.

- Protect your bare/server repos against history rewriting

  If you initialize a bare git repository with "--shared" it will automatically get the git-config "receive.denyNonFastForwards" set to true. You should ensure that this is set just in case you did something weird during initialization. Furthermore, you should also set "receive.denyDeletes" so that people who are trying to rewrite history cannot simply delete the branch and then recreate it. Best practice is for there to be a speedbump any time someone is trying to delete or rewrite history, since it is such a bad idea.

- Experiment!

  When you have an idea or are not sure what something does, try it out! Ideally try it out in a clone or copy so that recovery is trivial. While you can normally completely recover from any git experiment involving data that has been fully committed, perhaps you have not committed yet or perhaps you are not sure whether something falls in the category of "trying hard" to destroy history.

Miscellaneous "don't"s

In this list of things to *not* do, it is important to remember that there are legitimate reasons to do all of these. However, you should not attempt any of these things without understanding the potential negative effects of each and why they might be in a best practices "Don't" list.

DO NOT

- commit anything that can be regenerated from other things that were committed.

  Things that can be regenerated include binaries, object files, jars, `.class`, flex/yacc generated code, etc. Really the only place there is room for disagreement about this is if something might take hours to regenerate (rendered images, e.g., but see Dividing work into repositories for more best practices about this) or autoconf generated files (so people can configure and compile without autotools installed).

- commit configuration files

  Specifically configuration files that might change from environment to environment or for any reasons. See Information about local versions of configuration files

- use git as a web deployment tool

  Yes it can be done in a sufficiently simple/non-critical environment with something like Abhijit Menon-Sen's document on using git to manage a web site to help, though there are other examples. However, this does not give you atomic updates, synchronized db updates, or other accouterments of an industrial deployment system.

- commit large binary files (when possible)

  Large is currently relative to the amount of free RAM you have. Remember that not everyone may be using the same memory configuration you are. Support for large files is an active git topic, so watch for changes.

  After running a `git gc` you should be able to find the largest objects by running:

  ```
  git verify-pack -v .git/objects/pack/pack-*.idx |
  grep blob | sort -k3nr | head |
  while read s x b x; do
    git rev-list --all --objects | grep $s |
    awk '{print "'"$b"'",$0;}';
  done
  ```

  Consider using Git annex or Git media if you plan on having large binary files and your workflow allows.

- create very large repositories (when possible)

  Git can be slow in the face of large repositories. The definition of "large" will depend on your RAM size, I/O speed, expectations, etc. However, having 100K-200K files in a repository may slow common operations due to stat system call speeds (especially on Windows) and having many large (esp. binary) files (see above) can slow many operations.

  If you start having pack files (in `.git/objects/pack`) which are larger than 1GB, you might also want to consider creating a `.keep` file (right beside the `.idx` and `.pack` files) for a large pack which will prevent them from being repacked during gc and repack operations.

  If you find yourself running into memory pressure when you are packing commits (usually `git gc [--aggressive]`), there are git-config options that can help. `pack.threads=1` `pack.deltaCacheSize=1` `pack.windowMemory=512m` all of which trade memory for CPU time. Other likely ones exist. My gut tells me that sizing ("deltaCacheSize" + "windowMemory" + min("core.bigFileThreshold[512m]", TheSizeOfTheLargestObject)) * "threads" to be around *half* the amount of RAM you can dedicate to running `git gc` will optimize your packing experience, but I will be the first to admit that made up that formula based on a very few samples and it could be drastically wrong.

  Support for large repositories is an active git topic, so watch for changes.

- use `reset` (`--hard | --merge`) without committing/stashing

  This can often overwrite the working directory without hope of recourse.

- use `checkout` in file mode

  This will overwrite some (or potentially all with `.`) of the working directory without hope of recourse.

- use `git clean` without previously running with "`-n`" first

  This will delete untracked files without hope of recourse.

- prune the reflog

  This is removing your safety belt.

- expire "now"

  This is cutting your safety belt.

- use `git repack -ad`

  Unreferenced objects in a newly redundant pack will get deleted which cuts your safety belt. Instead use `git gc` or at least `git repack -Ad`

- use a branch argument to `git pull` or `git fetch`

  No doubt there is a good use case for, say, `git pull origin master` or whatever, but I have yet to understand it. What I *do* understand is that every time I have seen someone use it, it has ended in tears.

- use git as a generic filesystem backup tool

  Git was not written as a dedicated backup tool, and such tools do exist. Yes people have done it successfully, but usually with lots of scripts or modifications around it. One successful example of integration/modifications is bup.

- rewrite public history

  See section about this topic. It bears repeating, though.

- change where a tag points

  This is another way to rewrite public history.

- use `git-filter-branch`

  Still another way to rewrite public history.

  However, if you are going to use git-filter-branch, make sure you end your command with `--tag-name-filter cat -- --all` unless you are really really sure you know what you are doing.

- create `--orphan` branches

  With the notable exception of gh-pages (which is a hack github uses for convenience, not an expression of general good style practice), any situation where creating an orphan branch seems like a reasonable solution you probably should just create a new repository. If the new branch cannot really be thought of as being related to the other branches in your repository so that merging between the two really has any conceptual relevance, then the concept is probably far enough apart to warrant its own repository.

- use `clone --shared` or `--reference`

  This can lead to problems for non-normal git actions, or if the other repository is deleted/moved. See git-clone manual page.

- use `git-grafts`

  This is deprecated in favor of `git-replace`.

- use `git-replace`

  But don't use `git-replace` either.

Disclaimer

Copyright

Thanks

Thanks to the experts on #git, and my co-workers, for review, feedback, and ideas.

Comments

Comments and improvements welcome.

Use the github issue tracker or discuss with SethRobertson (and others) on #git

Other technical projects