Git Workflow Discipline

Version 1

By Gavin Davies (gavd.co.uk)

“Seest thou a man diligent in his business? He shall stand before kings”

- *King Solomon*

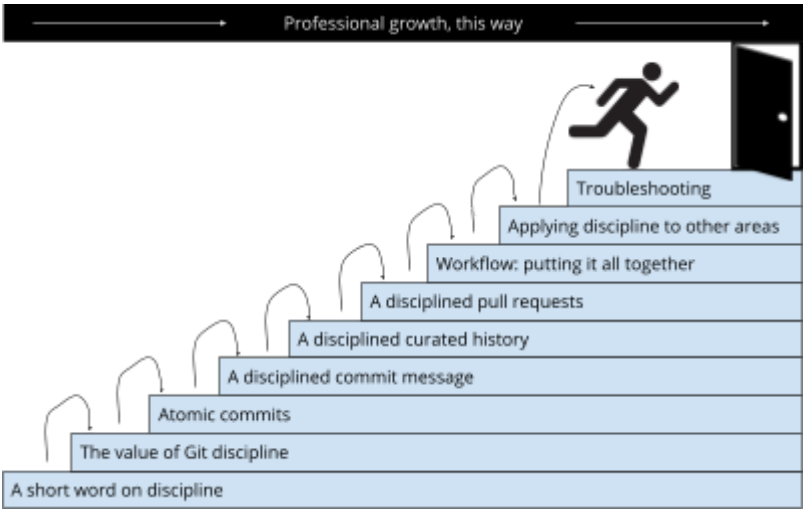
About this book

This short ebook is a highly opinionated, non exhaustive guide to working with Git in a disciplined manner. Highly opinionated, in that I am not attempting to be balanced - I am presenting to you the way that I work and the benefits I have had over many years of using Git. Non exhaustive, in that I am not trying to explore every feature of Git - I'm restricting the scope of this book to a guide to working in a disciplined manner.

The only prerequisite is to have a basic familiarity with Git. If you don't know Git, you can get Scott Chacon's excellent Pro Git book for free from https://git-scm.com/book/en/v2 . Ideally, you'd also be familiar with pull requests (known as merge requests in some Git tools) but that is not essential.

This book is about Git, but the principles are applicable to other source control systems, such as Mercurial.

Each chapter of this book is intended to build upon the last, like a staircase:



How each chapter of this book builds on the last

Using graphics from https://pixabay.com/vectors/the-door-open-doors-1389755/ and https://pixabay.com/vectors/stick-man-runner-silhouette-figure-295293/

A short word on discipline

Many people have a negative reaction to the word “discipline” - they see discipline as the imposition of something they don’t want to do.

There’s another way of understanding discipline - as incredibly liberating. Through discipline, good habits become almost automatic. Through discipline, whole classes of problems simply evaporate, leaving you to focus on other matters.

“Freedom is what everyone wants — to be able to act and live with freedom. But the only way to get to a place of freedom is through discipline. If you want financial freedom, you have to have financial discipline. If you want more free time, you have to follow a more disciplined time management system. You also have to have the discipline to say “No” to things that eat up your time with no payback” - Jocko Willink

Interestingly, discipline is often “upstream” of where you may think it is. It is generally developed through planning, habit, thought and iteration, rather than in the spur of the moment. To give a trite example, I know that my wife is naturally disciplined about eating chocolate. She can have a little bit, then leave it alone. I, on the other hand, will not stop until all that remains is a pile of empty wrappers and a stomach ache! So, as an act of discipline, I bought my wife a safe to store her chocolate in. I’ve lost 22lbs!

Likewise in my work, I strive for quality and clarity of expression. I’m not some superhuman 10X engineer (if such a

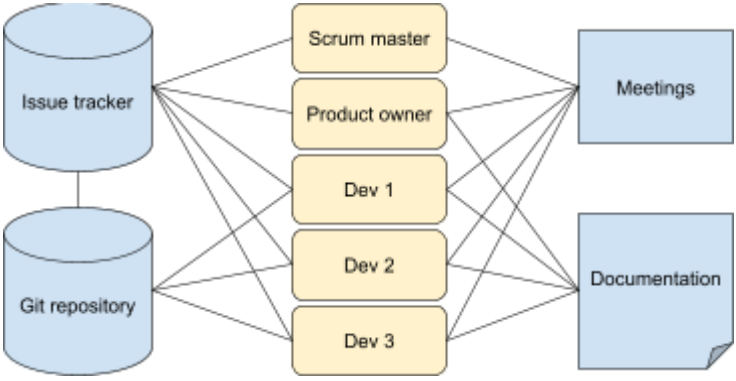
thing exists), but I have learned through bitter experience to set safeguards in place against my own potential for incompetence! The techniques in this book have vastly improved the quality and clarity of my work.

The value of Git discipline

So let's get specific about Git. Why is it important to be disciplined with our use of our source control tool?

Git is part of how we communicate with our colleagues

Our source control system is a key point of integration between a team. This might look something like the following:



We all touch this repository (or repositories) daily (directly or indirectly), so we should consider this interaction to be of equal importance to meetings, tickets, documentation and face-to-face conversations.

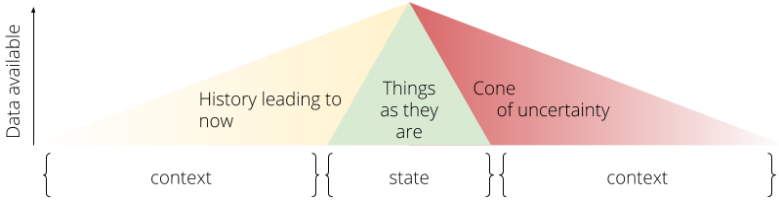
Git is not just "somewhere to save code". It is also the history of our product and our central point of collaboration. Each commit carries with it a small piece of documentation in the commit message, hence as we share this codebase we communicate with one another via commits and pull requests.

Without wishing to be too graphic, leaving the shared repository in a mess is like leaving a mess in the bathroom - it makes it harder for others to do what they need to do in there! Inconsiderate AND impractical!

Conversely, having the discipline to use Git with clarity and expressiveness is like oiling the moving parts on a bicycle - it helps to keep everything running smoothly. When we are intentional about clarity, rigorous in documentation, and explicit in our explanations, our team cohesion benefits greatly. This should give the benefit of lower attrition. As we all know from experience, replacing a team member is expensive and time-consuming, and often rather sad. In later chapters, we will explore ways to communicate efficiently with our usage of Git.

Disciplined Git usage contextualises our work

Everything that we understand to any meaningful degree, we understand within the context of its history and its future:



Git holds both the state and the history of our product - all of our work exists within the context of what has gone before it. If we do not have a clear history, a whole vector of information is just being thrown away!

If I can't look at `git log` for a file and get a coherent picture of how it has changed over time, I have only the current state of the code with which to construct a mental model of the problem I am working on. Likewise, when I come to debugging a problem, one vector I have for doing so is Git history (via a tool like `git bisect`). If commits leave the system in a broken state, I will get false negatives, devaluing the bisect tool.

Conversely, solid Git discipline gives us the history of WHAT changed and WHY it was changed, thus providing a treasure trove of information for understanding, maintaining and enhancing a product.

> For more thoughts on context, see https://gavd.co.uk/2019/07/importance-of-context/

So, Git discipline is about clear communication and contextualisation of what we are doing. It's the opposite of "chucking code over the wall". It's about stepping up, taking ownership of the codebase and being professional in our dealings with our colleagues - and our future selves!

In later chapters, we will explore what this discipline looks like in practise and the techniques we can use to be more disciplined.

Atomic commits

Throwing about the term “atomic commits” is off-putting to some people (in the same way as dismissively saying “*oh, well, Git is easy once you realise it’s just a Directed Acyclic Graph, also I have 9 PhDs and my Dad drives a tank*” can come off as obnoxious). Us mere mortals can define an atomic commit as:

- Does one thing and one thing only
- Can be applied by itself without breaking the system (e.g. git cherry-pick)
- Can be backed out without breaking the system (e.g. git revert)
- Is a complete operation that does not leave the system in a broken state

Benefits of atomic commits

Consider it this way; you don’t want a commit that solves BOTH problem A and problem B, although that may appear tempting. It’s better to have separate commits for A and B, each of which does **one thing and one thing only.** That has several advantages.

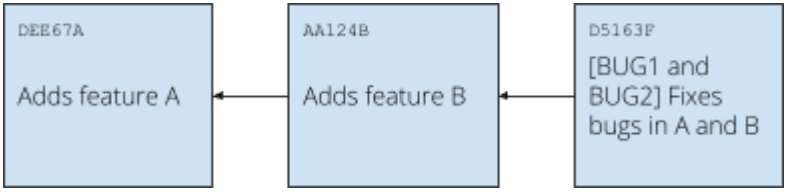
Atomic commits benefit 1: single purpose

Firstly, an atomic commit is clean and clear - each commit can be understood in isolation and its commit message (documentation/release notes) are specific to it. It’s a nice, clean unit.

Atomic commits benefit 2: can be applied or reverted cleanly in one operation

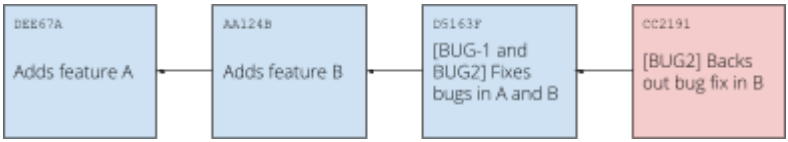
Secondly, each commit **can be applied OR backed out without breaking the system**. With atomic commits, you can use `git revert` to neatly back out a single atomic commit if one of your two fixes wasn't quite as clever as you originally thought (which happens to me on a near-daily basis). You don't have to pull out a load of perfectly fine code for A if it's only B that you want to back out, and you have clean traceability through Git of what has been done.

Let's go through an example. Let's say that we had a project with two features, A and B and two bugs with issue tracker references BUG-1 and BUG-2 were raised against A and B respectively. You fix BUG1 and BUG2 in a single commit:



> Sidebar: commits in Git "point" to their parent commit. So, in this diagram, the arrows are pointing "back in time". Also, the funny strings of letters and numbers are SHA-1 hashes of the commit, which uniquely identify Git commits.

This sounds good, until you realise that the fix for B in commit with the SHA-1 D5163F has created more problems in production than it has solved. You urgently need to back it out, but you can't just revert a single commit, so you end up with:



The commit in red is dangerous because:

- It partially unpicks a previous commit (it's not a clean revert)
- You have had to do it manually and may easily miss things
- It may affect the fix for BUG1 in unexpected ways

If you had done this atomically, you could simply revert the specific fix:

This is preferable because:

- You have traceability over what has happened and why
- You are able to revert the commit as a single atomic unit
- BUG1 fix will not be affected

Atomic commits benefit 3: never leaves the system in a broken state

An atomic commit should never leave the system in a broken state - it should be complete enough that it doesn't require

another commit to make it “whole”. This is NOT to encourage “big bang” commits - it’s similar to the idea of a minimum viable product and iteration common to Agile methodologies - you are looking to deliver a commit that makes one self contained change. Subsequent commits can iterate upon that change, add and remove features and fix bugs, but each is a fairly small step.

This book does not prescribe a particular Git workflow, and atomic commits are useful for all workflows. For example, with atomic commits, you should easily be able to cherry pick commits into release branches if you use them.

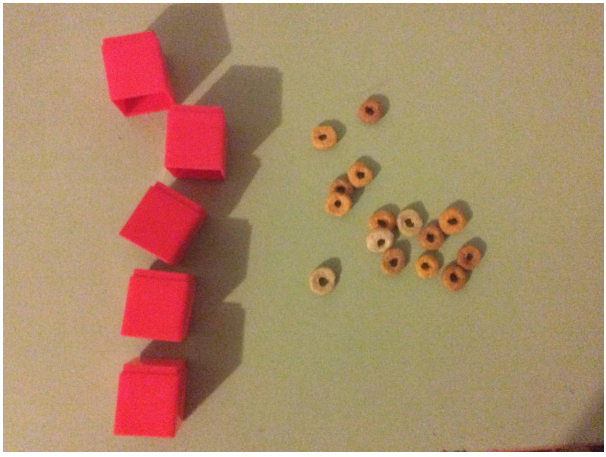
Hopefully now you have an idea of what an atomic commit is, but just in case, let’s describe a counterexample in the next subsection.

The dreaded non-atomic commit

A non-atomic commit may leave the system in a broken state (as in, it needs additional commits for tests to run or to be able to bootstrap). It may be applicable forwards to a branch, but if backed out, it creates havoc. It jumbles together partial fixes for 5 separate bugs. It’s probably got an overly terse commit message reading “stuff” or “WIP” or “FIX”.

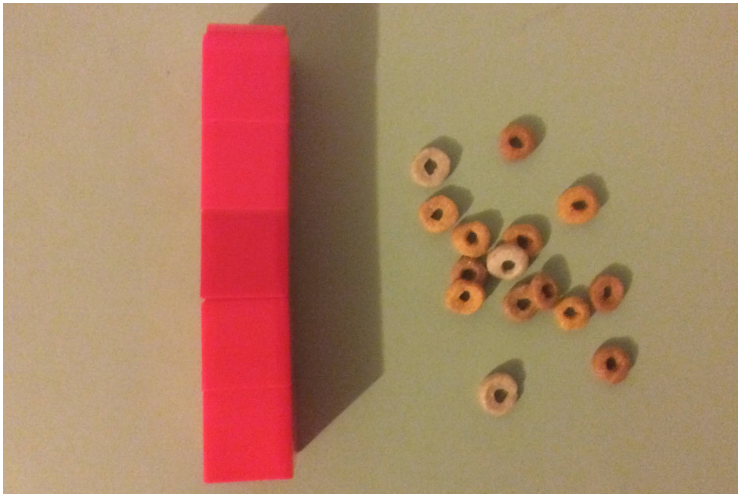
This is exactly the kind of commit this book is intended to help you to avoid creating!

Consider these this photo:



On the left, building blocks represent atomic commits. On the right, some breakfast cereal represents non-atomic commits.

The atomic commits can easily be sequenced, where non-atomic commits remain as chaos - good luck building a coherent picture with them!



Crafting an atomic commit

Sometimes, my working copy will have a few unrelated changes in it. Thankfully, Git allows us to selectively stage parts of our working copy (`git add -p`) so we don't have to commit everything we are working on. When crafting a commit I find it easiest to think:

- Does this commit solve one and only one problem? If not, it's not atomic!
- Can this commit be applied in one go? If not, it's not atomic!
- Can this commit be rolled back cleanly? If not, it's not atomic!
- Is the system in a broken state after committing this? C'mon, sing the chorus: if not, it's not atomic!!

Don't get too hung up!

If you feel like you've made a mistake with a commit, Git supplies you with plenty of ways of fixing it. Even with that in mind, though, don't overthink it; sometimes it's not obvious what belongs as its own commit.

Furthermore, there are plenty of strongly opinionated people, me included, who will tell you what to do - take us with a pinch of salt! Listen, by all means, and think about it, but ultimately make up your own mind.

"Test everything; hold fast what is good." - Paul of Tarsus.

Further reading on atomic commits

https://www.codewithjason.com/atomic-commits-testing/

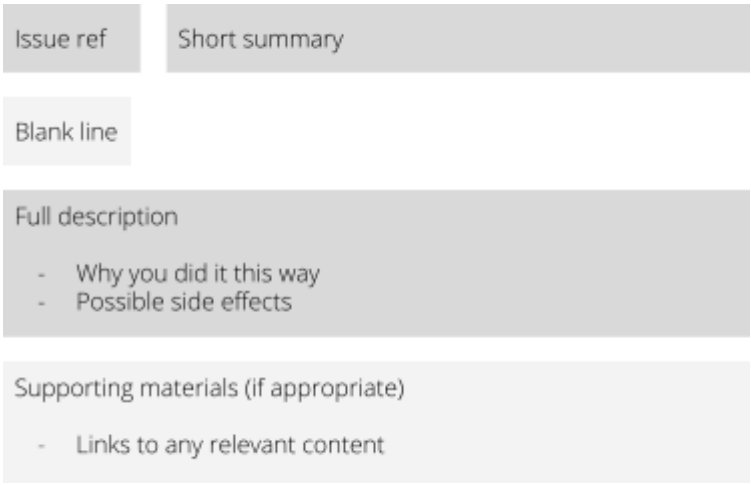
A disciplined commit message

Each Git commit has a delta (the patch that the commit applies) and a message. It may be helpful to consider this physically - imagine the delta (changes to apply) was on a USB drive. With this USB drive, you could include a sheet of paper describing what the change would do, if applied. Then, your team can read the piece of paper to help decide whether they want to apply the delta.

The message can be thought of as the “release notes” of this piece of work. You are saying what’s changed, and why, and putting it in context of the system as a whole. Again, this speaks to the discipline of what we do as professionals.

Anatomy of a good commit message

The “shape” of a disciplined commit is something like this:



Issue ref

Most work in the software industry is done in response to a ticket (or ticket, story, or whatever you want to call it). This ticket will have a unique identifier, and that's what you can put in here. If the work you're doing does not have a ticket associated with it, consider whether it's worth actually creating one, for future traceability. The ticket is a key **point of intersection with the business people** on your team. This is their visibility. Most issue trackers can be configured to automatically detect the issue ref in the commit message. This then gives you a two way lookup between the code documentation (Git history) and the business requirements (issue tracking).

Short summary

This is like an article headline, so it should ideally be less than 70 characters so that you can understand it clearly in

something like `git log --pretty=oneline --abbrev-commit`.

The tense in which I word Git commit messages is a bit unusual. I would write “Fixes overflow bug in Foo module”, rather than “Fix overflow bug in Foo module”. I do this because the message, in my mind, is a description of the patch. I think of it like a Post-It note attached to a floppy disk! (yes, I am that old)

You don’t have to use this tense; most people write in what I think is called the imperative tense (I have a British A-level in English Language + Literature but bizarrely I can’t recall ever having been taught the technicalities of grammar!), such as “Fix overflow bug in Foo”. It doesn’t matter all that much, write in a way that helps you to feel comfortable documenting your changes.

Blank line

This serves to visually separate the “headline” from the “article”.

Full description

A commit is effectively a patch (you can see this when you use the `git diff` command). The commit message is the documentation of the patch and should tell you exactly what the patch does and, most critically, **why**. The “what” can be inferred from the delta, but the delta cannot tell you the intention of the programmer in making this change.

Examples may be read something like:

“I changed this setting because the PHP 15 API will no longer support this function, and some of our clients are due to upgrade in January”.

“I removed this code because the branch could never logically execute, if you look at the if statement you’ll see it’s ALWAYS falsy”

“I read these 3 articles (links provided below) that all described why what we did here was insecure. I have replaced it with a known-secure algorithm and if you run the tool at tools/updatesec.sh it will fix all the hashes in the database.

Supporting materials (optional)

In my University days, the one thing that was drilled into us was to thoroughly reference our sources. I see no reason to discontinue this practise in my professional life - I do not wish to present myself as an authority in my own right with my pull requests!

So, sometimes I solve a problem based on articles or books I have read. If that’s the case, I will link to these materials in this section. I might say something like *“this solution was offered on StackOverflow [link] for someone with a very similar problem. I researched it in the language documentation [link] and it seems legitimate. It solves our problem as per my test script”*.

What NOT to include?

Obviously no-one wants to read War and Peace for every commit!

Here are a few guidelines on what to leave out:

Don't duplicate information that's already elsewhere, provided it's easy to find!

For example, your issue tracker linkage should already be in the headline (e.g. #123), and perhaps also mentioned in the supporting materials section.

Git is a historical axis of information.

Does the information you're writing in the commit belong in the project's documentation? Git is historical information; documentation is about the present.

If you're documenting functionality in your commit message, take a moment to consider if you should be updating the project's documentation instead. Think carefully about which information goes where.

For example, if you're writing some instructions to run a tool (known as a runbook), you'd be better off adding a README, or updating a wiki somewhere, and providing a link in the commit message.

Any personal gripes

Sometimes I have been known to vent in commit messages. I try not to do this any more, it's unprofessional to allow resentment to be expressed in this way.

Examples of this are "replaces dumb implementation with something smarter". This is clearly unacceptable. We engineers are not unfeeling machines (despite how we are sometimes portrayed!) but that does not give us the excuse to be anything less than professional.

If you really feel that you need to vent about something, a nigh-immutable vector of historical record of your project is not the place to do it!

Further reading

There are a lot of great posts out there about making good commits, here are a couple:

- https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html
- https://8thlight.com/blog/makis-otman/2015/07/08/git-disciplined.html

A disciplined curated history

In the previous section, we established what a good commit looks like. However, I seldom get it right the first time!

Thankfully, Git is a distributed version control system (DVCS). This means that we have our local repository, and if I haven't pushed my commits yet, then I can change them to my heart's content!

Use of branches

Even if I HAVE pushed code, there is often plenty we can do, provided we are using a branching workflow. More on that in the next chapter. To keep things simple, this chapter is just focused on getting it tidy before you push.

Generally, I create a branch for every change I make and submit it via a pull request (more on those in the next chapter).

This gives me a degree of isolation, meaning I can work from a stable base, at the cost of taking on the burden of having to keep my branch up to date from the source branch.

Caveat - other approaches exist!

Some organisations have solid strategies based around working directly in a shared branch such as `main` or `development`, usually because they value super rapid integration. Do whatever suits your organisation!

Whatever you do, long running branches are a really bad idea.

Squashing commits

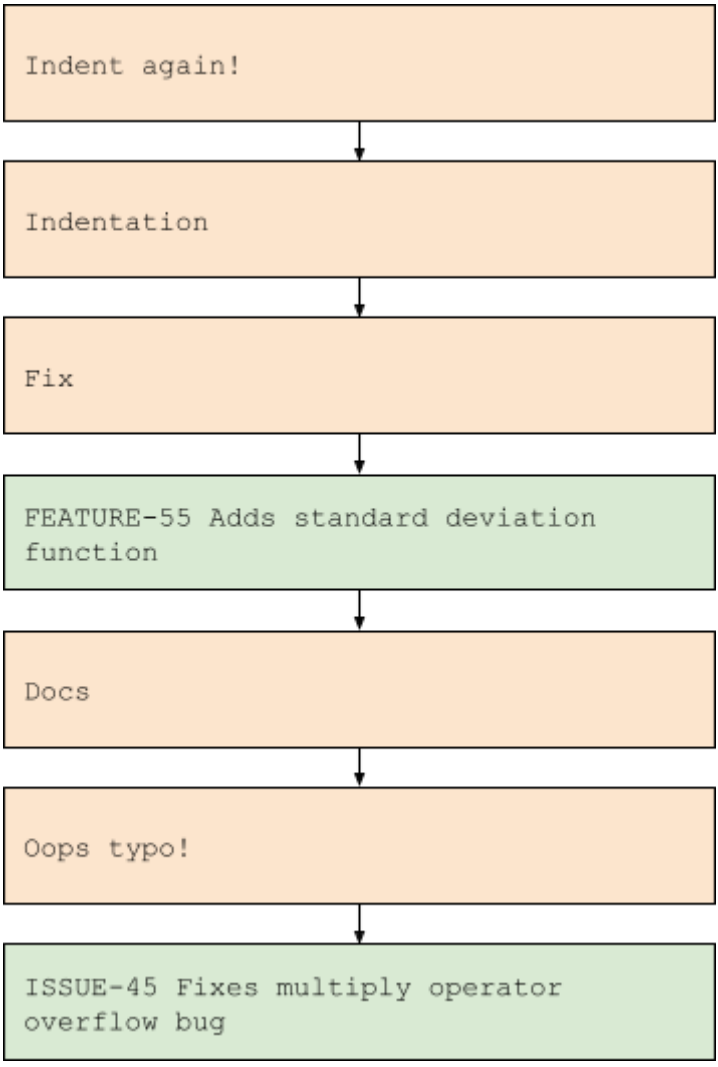
Which of the following commit logs is easier to follow?

Log A

`Indent again!`
`Indentation`
`Fix`
`FEATURE-55 Adds standard deviation function`
`Docs`
`Oops typo`
`ISSUE-45 Fixes multiply operator overflow bug`

Log B
`FEATURE-55 Adds standard deviation function` `ISSUE-45 Fixes multiply operator overflow bug`

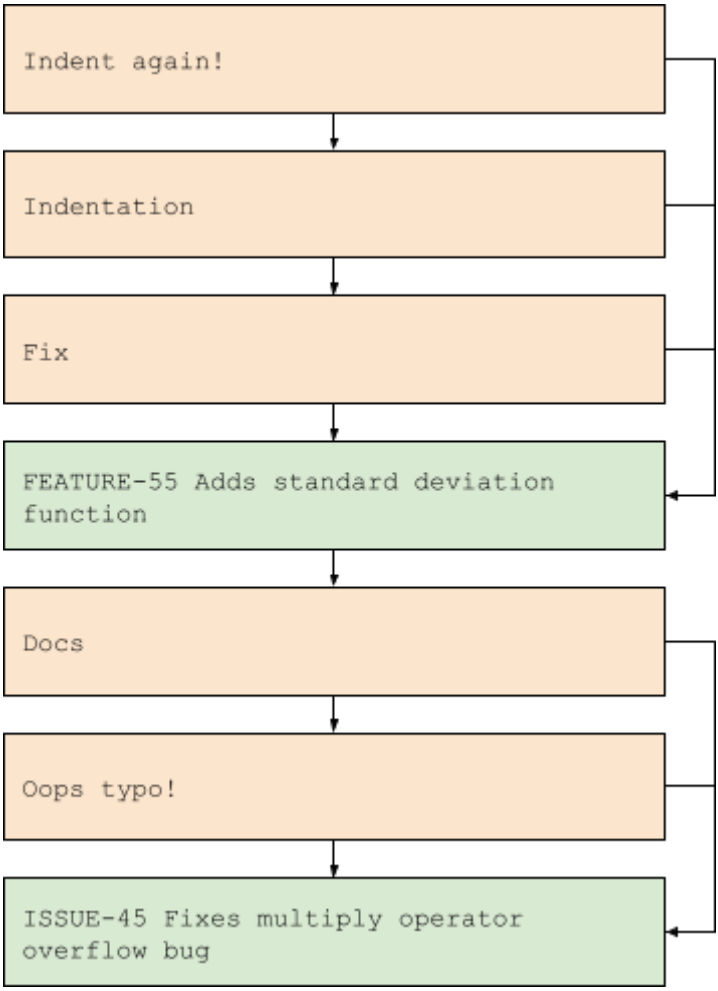
Here is a sample commit history:



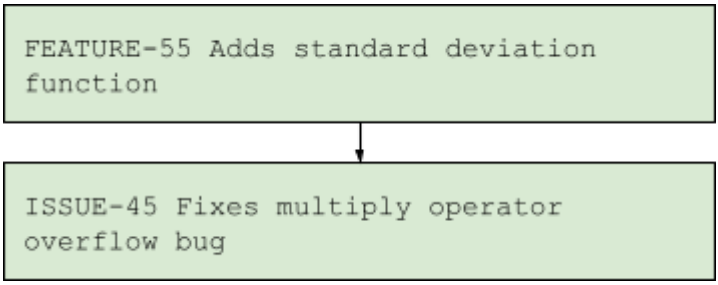
In Git, commits “point to” a previous commit (for a full understanding of this, read up on graph theory), so we can

construct a linear history. So in this diagram, the newest commit is “Indent again!”

Looking at this history, we can see two “high value” commit messages (green boxes), with a bunch of “noise” commit messages (shown in amber). We can easily squash these into two **high value** commits:



Then we have a short, contained, high value, clearly documented history:



Word of caution: don't go too crazy with the squashing. Refer to our rules on atomicity in the last chapter.

But that's rewriting history! You MONSTER!

Think of it more as *curating* history. A historian takes the facts and weaves them into a coherent narrative, and what you're looking to do is similar; when you push commits, you have a golden opportunity to present an idealised version of history that is clear to follow.

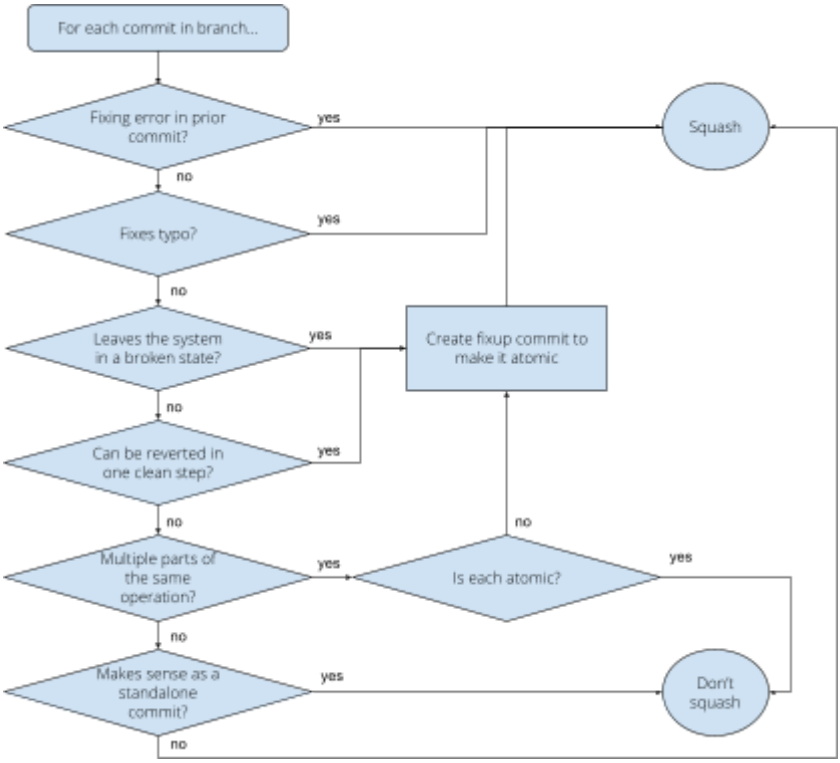
After all, if you are investigating changes I made, do you really care about a few typo fixes? Is that really relevant to the investigation, or will it just add noise?

Furthermore, I only rebase in my own branches. Once something is merged to a mainline branch, it's pretty much etched in stone!

To squash or not to squash?

Sometimes it's obvious that a commit should be squashed. For example, let's say you make a commit and immediately

you `git show` and spot a typo. There is no value whatsoever in a separate commit being in history showing you fixing your typo - it's just noise! Here's a flowchart showing how I decide whether to squash:



This flowchart is far from exhaustive, but hopefully it'll give you some rough guidelines.

Sometimes it's harder to figure out if a squash is called for. Different people have different opinions on this, but this is my approach. Just treat this as training wheels - once you've

worked this way for a while you'll get a feel for what you think belongs in its own commit and what can be squashed. Don't overthink it - this whole book is about getting into good habits, not about turning you into a neurotic overthinker!

I am not advocating for squashing every single commit. If a commit is neatly atomic and we can back it out trivially by reverting it, then by all means keep it as-is. If, however, a commit is NOT atomic, or there are multiple commits to solve a single problem, you cannot use `git revert` and you have a far more difficult job of backing out a single change.

Commit often!

It's a good idea to commit often, so that (a) you have a local log of what you've been working on and (b) if something goes screwy, you haven't lost work. This may seem to be in tension with crafting good commits, but the power of Git is that you can go back and make a coherent history when you're ready to push. When I am doing experimental work firing "tracer bullets" (hit and hope programming!), I often make a number of dozens of "safety" commits simply marked "WIP" for "work in progress" in my working copy. I would not DREAM of integrating such commits into the main branch of my project, and I invariably squash/fixup/remove/reword these commits before pushing!

A disciplined pull request

Your organisation may not use pull requests, but perhaps you'll read this section and consider using them. Either way, it's a good idea to think about the point of integration of your work with everybody else's efforts. It also helps team cohesion when everyone knows their work has to be peer-reviewed.

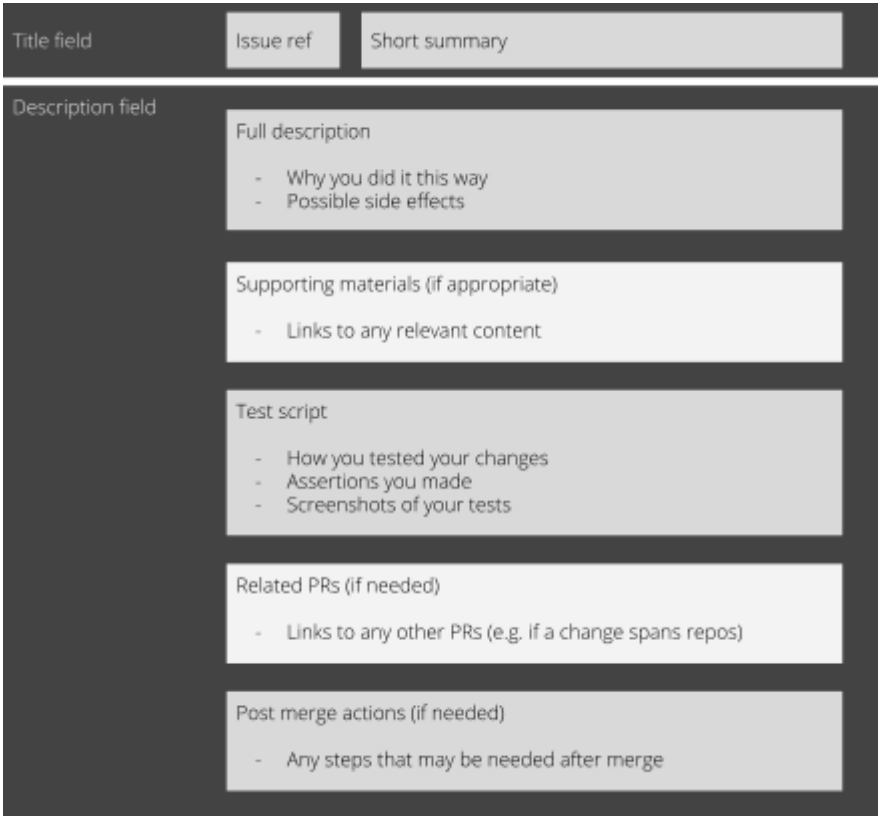
How to conceptualise a pull request

Pull requests (PRs, also known as Merge Requests or MRs) are where your commits are "staged" for integration with the mainline product. You could think of it as a "job interview" for your work. You can think of your pull request comments as the resume/CV for your efforts.

You are saying "I believe this is something that should be integrated into the product", and as such, you must try your best to express yourself with clarity.

Thankfully, if you've followed the guidelines in this book so far, you're already nearly there!

Anatomy of a good pull request



Issue ref

Just like the “Anatomy of a good commit message”, a PR should have an issue reference as part of the title field. This is not a duplication of information; many Git web front-ends such as Github, Bitbucket and Gitlab will automatically link issue references in the PR title to your issue tracker. This is an enormous time saver for your colleagues reviewing your work, and supplies vital context for your changes.

Short summary (single line)

Again, this is very similar to the recommendations for individual commits. A PR may have multiple commits with their own short summaries, but your PR short summary must summarise the sum of all the commits. You could consider the PR to be a ladder composed of rungs (commits). When writing your short PR summary you are writing the overall description.

Full description

A PR should detail what has changed and why. You should demonstrate that you've considered potential side effects and maybe alternative approaches. Don't assume that your reader knows everything you know! You never just dump code and run, you care for your work from the first reading of the ticket to the deployment to production through to decommissioning, and this is a critical step - your code is doing an exam to graduate high school!

Supporting materials (optional)

Just like adding supporting materials in commit messages, we should have them here too for the benefit of our reviewers.

Test script

Every PR should provide a test script section which tells your colleagues how you tested your change. This means that (a) you have demonstrated that you have done due diligence in testing your changes and (b) anyone coming along should be able to test your changes before integrating them.

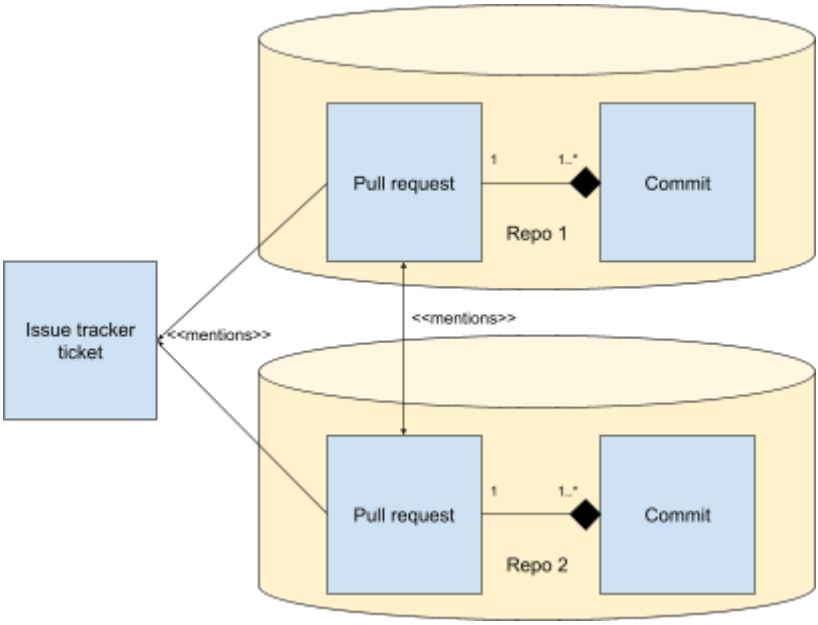
A test script should be a set of steps with assertions, for example:

- Run the project using the supplied container as per the README [link]
- Click on the “login” link
- **Assert that** the new logo is shown

There is rarely such a thing as a truly trivial change, so even if the test script is just “review the diff and **assert that** the changes I made to the README are accurate”, that is sufficient.

Related PRs (optional)

We may have a section “Related PRs” for when a change may span multiple repositories. In this diagram, we see an example of this:



We have raised two pull requests. Each refers to the other, which is a circular dependency of sorts, so this requires you to create one first and then the other, and then edit the first one to mention the other.

If there's a specific order that the pull requests would need to be merged in, make that very clear in your pull request documentation.

Post merge actions (optional)

Sometimes, if a PR is merged, other changes must take place. For example, if a PR contains a script to modify a database, the post merge actions could say "run tool foobar on the command line on the staging database server".

There may also be actions to update public facing documentation, should your PR add or modify an API endpoint,

Having a post merge actions section helps to contextualise your work within the overall direction of development by considering the broader scope.

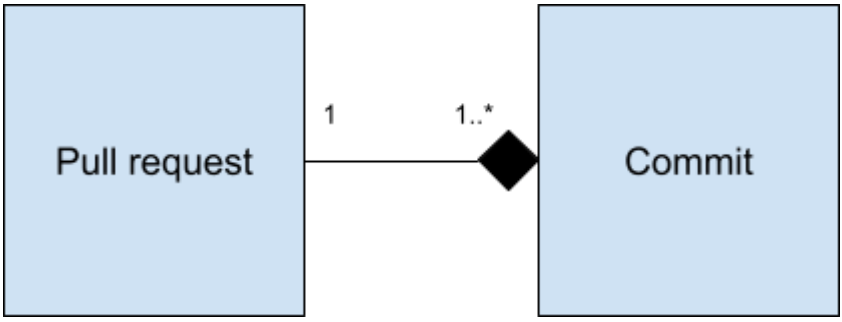
When to raise a pull request? Integrate often!

Many of us in the software industry use agile processes these days. If we are to be agile, we want our feedback loops to be as short as possible. Therefore, I prefer to integrate as early as possible so that:

1. The barrier to integrating future work is lower.
2. My team can see clearly what I am working on, and identify problems with my approach. This allows us to “shift left” and tackle problems as early as possible, before massive investment has been made.
3. I am not asking my colleagues to review 900GB of code changes in one go.

In the same way that commits should be atomic in the microscale, so should pull requests in the macro scale.

A pull request may contain many commits:



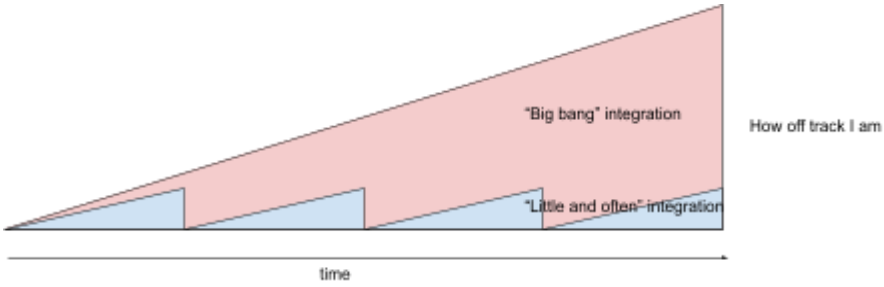
If this is the case, these commits should be atomic steps to solving the same overall issue.

So, I raise a pull request as soon as I have something that is either immediately useful in the product, or something that I can at least integrate and hide behind a feature toggle.

Long running unintegrated branches are a really bad idea for several reasons:

1. Harder to integrate as time goes on (due to drift).
2. Lower visibility and transparency with your team.
3. The longer I am working on my branch without showing my colleagues, the less likely I am to stay on track

The diagram below demonstrates how frequent integration reduces the risk of getting off-track:



Don’t expect people to do the work of machines

This is edging towards being out of scope of this book, but make sure that the changes in your PR pass any quality standards that your organisation may have. The best place to do this is firstly on your development machine, and secondly the act of raising a PR should kick off a build that runs a suite of tests, static analysis and quality testing tools. Do not expect humans to waste their time checking that you’ve used curly braces correctly according to the project standards! Leave that work to the machines and let the squishy brains do the abstract thinking!

Beware the phenomenon known as “bikeshedding”:

> "The term comes from an illustrative anecdote of a committee discussing a plan to build a nuclear power plant. In their meeting they spent the majority of their time arguing over the color to paint the bikeshed in the back, because that was the part of the plan that everybody could understand."
> - https://phinze.blog/2014/05/24/useful-tech-terms-part-1.html

Pull requests are at risk of bikeshedding whereby nobody tests your actual code but everybody bickers over whether you've indented your code correctly. It's best to obviate this risk - apply a standard and hand it off to the machines!

Reviewing pull requests

As well as raising pull requests, we also review them. Here are some questions I ask:

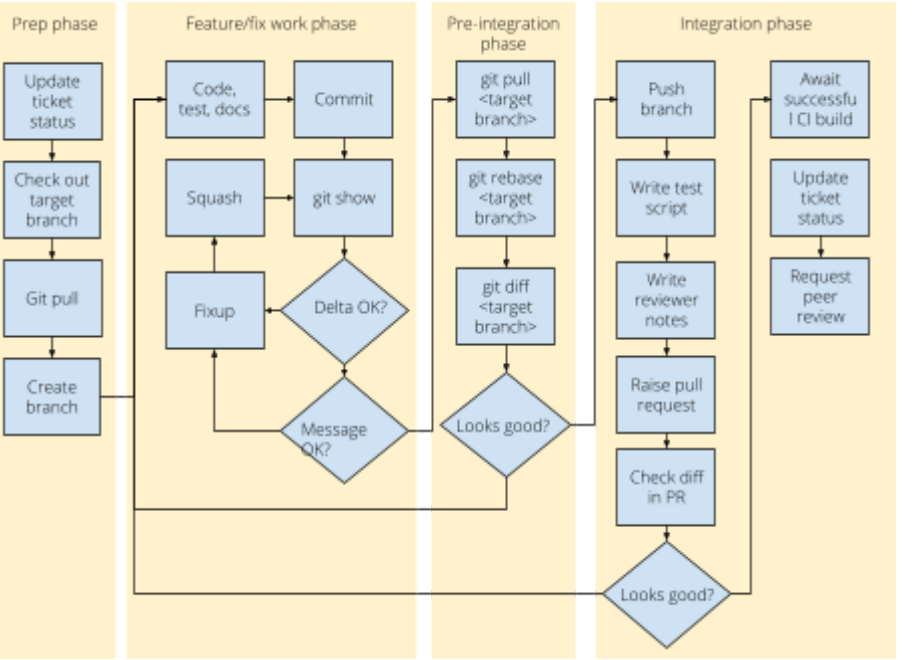
1. Is it accurately described?
2. Is this going to benefit the project overall?
    a. Is it actually pushing the product in the direction we're going in?
3. Can I test it?
    a. Has a test script been supplied, or is there some clear way to verify this change?
4. Is it done in a sensible way?
    a. Has the developer re-implemented something that already exists? If so, you've got a great teachable moment!
5. Is the history clean?
6. Do all commits meet the criteria in this book? (atomic, clearly documented, etc..)

When you review a pull request, you are effectively signing your name to it and saying "this is good enough for our product", so you should test it thoroughly. This isn't to say things have to be PERFECT (being overly pedantic will upset your team!) but we succeed as a team or we fall behind as a team, so take ownership of everything you review, don't just blindly integrate.

Workflow: putting it all together

There are many Git workflows out there, from simple single-branch linear workflow, through Trunk Based Development, to complex models like Gitflow. They're pretty situational - you pick a workflow that works for your team and this book is written in such a way as it should benefit ALL Git workflows.

Here is an approximation of the way I work on features or fixes day-to-day:



You may notice that I do a LOT of diff reviewing - in fact, I do it in 3 separate phases! The number of times I have spotted problems in this manner really is unbelievable. I've closed dozens of pull requests within moments of opening them because I've spotted something that I didn't spot in my local development.

In case you haven't come across the term before, the "delta" a fancy computer sciency word for the changes made in a commit - you may hear people refer to it as "the diff".

Because, as we've mentioned, everyone's workflows are so different, I'll leave it at that but suffice to say, I've applied everything in this book and it's all automatic now.

Applying discipline to other areas

“He that is faithful in that which is least is faithful also in much: and he that is unjust in the least is unjust also in much.” - Jesus Christ

I really hope that this book has given you some ideas on how to improve your discipline in one area of your professional life. No matter how you approach Git, whether you use the patterns I use or not, I hope that having read this book will make you think about your use of this tool, and consider how you can do it better.

Commit discipline is important but it’s just one small part of your job as a software professional. When you first start this level of discipline, the cognitive load will be high, but over time, it will become second nature. Then, you will start to actually reduce the cognitive load your version control requires because it will be semi-automatic, freeing up more brain power for other activities. This is what I meant at the start of the book with the “Discipline == Freedom” quote.

Furthermore, a disciplined engineer is extremely attractive to employers - people are more likely to enjoy working with disciplined engineers.

If we step back from the nitty-gritty of Git, this book is really about being intentional about communication. Yes, Git’s an important part of our job, but hopefully you can see how this kind of disciplined communication can be applied everywhere in our jobs:

- Writing specifications
- Giving presentations
- Communicating well - good emails and Slack posts
    - https://iridakos.com/how-to/2019/06/26/composing-better-emails.html

I'll leave you with three things:

- Be considerate - always assume that other people are reading your work, even if it's "future you". Consider what they need to know, and how you can help them become better professionals.
- Be clear - make sure what you say is easy to follow. Don't use 100 words when 10 is enough. Don't replicate when you can link.
- Be disciplined - what is strange at first will rapidly become second nature through discipline. Don't be too hard on yourself, it will take a while, but you'll get there, and you'll be a better professional for it.

Troubleshooting

This takes too long, I've got work to do!

Firstly, it doesn't take long at all once you've gotten into the habit. Taking a minute or two to craft a decent commit message or to rebase a branch cleanly is no great hardship.

Secondly - THIS IS YOUR WORK! Communicating to members of your team, both present and future, and even your future self, is absolutely as important as the current state of the codebase. Your commits are one of the most critical interfaces between you and your team, so surely they should be as clear as possible?

I encourage you to think more broadly about what your job is, about what it means to be a professional.

I don't understand what is a rebase

A few years ago, I wrote this conceptual guide to understanding Git rebase that should help you http://radify.io/blog/comic-continuity-and-git-rebase/

My code is well commented, why would I need to write commit messages?

Whilst there is undoubtedly crossover, commit messages serve a subtly different purpose to code comments as commits are intrinsically tied to a point in time and a specific task. As such, code comments are no substitute for

expressing your intent with your commit message, where you say WHY you made a certain change.

I can't get my team to "buy in" to Git discipline

Firstly, listen to their concerns. REALLY listen. Do they have valid points? Think about how you're approaching this. Are you shoving it down their throats? Very few of us like to be ordered about.

Lead by example. Be disciplined, demonstrate value.

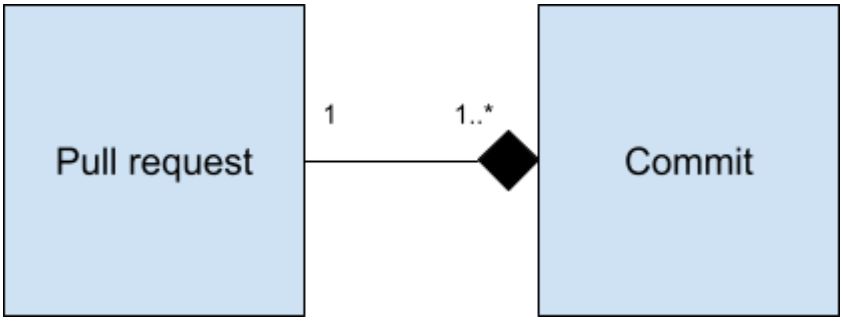
If you can get one person on your team to buy in to disciplined Git usage, the power of shame will eventually raise the game of the rest of your team ;-).

Also, this book is free, so give them a copy!

The issue number is in the branch name or PR description, why do I need it in the commit as well?

The commits are a permanent record. You can think of them as an audit trail.

Feature branches and pull requests, however, are not permanent - once they're merged, they're gone! Remember also that a pull request may have more than 1 commit:



So, if you only use the issue number in the branch name or PR description, once the branch is merged, you've essentially thrown away your audit trail. You don't have a full log of every change relating to a ticket. `git blame` will not show you which line relate to which ticket.

So, yes, add the ticket number to the branch name AND PR description, but also be sure to include it in each commit message!

I'm struggling with the technical side of Git

https://git-scm.com/book/en/v2 is a first rate guide to Git. Keep at it! Git has a learning curve like a brick wall at first but sooner or later it will "click" for you.

Why do I need to write a test script in my PR? Shouldn't that be automated?

Should code have automated tests? Yes.

Should your colleagues know how to run your code? Yes.

Is it your responsibility to make sure of both? Yes! As IT professionals, we take ownership of the whole engineering lifecycle. We succeed as a team, or we fall behind as a team, so we take radical responsibility for every aspect of what we do. This is the opposite of culture that leans into constantly looking to blame others and shirks personal responsibility.

Add automated tests where you can, but also remember that your colleagues will likely also want to run your code on their development environments, or on a staging system, or similar, before integrating it into the product. It is “cheaper” to catch a problem at this stage rather than back it out later.

Even if your colleagues never run your test script through, at the very least it shows your working and allows them to make an informed decision about whether to merge your work.

History

Version	Date
Draft 1	August 2019
Draft 2	14 September 2019
Draft 3 - incorporating feedback from Toby Maxwell-Lyte and Benjamin Lavery-Griffiths	18 October 2019
Draft 4 - minor updates	20 August 2020
Draft 5 - reformatting, incorporating feedback from Rhodri Pugh	17 September 2021
Draft 6 - incorporating feedback from Mark McKee	18 September 2021