Intro to Gerrit

Everything about Gerrit: how to get started, advanced tips and tricks, and more!

- Getting Started with Gerrit
- Review Flow
- Why Gerrit?
- Gerrit with Jujutsu
- Tips and Tricks
- Advanced Features
- Troubleshooting

Getting Started with Gerrit

Thanks for showing interest in contributing to Lix! Gerrit can seem daunting at first, but it is our hope that you'll learn to navigate it and use it confidently after finishing this tutorial.

Perhaps the first question you have is: "Why Gerrit"? Well, glad you asked! In fact, we have an entire page describing that. But in short: it's just very nice for working with code. Instead of "PR from a branch" model that Github uses, Gerrit assigns a Change-Id to a commit, which is preserved even when the commit itself changes. This is perhaps familiar to Jujutsu VCS users.

This model allows us to largely ignore branches, which have long been known to introduce many operational complexities. Instead, we can focus on getting the work done, with our tooling supporting us in this goal!

Setting Up

Here are a few things you will need. Let's go over them one by one.

1. Go to our Gerrit instance and sign in. It will prompt you to sign in with Github SSO. If you don't want to use your Github account, that's fine; just hit us up on Matrix and we'll create you an account!
2. Get your SSH key ready! If you don't have one, run `ssh-keygen -t ed25519` to generate the SSH keypair with ed25519 cypher. It is the most secure SSH cypher currently available, and quite ergonomic to use! Passphrase isn't necessary for Gerrit, but if you want to use it, we recommend to also set up ssh-agent.
3. Go to your user settings in Gerrit, scroll down to "SSH keys", and add your public SSH key (from `~/.ssh/id_ed25519.pub`).

Configuring Git Repo

Now, we need to make some setup for the Git repo. If you haven't yet cloned Lix repo, use this fancy one-liner to do it (just remember to change `USERNAME` to your actual username!):

```
git clone "ssh://USERNAME@gerrit.lix.systems:2022/lix" && (cd "lix" && mkdir -p `git rev-parse --git-dir`/hooks/ && curl -Lo `git rev-parse --git-dir`/hooks/commit-msg https://gerrit.lix.systems/tools/hooks/commit-msg && chmod +x `git rev-parse --git-dir`/hooks/commit-msg)
```

If you want to get a deeper understanding, or you already cloned the repo from Forgejo - read on!

This is a little awkward, but yes - we store a Git repo in both Forgejo and Gerrit. This is a technical limitation of our infra. The "actual" repo lives **in Gerrit** - and it is automatically mirrored to Forgejo. This means that we want to push changes directly to Gerrit.

1. If you cloned the repository from Forgejo, rewrite the origin to point to Gerrit instead: `git remote set-url origin ssh://USERNAME@gerrit.lix.systems:2022/lix`
2. Gerrit wants a `Change-Id` footer in your commit message to work (and track changes to your commit). Adding it by hand is very inconvenient; thankfully, there's a Git hook that adds this footer if it doesn't already exist, even on amends of existing commits. It is added **automatically** if you are in `nix develop` shell. Otherwise, you can add it manually:

```
mkdir -p .git/hooks
curl -Lo .git/hooks/commit-msg https://gerrit.lix.systems/tools/hooks/commit-msg
chmod +x .git/hooks/commit-msg
```

3. Now you can just `git commit` your change. Before that, create a branch, but your commit is the unit of review here!
4. Gerrit also expects you to push in an unusual way, too: `git push origin HEAD:refs/for/main`. Just run this command to automate it: `git config remote.origin.push HEAD:refs/for/main`. Now you'll be able to `git push` like normal!
5. Now, you can just `git push`, and you'll see a link to your CL (Change List)! Now pat yourself on the back and wait for review :)

What Next?

Now, you should be getting a review in a few days - especially if your CL is quite small. Generally, Lix team goes through the open CLs quite often, and helps the new ones get to completion. Sometimes, people might be busy or just miss your CL - so don't take it to heart, and ask in Matrix for a review!

Once you got a review, you can address some issues - and fix others. When fixing issues, don't create new commits - use `git commit --amend` instead to amend the first commit that you pushed to Gerrit initially. Remember, each commit is a separate review piece - so unless you want to create a new CL, just amend the existing commit!

When the review process is done and everybody is satisfied, you'll get approvals from maintainers, and your change will become "Ready to submit". Now is your time to shine! Unlike Github, Gitlab or Forgejo, maintainers aren't the ones who can press the "Submit" button. You can get a last look at the change - and when you decide you're ready, push the "Submit" button, and your change will be part of the codebase. Congratulations!

Now, perhaps this experience was too easy, and you want to learn more Gerrit. Fair enough! Here are some pointers for you:

- We have a [different page] on Gerrit, that lists a bunch of cool tricks and features.

- You might want to check Github to Gerrit user guide: it largely goes over the same things as this documentation page, but it can give you more information about specific topics
- You might want to explore more of our Gerrit documentation!
- There's official user guide
- This user guide is part of official guide! There is also
- Github to Gerrit user guide, which this guide is largely based on.

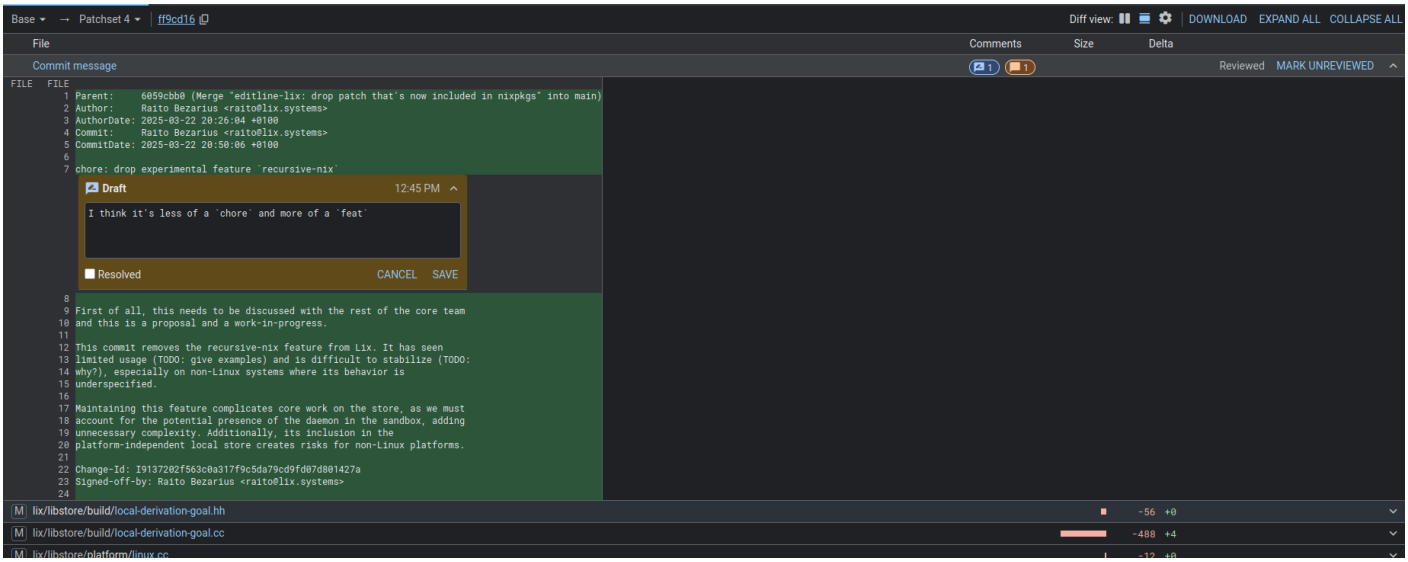
Good luck on your journey, and thanks for your contribution!

Review Flow

The review flow is quite similar to how Github does it, but there are a few differences here too. The UI also hides a few pretty powerful features!

Reviewer Flow

You can scroll down to the changes and start a review:

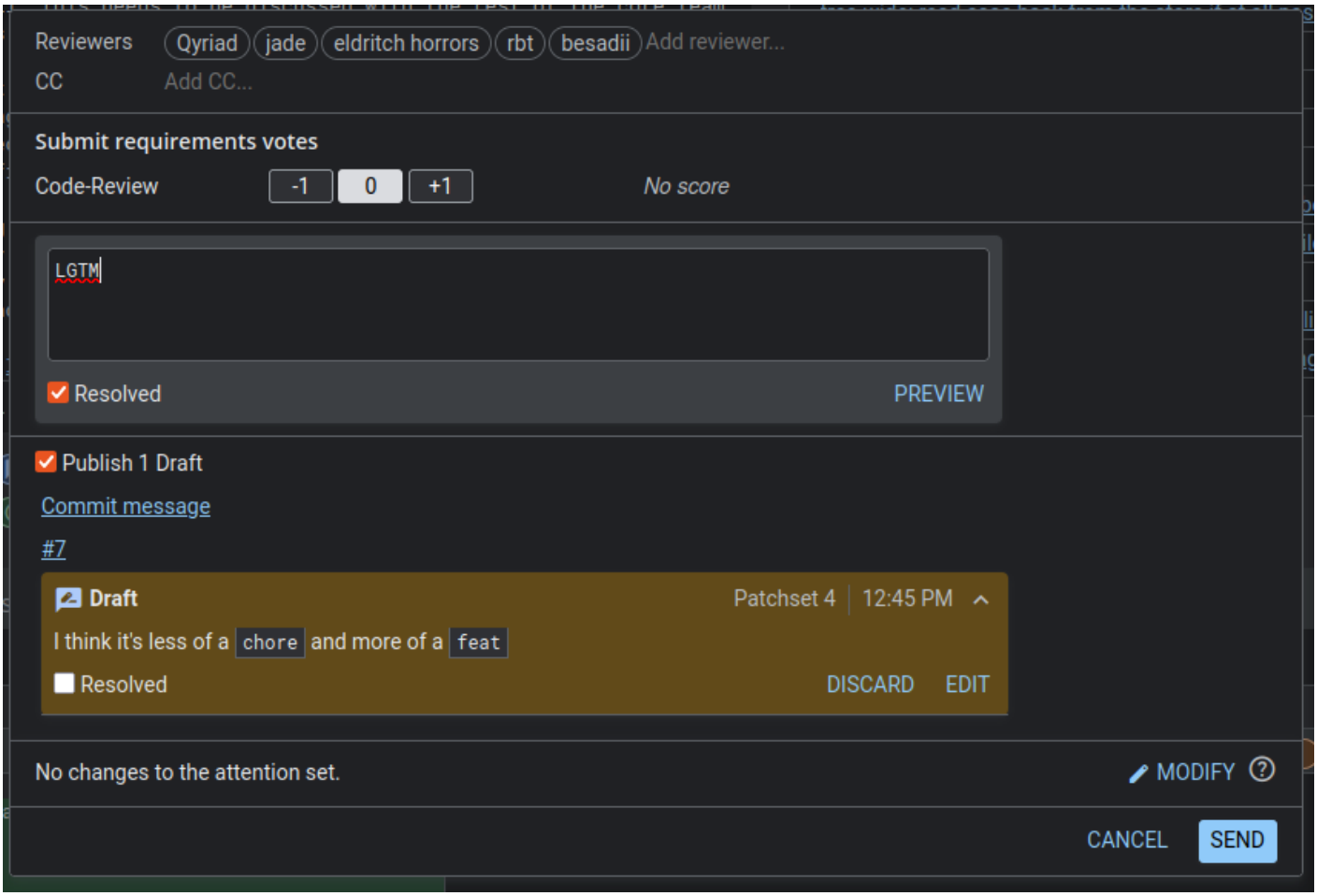


You can click on a line number to add a comment. Like on Github, the comments will be saved in a "draft" state, and will be submitted all at once when you finish the review.

When you expand the change to a file, the file will be marked as "reviewed" automatically - for your convenience. If you press "Expand All" button, you'll have to mark files as "reviewed" manually.

Finally, you can also change the "revisions" of comparison: you can choose to review the diff between Patchset [version] 3 and Patchset [version] 4! FIXME: this might be done automatically on re-review?

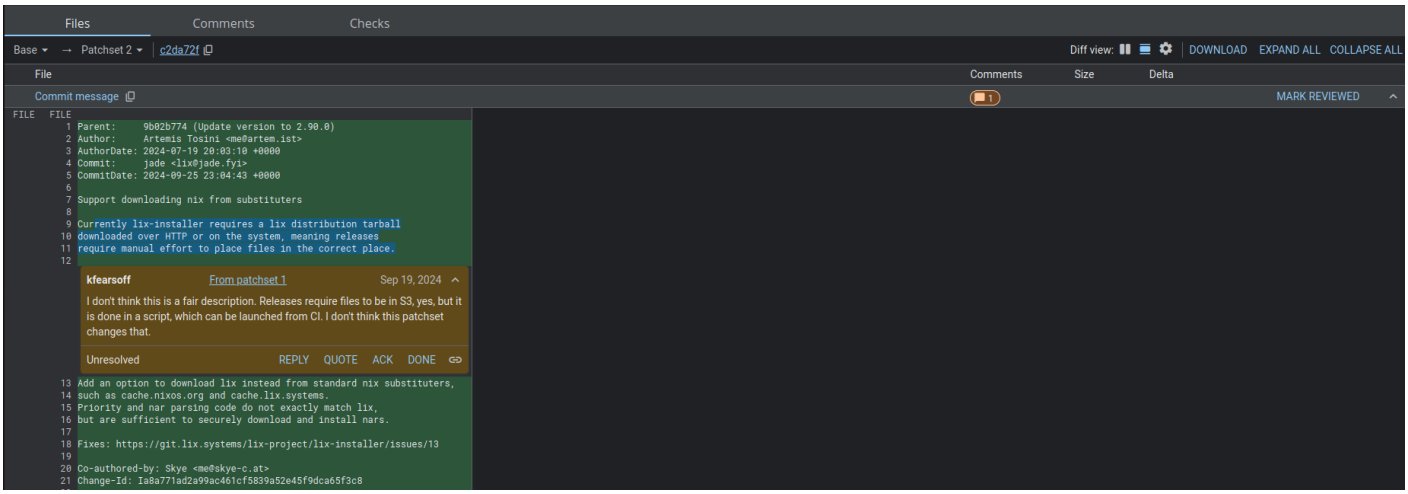
Finally, when the review is done, you scroll up and press the "Reply" button to submit a review, where you can add a final comment:



There, it's possible to mark some comments as resolved (maybe they're a non-important nitpick!), and also add a Code-Review vote. This is similar to Github's "approve/comment/request changes" on a PR, but a little more flexible. For a change to be ready for merge, it needs +2 on Code Review. Lix maintainers can +2 your change by themselves, so in normal circumstances, your change will require maintainer approval or 2 reviews from non-maintainers.

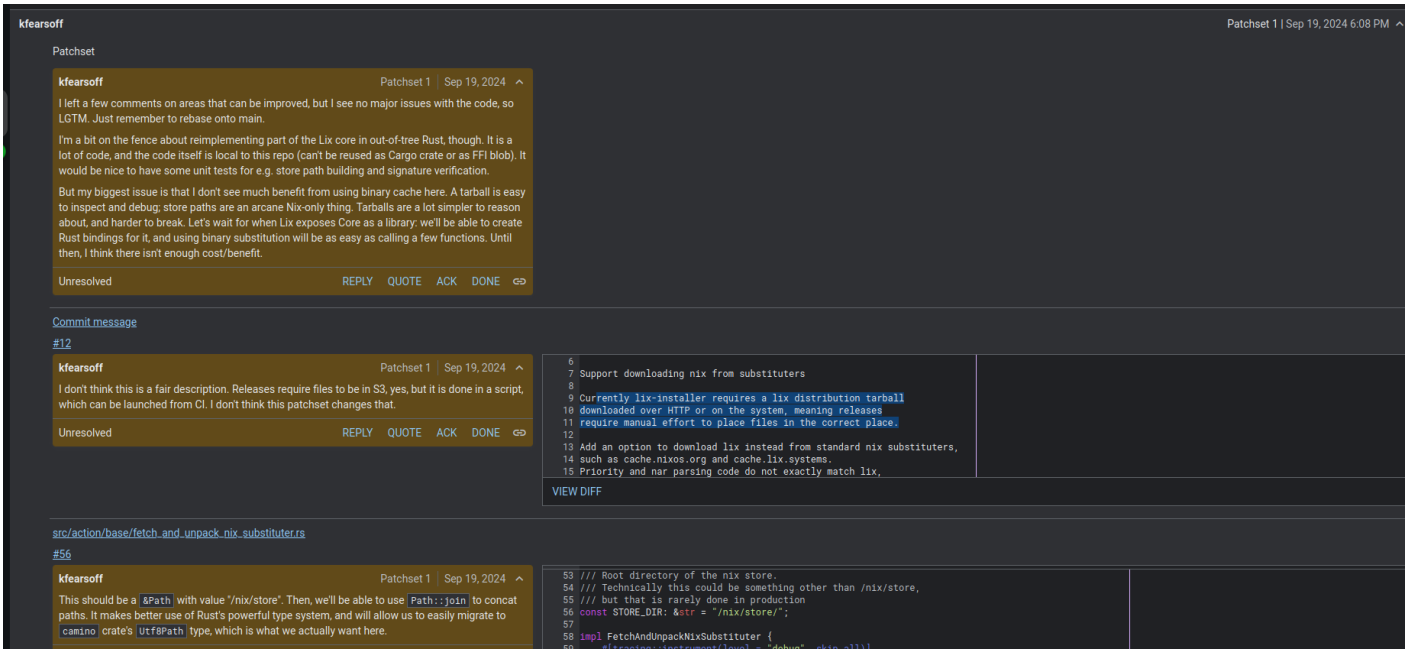
Reviewee Flow

There are two ways to view the review. One is to look at it directly in code:



This will show you the whole comment thread.

Another way is to scroll down to the change log and expand a review:

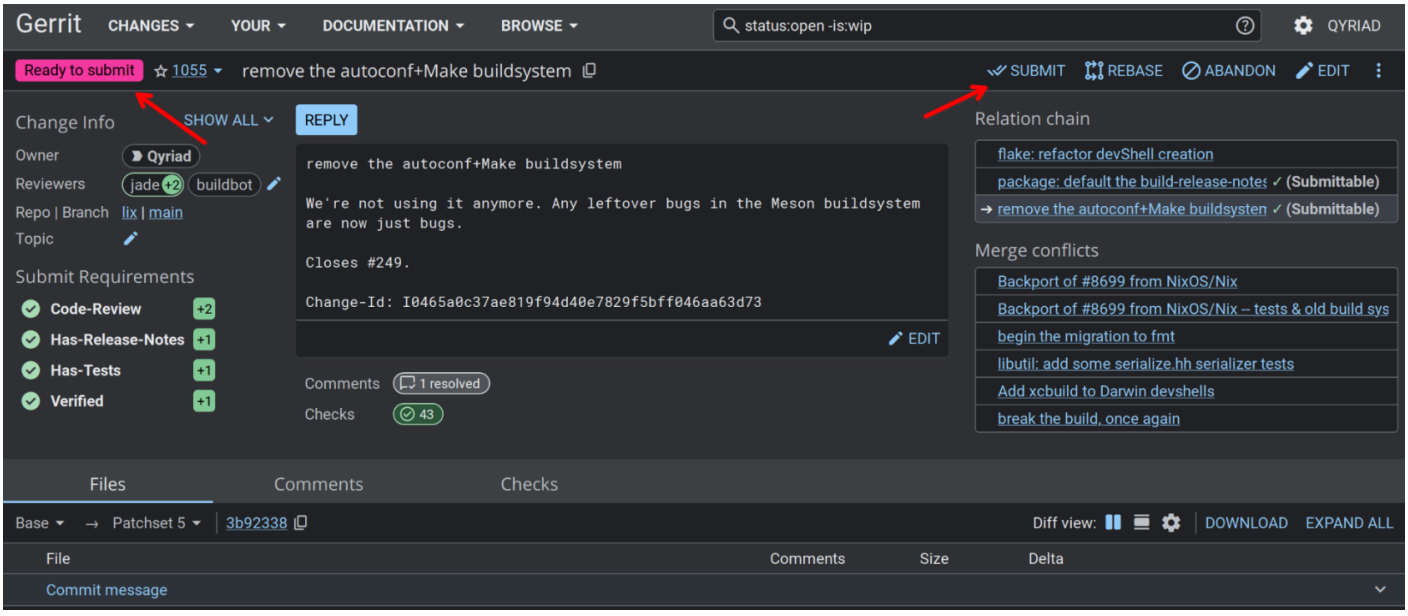


This will show you the entire review at once, with threads collapsed.

Choose whichever you like best!

Merge Flow

Once approved, the change can be merged. Maintainers will sometimes do that, but we generally prefer to leave the decision to the person who submitted the change! Take a final look at your change, verify that everything looks good, and press the "Submit" button!



Why Gerrit?

Gerrit produces better code:

- Gerrit enforces good commit messages. "PR message" and "commit message" are the same thing in Gerrit; there's no duplication, and information about a change can be seen in regular commit history.
- Gerrit enforces good commit hygiene, since adding another commit is really just splitting a commit with `git revise -c` or other tools; since there are no PR dependencies or branches to worry about, splitting commits is no longer a big ask.
- Relatedly, this directly makes reviews smaller since the overhead of doing another change is low.

Gerrit makes reviewers' lives easier and reduces review round trips:

- As a reviewer, you can look at what changed since you last reviewed, even in the presence of rebases, by looking at the patchset history of a CL. This avoids pointless rereview; you can actually diff versions of changes properly.
- The change author generally merges the change after approval, without them needing commit access. This means that they can do a final once-over of the change and make sure that they are ok with its state before merging it. This reduces miscommunication causing merging of unfinished code.
- As a reviewer, you can edit someone's change and/or commit message to fix a typo (in the web interface) and then stamp it, while giving them the final say on merging the edited change.
- You can give feedback like the following: "I would merge this as-is but you can consider this feedback if you would like" and then let the change author decide to merge it.
- Since the permission-requiring step in Gerrit is approving the change, not merging it, every change author can have final say in when the change gets merged.
- Review suggestions get applied as a batch without cluttering commit history in a confusing manner.
- You can download someone's change to look at it locally in one command that you can copy paste from the Gerrit interface (keyboard shortcut: `d`).

Gerrit makes your life easier as a contributor:

- Submitting a new change is just a matter of committing it and pushing it. You don't need to think about branches or the web interface or extra commands. Want to do more changes building on it? Just commit them and push them.
- Branches are not required and you can easily build off of other peoples' changes by fetching them and rebasing against them; change dependencies are simply commit parents. They can then be merged in whichever manner they will be merged.
- If you are doing a larger change, it is natural to merge it piece by piece, adding little improvements as you go, and putting the highest risk parts of it at the tip, making the

obviously good parts of the change land and keeping your diffs and rebases against main smaller.
- Gerrit makes it clear which comments still need action in a clean way, compared to GitHub where resolved comments get regularly broken or disappear altogether.
- Gerrit guesses (with reasonable accuracy) who a change is blocked on and shows it on the dashboard with a little arrow next to their name, allowing you to see at a glance which changes are your responsibility at a given time.
- There is a rebase button that just works. Trivial non-conflicting rebases do not require a rereview.

That being said, there are some downsides:

- Gerrit is very mean to you if you don't have your commit history in a clean presentable state, which takes some getting used to and Git does not make editing history easy, so it does involve a little more fighting of Git. However, this also means that the reviews can be of cleaner and smaller pieces of code with fewer unrelated changes.
  - **jujutsu fixes this**: if you are finding an amend based workflow frustrating, we highly recommend using Jujutsu in place of Git. See Jujutsu usage with Gerrit for more details.
  - This makes pushing work in progress code with questionable commit history harder; see below for solutions to this.
- Gerrit requires a little bit of local setup in the form of adding your SSH key or setting up the HTTP password. It also requires a Git commit-msg hook, but nix develop automatically does that for you.

Gerrit with Jujutsu

Jujutsu has a very similar model to Gerrit and natively supports sending Gerrit changes!

You'll need to configure a Gerrit remote. If it's not `origin`, you can instruct Jujutsu to use it by specifying it in `.jj/repo/config.toml`:

```
[gerrit]
default-remote = "gerrit"       # name of the Git remote to push to
default-remote-branch = "main"  # target branch in Gerrit
```

After that, `jj gerrit upload -r <revision>` will automatically add `Change-Id` footers and send your changes to Gerrit, printing a link to a resulting patchset to the terminal.

See Jujutsu docs for more information.

Tips and Tricks

SSH Tuning

Add these lines to your `~/.ssh/config`:

```
Host gerrit.lix.systems
    User YOUR_GERRIT_USERNAME
    Port 2022
    # Keep sessions open for a bit in the background to make connections faster:
    ControlMaster auto
    ControlPath /tmp/ssh-%r@%h:%p
    ControlPersist 120
```

Now you can use `ssh://gerrit.lix.systems/lix` instead of `ssh://USERNAME@gerrit.lix.systems:2022/lix` URL, and have faster iterations on a change!

Splitting Commits

Sometimes a commit that was supposed to be a single feature gets out of hand. You have options. `git rebase -i` is a "default" suggestion, but you might want to look at `git-revise` or even at Jujutsu, which has a pretty cool squash workflow to do this!

Advanced Features

If you feel confident in your Gerrit-fu, this page is for you. Perhaps you've already noticed how Gerrit brings a few good improvements to the workflow just based on the commit-centric design. Now we are getting to the really cool stuff: Gerrit-specific features that further enhance your experience!

WIP Changes

Want to mark a change as WIP? You can do it in the Web UI. Or you can do it immediately on push:

```
git push origin HEAD:refs/for/main%wip
```

Submitting Chains of Changes

In traditional Git forges, it is hard to submit multiple changes to one repo at once. Perhaps you want to avoid unnecessary rebuilds - or broken intermediate states. You might have to coordinate with people, plan downtime, or create meta-PRs to do that.

Thankfully, Gerrit has topics for that! Assign a topic to necessary CLs, or set it on push:

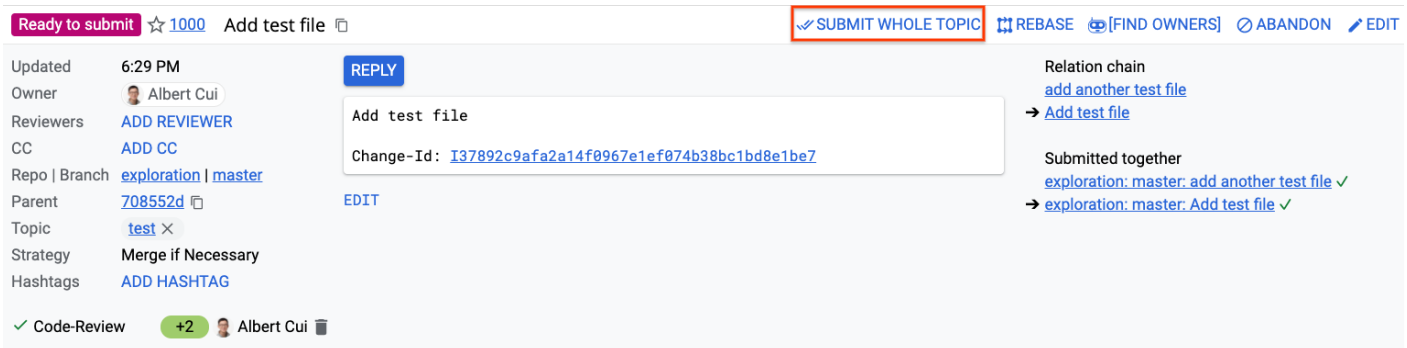
```
git push origin HEAD:refs/for/master%topic=<username>-<ISO8601 date>-<topic name>
```

For example:

```
git push origin HEAD:refs/for/master%topic=kfearsoff-2025-03-29-rebranding
```

Why the convoluted naming scheme? Unfortunately, Gerrit doesn't namespace topics, so we do a funny naming scheme to prevent accidental collisions. We are truly sorry for this.

Once all changes in a topic are reviewed, they can be gracefully sent all at once!



Syncing Multiple Repos

In traditional Git forges, it is always a huge pain when you need to submit a batch of changes at once to multiple repos.

Topics to the rescue, again! Just follow the instructions for chains of changes, but in multiple repos. Isn't that awesome?

Troubleshooting

"Remote Unpack Failed" on push

Run a `git fetch` and try again.

Re-run a CI

Do an empty commit amend: `git commit --amend`. This will change the commit ID (because the date has changed!), so it will now be a new CL revision: you can `git push` it and pray for good CI run!