Development Lifecycle

> 💡 This page documents one possible development process engineers can follow. It's not set in stone, but these are a set of practices that help high-performing teams work well together. If you have questions or suggestions for improvement, drop a message in #guild-engineering.

1. Create a branch off of `main`

Name the branch something descriptive, with your name prepended.

```
[repo] (main)$  git checkout -b 'my-name/my-topic-branch'
Switched to new branch 'my-name/my-topic-branch'
```

> 💡 Depending on which project management tool your team uses, there might be an integration to do this for you! Ask your teammates for tips on your team's process.

2. Writing code

When working in a new codebase, it's a good idea to read through the guidelines so you're aware of the conventions we follow. You should also look over the application-specific documentation, including the project README.

3. Commit your work

🖋️ Crafting git commits

You might use commit messages like `wip` as you work, but make sure you take the time to re-write those commit messages and rebase your work into atomic commits that reflect the logical flow of the changes. Read the guidelines for 🖋️ Crafting git commits to make sure you conform to our style guide before pushing up your branch

⛔Avoid long-running branches

Long-running branches often become unwieldy and difficult to manage with merge conflicts and lots of changes that are hard to review and even harder to revert if there's a problem.

> ⛳ If your change will be behind a feature flag, try to add the feature flag as early as possible. This will make it easier than trying to retrospectively introduce a feature flag later.

If you have a large feature to implement, consider breaking it up into smaller changes and open separate PRs for each. Try if you can to get the code in behind a feature flag and then incrementally iterate on it.

If it's a large piece of work that can't be merged incrementally - such as a major package upgrade that requires a lot of changes - create an empty base branch to merge smaller PRs into so they can be reviewed individually. Make sure the base branch is also reviewed before merging all the changes into `main`, although this can be a lighter-touch review.

If you're working on a PR that requires a fairly stand-alone change, don't feel like you need to squeeze all your changes into a single commit or PR just so they can be reviewed in one hit. If we can get that standalone change merged separately first, then let's do that. Smaller commits and smaller PRs are far easier to manage, review and reason about.

3. Create a pull request on GitHub

Providing a good, detailed pull request description will help your work be reviewed well, and reduces the burden of effort on your colleagues who are reviewing.

Most Farewill repositories have Pull Request templates to help guide good PR descriptions. Any unused sections should be deleted before requesting a review. Sections like description, status, and the self-review checklist should all be completed before a pull request is considered ready for review, and updated as the work progresses.

🖼️ UI changes

- Consider including a screenshot or animated gif of your changes in the PR description. This saves the reviewer time and provides a nice point-in-time reference.
- Testing UI changes on the review app is the responsibility of the PR creator, not the reviewer
- Make sure you test across all our supported browsers. See 🌏 Browser support

4. Submit for review

Early reviews

You're also encouraged to open a PR early if you want an opinion on something before you spend to much time on it—just make sure you let the reviewer know it's a work in progress. When opening a pull request, try to give the PR a useful title. If you want the CI to run, you'll need to open a pull request. If you want to share your work for discussion but aren't ready for a review, feel free to open a draft pull request.

How to ask for help

- If you have general, high-level questions about the approach you've taken in a draft PR, use the PR description to explain your thinking, and to specify what kind of help you're looking for.
- If you have questions or want to call out a particular line of code for more detailed feedback, the easiest way to start a conversation is to use the GitHub UI to select the lines and add a comment. That way, any responses can be threaded in a comment thread and it's easy to see the specific pieces of code in question.

Picking a reviewer

Reviewers are assigned automatically. You have the power to change the reviewer once selected, so feel free to make an adjustment at your discretion. Reasons you might want to change the assigned reviewer include:

- You've used `git blame` to figure out who has worked on this area of the code; sometimes there will be a clear person who has the best knowledge to review your changes. (Use this option sparingly—generally, we want to spread knowledge through code review, so defaulting to a reviewer who has worked in this part of the codebase might not be the best idea. Or they might have worked on it a long time ago and won't remember much.)
- The assigned reviewer is very busy and won't have time to review, in which case you can pick someone who is not too busy at the moment (check their standup)
- You've been pairing on this work with someone who already has a lot of context about it, so it will be easier to have them review it.
- The assigned reviewer isn't at work or is otherwise unavailable

ℹ️ In general, please don't ask multiple people (or everyone) to avoid confusion. If you need input from multiple people, that's fine, just make sure you've made your

expectations explicit in the PR description or in comments.

- Example: "I've tagged @reviewer1 to take a look at the implementation approach in Foo, but I'm hoping for @reviewer2 to do a more thorough code review for general feedback"

Ideally wait until the CI server is green before requesting a review. Otherwise the reviewer isn't reviewing the final changes that will be merged.

During the review

If you're making changes that you'd like to discuss with the rest of the team, it's better to post a message in Slack and link to your PR. Draft PRs can be a good way to generate discussion on a code spike or multiple options for implementation solutions. It's often easier to talk about code when there's some code to look at, but your code doesn't need to be perfect or complete to start a discussion.

The reviewer(s) should look at your PR and make comments as appropriate. Others are of course welcome to chip in but are not expected to do so. The review process is likely to be iterative as you make further changes to respond to feedback.

It's important that while your PR is being reviewed that the person reviewing can see the incremental changes you have made, so resist the temptation (unless it makes absolute sense) to squash the commits until it's been approved.

Addressing review feedback

It can be tempting to make changes in response to PR feedback and bundle them up into one commit (e.g., "Code review feedback"). You *can* do this, but when you go to rebase your commits before merging, it will be hard to unpick the changes and put them in the right place.

A better solution is to make small commits with meaningful commit messages, where your commit message indicates the target change that you'll eventually squash into. So if you have a feature commit to `Add list of Tasks for Wills Cases` but you need to fix some typos, you might make another commit `Fixup task list to fix typos`. Once your PR is approved, you can easily reorder and squash the commit in the right place, discarding the latter commit message.

5. Merge and deploy

When you're happy that the reviewer's comments have been dealt with, merge the code and deploy. All developers are expected to merge their own code. Please ensure the commits are squashed into the fewest number of logical ones, and that you've rebased any fix-up commits from the review process. GitHub offers two good options for merging pull requests: rebase + merge, and squash + merge.

Rebase + merge

Rebase + merge is a great option if you're rebased your branch into atomic commits with meaningful commit messages and want to preserve this commit history when you merge your work into `main`.

Squash + merge

If your branch contains **only one commit**, or if it contains multiple commits that you want to squash into one, use the "squash and merge" option. Once you click "squash and merge" you'll have the chance to update the commit message for the commit that will be made to `main`. This is a great opportunity to make updates, like fixing typos in your commit message or rewriting the commit message to ensure it reflects any changes you've made during the review process.

Most of our repositories have a Heroku pipeline configured. If all goes smoothly, you can expect your changes to automatically deploy to production within 10 minutes or so or being merged into `main`.