Git and PR standards

General notes

Below is a collection of rules, some mandatory and some optional. It is an (incomplete) attempt to codify my mental model for how to work with git and pull requests. It is informed from my years of working on long lived codebases at product companies as well as my time at Catch. It therefore represents the biases of such a background. While many of the principles here are broadly applicable to many use cases they are not transferable to every situation and would make less sense in an agency producing brochureware websites.

The most important advice I can give is use your judgement. You are paid to think, not to be a robot. For example you shouldn't refuse to deploy a critical production fix at 9pm on a Friday because you can't find anyone to review your PR. (By the same token don't just merge your fix without trying to find a reviewer)

You will notice I use the word context often. It's because context is key. A mutual understanding between the developer and a reviewer of what is being done and why is critical to enable good quality code reviews and rapid progress of PRs through the review process. Similarly future maintainers will find this context critical as they may not be able to rely on asking you why you did what you did. We have commits from 8+ years ago and more developers who worked on this codebase have left Catch since writing the code than remain.

Git commit standards

Merge commits MUST not change the default commit title provided by git or GitHub
Why? The default title contains useful information and follows a well known format that other developers can easily identify.
Examples:

Correct: `Merge pull request #11075 from CatchoftheDay/pf-proxy-headers`

Correct: `Merge branch 'fix-cart-parcel-club-discount-SM-1127' into testing`

Incorrect: `Resolve merge`

Commit messages MUST use the imperative mood and start with a capital letter. They should take the structure of a full sentence
Why? This is a commonly accepted standard and aligns with the default messages generated by git (e.g. Merge branch 'develop' into master). This enables consistency and reduces unnecessary verbiage allowing easier parsing and greater information density.

Incorrect: `Updates database password parameter` (not imperative)

Correct: `Update database password parameter`

Commit MUST have well structured and have descriptive titles. Every commit should follow the unix philosophy of do one thing and do it well. Commits with words like WIP, changes, fix bug, or add unit tests are unlikely to meet this standard.
Why? Git commit history lives forever. When it is well structured it is an extremely useful artifact to our future selves. Commit messages provide the most relevant information about your state of mind and intent at the time you did a piece of work.
Why? If a reviewer can't make sense of your changes one commit at a time they can't make sense of your PR
Why? Reviewers, especially in the case of large PRs may find it much easier to build context by seeing the *what* and *why* of each commit at a time rather than the entire body of work.
Examples:

Incorrect: `Fix CT-123`

Correct: `Extend search bar to full width - fixes CT-123`

Commits MUST be rebase to remove distracting errors from history. (e.g. no 'change lenght to length' commits).
Why? These add nothing to the reviewabilty of the code, nor do they add anything meaningful to commit history if the mistake never makes it to master or production. A reasonable exception to this is if the PR is large and the rebasing would take an inordinate amount of time.

Incorrect: A commit with the message `Fix typo lenght to length` which corrects a typo in an earlier commit within the same feature branch.

Correct: Interactively rebase the branch to squash the typo fix in to the commit that contained the type

Pull request standards

Below is a collection of rules. This is an incomplete documentation of my mental model for how to work with pull requests. The most important advice I can give is use your judgement. You are paid to think, not to be a robot. Don't refuse to deploy a critical production fix at 9pm on a Friday because you can't find anyone to review it. (But don't just merge it without trying to find a reviewer)

You will notice I use the word context often. It's because context is key. A mutual understanding between the developer and reviewer of what is being done and why is critical to enable good quality code reviews and rapid progress of PRs through the review process.

Structuring Pull Requests

The PR MUST explain *what* changes have been made and *why* they were made.
Why? This is a clear statement of purpose and intent from the author. E.g. *Restrict gift card redemption to registered users* is a much clearer statement of intent than *Fix checkout bugs*. A future contributor can easily tell from the first commit message that if the change excludes social login users from redeeming gift cards it is probably a bug. They have no hope with the second commit message.

PR title MUST start with a capital letter and include the ticket number at the end of the PR title e.g. *Update composer dependencies SM-101.* Much like a commit message a PR title MUST use the imperative mood
Why? Enables JIra to link the PR to the relevant ticket. This is machine readable information that is of lesser relevance to a human

PR SHOULD NOT include work extraneous to the jira ticket.
Why? Extraneous work is of lower priority than ticketed work. Coupling it to a ticketed PR means it can block the substantive changes from being approved and deployed.
Unticketed work has by definition not being assessed or prioritised by a PM or squad lead and may in fact be rejected. Coupling it to work we have decided to do risks slowing that work down.
Why not mandatory? We trust you to use your judgement. Improving the structure of code sometimes makes sense. We trust your judgement to decide when this should be a separate PR and when it can be rolled up.

You MUST mark a pull request as draft if it isn't ready to be reviewed.
Why? Any developer at any time can choose to provide feedback on an open PR. Don't waste their time with unfinished work.

You MUST generate snapshots for frontend tests in a commit separate but immediately after the substantive change that requires the new snapshots.
Why? Snapshots create large diffs that are typically of little to no interest to human reviewers. They should be kept separate from substantive changes to enable reviewers to focus on the important parts of a PR. This speeds up reviews and thus reduces cycle time.

You MUST complete every section of the PR template for a repository in every PR (except deploy PRs in cotd-website and unity)
Why? These templates represent the minimum standards we have agreed upon as developers to give us the context we need to review a PR. If you disagree raise a PR against the template and open a discussion.

You SHOULD include screenshots for work that makes minor cosmetic changes
Why? This adds a lot of context for reviewers and saves them the trouble of checking out your branch and running it locally.

The PR MUST include all relevant context or links to it.
Why? Context is essential for effective PR reviews. It helps explain *why* the changes are being made. The submitter is best placed to collect all relevant information quickly. Centralising it in the PR eases the job of the reviewer, who needs to quickly build context to review effectively. Easing the reviewers job leads to faster reviews which reduces cycle time.

Selecting Reviewers

You SHOULD NOT request reviews on a PR with failed tests or merge conflicts. Your work is incomplete and therefore incapable of being fully reviewed. Respect your fellow developers' time.

You MUST NOT tag any reviewers for draft PRs unless you explain which parts of the work you want feedback on.

You MUST NOT use a scattergun approach to selecting reviewers. If you tag 8 reviewers because they were all suggested by github you are doing it wrong. You should know exactly why you have tagged each reviewer. If you only want a reviewer to comment on a particular part of the PR add a comment describing what you want them to look at and why in the description of the PR.

If your work impacts another squad you MUST tag somebody from that squad for review. You can tag a squad as a reviewer but use this with caution. It sends out a lot of notifications and you can fall victim to a tragedy of the commons where every squad member leaves the review to somebody else. You are much better off engaging directly with people. Consider asking for a reviewer in the squad channel or even better engaging with the squad before you commence work. You will then know which individual to tag for review.

If you have sought assistance on solution design or paired with somebody on your work you MUST tag them as a reviewer. You MUST NOT merge the PR without an approval from them.

You MUST NOT blithely implement PR feedback. Ensure you understand *what* you have been asked to change and *why.* You should be able to explain the change and why it improves the PR to anybody who asks. If you don't you MUST seek further clarification from the reviewer. You MUST NOT merge the PR until this is resolved.

General note on reviews

You are responsible for shepherding your work to production. Delivering code isn't delivering value. Creating a great experience for our customers is. Your work isn't done when you move a card to *In Review*. Our customers only benefit when you ship your work. It is your responsibility to chase up pending reviews and to respond to feedback.

Making your reviewers' lives easier allows them to provide better reviews sooner. This helps us all move faster as a product team. It is also of enormous benefit to future code spelunkers who may be looking at a piece of work many years after it was added to the codebase.

Reviewing Pull Requests

General notes on reviewing

Always assume the positive intent of the PR author. If you think a PR is fundamentally misconceived it's safe to assume the author is considering a context you are unaware of or is missing context you have been made aware of. In these cases dumping a wall of comments on a PR is counter productive. You are much better off pairing with the author to explain your concerns face to face with the opportunity to hear their feedback and learn any additional context they may have. If you don't have time for pairing consider whether you have time to properly review the PR.

Avoid drive by reviews. These are when you roll up to a PR you haven't been tagged on and you have little context on and start firing comments before driving off never to be seen again.

Ensure the commit messages explain what changed and why. If you don't understand *why* a change was made or *what* a piece of code does add a comment. The code, in conjunction with the commit messages should give you this. If it doesn't do you really know what you are committing or are you just engaging in a rubber stamping exercise?

Ask don't tell. *Why have you done X? What happens in scenario Y?* Are much more helpful than *Change this to account for Z*.
Why? They allow reviewer and developer to build a shared understanding of the problem and more complete mental models for the problem space. As a PR author you can avoid many of these questions with well structured commit comments that encapsulate all your thinking and reasoning. These have the added benefit of helping not just reviewers but also future developers who might be called upon to refactor the code or investigate a bug.

START RIGHT on reviewing PRs (as opposed to SHIFT LEFT for testing). Work that is in review is closer to being shippable and therefore more important than any work in progress (except if you are working on a critical issue or your squad has prioritised work poorly). You will deliver more value sooner to customers by reviewing PRs ahead of focusing on completing your own cards. You should start your day, or after returning from a break (e.g. lunch) by completing any code reviews asked of you. *20% of cards 100% complete is better than 100% of cards 20% complete.*

Responding to code reviews

You SHOULD mark any review comments as resolved once you have actioned them
Why? This reduces the cognitive burden for other reviewers and lets existing reviewers know you have accepted their feedback.

You MUST NOT mark any review comments as resolved unless you have agreed a solution/approach with the reviewer OR if the comments are no longer relevant.

Merging Pull Requests

PR MUST have an approving review before merging

PRs SHOULD NOT be merged until all reviewers who requested changes have provided an updated approval review.
Why? It's important that new work in response to review comments meets the satisfaction of the reviewer. Occasionally there is a misunderstanding or lack of shared context. Final review ensures this is not the case.

Why SHOULD and not MUST? Some comments are abundantly clear and trivial in the context of the PR. If this is your judgement and you are satisfied you have addressed the feedback you can merge the PR without requesting another review. Good examples here would be fixing spelling mistakes in a comment or variable name, changing the visibility of a function or property.

PR MUST NOT have any unresolved discussions or unaddressed comments. **Nb,** this does not require you to accept all feedback on a PR, but you must engage with the reviewer until you both agree a path forward. In the event you can't agree a path forward escalate to a lead/principal developer who will have a casting vote. It is expected that before you escalate to a lead all arguments for and against an approach have been explained in the PR. A lead should be able to make a decision 'on the papers' without entering in to further discussion.

Further reading

- https://github.blog/2015-01-21-how-to-write-the-perfect-pull-request/
- https://betterprogramming.pub/how-to-make-a-perfect-pull-request-3578fb4c112
- https://www.atlassian.com/blog/git/written-unwritten-guide-pull-requests
- https://gist.github.com/mikepea/863f63d6e37281e329f8
- https://github.com/erlang/otp/wiki/writing-good-commit-messages
- https://charity.wtf/2021/08/27/software-deploys-and-cognitive-biases/