**Pull Requests**

A pull request (PR) is a request submitted to a Gitlab repository to merge code into that project. The PR allows requested reviewers to view and discuss the proposed code. Once the PR passes all standards of reviews and all necessary revisions have been made, it can be merged into the codebase.

**Why are Pull Requests Important?**

Pull requests are important because they help ensure that quality reviewed code is merged into Gitlab repositories. Without PRs, messy and confusing code can easily run rampant in a codebase. Pull requests are a gatekeeping tool to prevent bad or malicious code from being merged into the codebase.

**Qualities of an Ideal Pull Request**

* Clear and concise
* Easy to review
* Inherently self-documenting

Please make sure that pull requests are small and are limited to a specific feature or task

**Commit Tips**

A pull request is made of a number of committed code changes.

An ideal commit makes it easy for others to understand the thought process that necessitated the changes made in each commit

Here are some guidelines you can follow to write a well-composed commit:

* [ ] Sum up the code changes you made in the present tense
* [ ] Keep commit messages as clear and concise as possible
* [ ] Limit the code changes to a single idea or step
* [ ] Avoid commits that have a ton of changes across many different files (unless they are all a similar change)
* [ ] Personally review your code

**Pull Request Tips**

Now that we have some well-structured commits, we can now push our changes to the Gitlab repository and open a brand new PR. Composing a PR is similar to composing a commit, except it encompasses the whole purpose of the code changes instead of the small changes made to each file or feature.

**Pull Request Title + Branch name**

For the pull request title, there's two major things to keep in mind:

* For the pull request title, sum up the general purpose of the PR as clearly and concisely as you can manage.
* It must link the task from your “task system” to the Gitlab PR **by using the prefix of the team with which the task is made, followed by the task number**

An example of a pull request title and branch name would be

**branch name**: SRE-12-fix-haze-deployment

**pull request**: SRE-12 fix haze deployment

**Pull Request Body**

For the pull request body, try to break it up into several sections (including but not limited to):

* Purpose of PR // What issues is it resolving?
* Main changes made
* Tests that passed // Edge cases covered
* Links to other PRs or issues

**The PR Discussion**

Once you have finished crafting your PR and have submitted it for review, developers will look over your PR and might make comments on your code to clarify something or ask you to make revisions. This discussion is important for ensuring high code quality and can help everyone find new or better ways to improve their Gitlab workflow.

Anything major that should need to change about the PR should also be noted in the task that was created for that change.

For this reason, it is important to respond to every comment made on your PR with either a 👍 or a comment to acknowledge that you made small changes requested, or with another comment if you need to explain something.

**Merging Pull Requests**

Once a Pull request is ready to go, it's up to the author of the pull request to merge the PR and

**PR Template**

* Feature/Bug description
* Related documentation
* Solution description
* Screenshots and videos
* Test cases
* Checklist

This template documents how to review code. Helpful for new and remote employees to get and stay aligned.

**Philosophy**

Why do you perform code reviews? What are your guiding principles for these reviews?

We review each other's code so that we can

* learn what changes are happening in our code base and get team
* learn from other people through dialogue
* get direct feedback about stying
* prevent nasty bugs by having multiple eyes

**Preparing Code for Review**

Preparation sets your reviewers up for success.

**Commit Messages**

Make sure your commit messages are thoughtful and descriptive.

**Gitlab PR Descriptions**

Your PR descriptions should be an extension of your commit messages. Write about both what the commit changes, and how you implemented the change.

**Performing Code Reviews**

**How to Review**

* Make two passes over the PR if it's substantial.
  + On the first pass, come to an understanding of the code change at a high level.
  + On the second pass, pay more attention to semantic details.

**Examples**

var commentCount = 0;

You might suggest that this be a let instead of var.

**Why?**

In order to ensure we have standard code across the board, we need automated tooling that has PR gating that enforces code quality. Having automated tooling takes the burden of PR reviewers to enforce code standards and simple coding bugs. This way the reviewers can focus more on the coding logic to make sure will meet business requirements and be performant.

**Features needed with tooling**

* PR gating to prevent merging
* Find dead code in our repos
* Comments above exported entities.
* Code duplication
* Code complexity (large functions, too many branches, etc)
* Code conventions (naming, etc)
* Security vulnerabilities
* Code coverage to enforce sufficient unit testing
* An easy way to view issues and identify where to add unit tests

Before You Start Building

**Create Documentation**

Document the feature *first*. Figure out how you're going to describe the feature to users; if it's not documented, it doesn't exist. Documentation is the best way to define a feature in a user's eyes.

**What do I use to document?**

For a really big systemic change, something that directly impacts product or other teams, etc., it makes the most sense to first document out your thought process on Notion.

**If the change is smaller/ not big picture, then it's best to probably skip this process and go straight into creating a task for the project on “your task system”.**

**Things to consider while documenting:**

1. What are you trying to solve?
2. Why are you solving it now?
3. What are the different approaches to solve this problem? What are their advantages and disadvantages?
4. Who will be impacted by this change? Who are the key stakeholders ? How will you communicate these changes to them?

Get your documentation reviewed by someone else before proceeding.

**Create Tests**

Once we know what we are trying to do, the best step to this is to build tests that describe the behavior of the new software. Good tests come in two formats:

1. Unit Tests
   * a test written by the programmer to verify that a relatively small piece of code is doing what it is intended to do.
   * narrow in scope
   * should be easy to write and execute
   * intended for the use of the programmer
   * if they do their job, testers and users downstream should benefit from seeing fewer bugs
   * Things outside the code under test are mocked or stubbed out
   * Shouldn't have dependencies on outside systems
   * Test internal consistency as opposed to proving that they play nicely with some outside system.
2. Integration Tests
   * Done to demonstrate that different pieces of the system work together
   * can cover whole applications and require much more effort to put together
   * usually require resources like database instances and hardware allocation
   * used for a wide variety of things, from full-on system tests against an environment made to resemble production to any test that uses a resource (like a database or queue) that isn't mocked out.
   * At the lower end of the spectrum an integration test could be a junit test where a repository is exercised against an in-memory database, toward the upper end it could be a system test verifying applications can exchange messages.

Confused on the difference between the two? [Check out this resource](https://stackoverflow.com/questions/5357601/whats-the-difference-between-unit-tests-and-integration-tests)

**Building**

Next, build the software!

This is a pretty complex idea that we won't talk about here. But be sure to leverage the tests that you have written, as well as the local environments that you have spun up in order to validate.

**Pull Request & Code Review**

Pull Requests, Tasking and Commits

The pull request is meant to be logically small, though not necessarily small in terms of line changes. What this means is that a 1000 line pull request should be logically similar. For instance, an endpoint to add a user, which has a lot of lines of test code, integration test code, and the code itself, is fine if its got 1000 lines or whatever the number is. If you start to complicate the pr by also working on a whole other feature that touches it, that hurts us.

You should also provide evidence of how you validated the change, including pictures of what you've built, how you plan to validate it in engineering, or anything you may need help with.

Finally, when you post the PR, it's a great help to others reviewing the pull request if you leave Gitlab comments throughout the PR, explaining generally what something does.

Code Reviews

**CI/CD Pull Request Validation**

Part of the Pull request validation is making sure that it can pass CI/CD, and be deployed healthily to our environments. We have four main checks:

* Build will ensure that your application builds
* unit-tests runs your unit tests that you have written

Add two scripts that should be run traditionally when running unit-tests:

total-coverage ⇒ total test coverage of the application

diff-coverage → test coverage of the most previous diff

They each take in a value that's going to dictate how much of a test coverage percentage that you want in the repository overall, as well as how much you want *in this specific pull request.*

**The hope is that as the application continue to mature, this is reflected in the amount of test coverage that the application is enforced at.**

* Integration tests spins up a sub environment using docker-compose, and will then run your tests
* lint will run “your lint module” on your code to ensure it's up to our standards. This will point out common flaws such as documentation, leaking secrets, extra whitepsaces, and much more.

**Deploy to Development**

After all checks are good from your pull request, which is validated in concourse, and 1-2 approvals have been attained on your commit, it's time to deploy to staging.

**You can deploy to development by simply merging the pull request to master.**

Once you merge to development, this will trigger a build job in the CI pipeline that your app is housed in. For instance, if you merged something to X, it would then start to build

Then, it will take this commit and apply it to the development environment, under the alias of latest.

**Deploy to staging**

Staging is a bit more intense of an environment to deploy to but should be similar to “deploy to development”

**Staging can act as QA if there is no dedicated QA env.**

**Deploy to Production**

Similar to the engineering environment, but would need manual approvals

**A commit is breaking production?**

If a commit is breaking prod, by now hopefully others are also aware of the scenario.

In order to roll back that commit, first

* cd repo && git stash && git checkout master && git pull
* git log --shorthand --no-merge
* find out which commit is breaking it, and which commit is a healthy commit.

You can view git commits doing the following

➜ git:(master) git log --short --no-merges

* Copy the commit that you know is still healthy, which is often a commit before the one that you believe is breaking the environment.

Given this is production software, this should all be done by a Site Reliability Engineer, or at the very least, someone with a lot of experience on the system.

**You can bring down the entire company's tech stack with a wrong move here. We provision out this access for good reasons. We don't want mistakes.**

* Once production has been rolled back, it's time to fix the repo. change directory to where the repo is. cd repo && git stash && git checkout master && git pull
* git log --shorthand --no-merge ⇒ Use this to find the commit that's breaking
* git revert <git commit hash/shorthand>
* git add --all
* git commit
* git push
* then roll out the change to staging, which will effectively remove the commit.
* Once validated in staging, you can push directly to production to fix the issue. This should be a no-op, as production is already running the image that you applied to it manually.

**A commit is breaking staging?**

* cd repo && git stash && git checkout master && git pull
* git log --shorthand --no-merge ⇒ Use this to find the commit that's breaking staging
* git revert <git commit hash/shorthand>
* git add --all
* git commit
* git push
* then roll out the change to staging, which will effectively remove the commit.

**Edge case: Rollback fails and need to push a hotfix commit**

**What if there are changes that are in staging, that we don't want to merge to production, and we need to roll out a hotfix to prod?**

Option 1

Have the commits that are not in production be validated and marked as ready for prod

Option 2

Make a separate change to revert those commits, then create a separate PR to merge your hot fix.

it's best to now strive to have those commits validated and ready to deploy to production, or have them taken out of the state of the repo and dropped back into a personal branch.