# Introduction

## What Is Gerrit?

Gerrit is a Git server that provides

* Code review
* Access control on the Git repositories

Gerrit is open-source, built on top of Git and in forms of a **web**.

## How Does Gerrit Work?

To understand how Gerrit fits into and enhances the developer workflow, consider a typical project.

|  |  |
| --- | --- |
| This project has a central source repository, which serves as the authoritative copy of the project’s contents.  Central Source Repository | Gerrit takes the place of this central repository and adds an additional concept: a *store of pending changes*.  Gerrit in place of Central Repository |

With Gerrit, **any change made by a developer is sent to this *store of pending changes*, where other developers can review**, discuss and approve the change. **After enough reviewers grant their approval, the change is submitted** and becomes an official part of the codebase.

## Why Is Gerrit Used?

* **Code Review**: Gerrit not only enfores **commit reviews before they are integrated into a target branch**, but also provides a **simple, quick, beautiful UI for code review** where devs can question, reply and notify each other straingforwardly.

*Code review is optional, but required by default (bypassing code review can be allowed by granting access rights for direct push)*

* **Access Rights**: Gerrit provides **fine-grained read and write permissions on branch level** (with raw Git only you always have access to everything once you can access a repository) .

This presentation concentrates on the code review aspect, access controls are not covered.

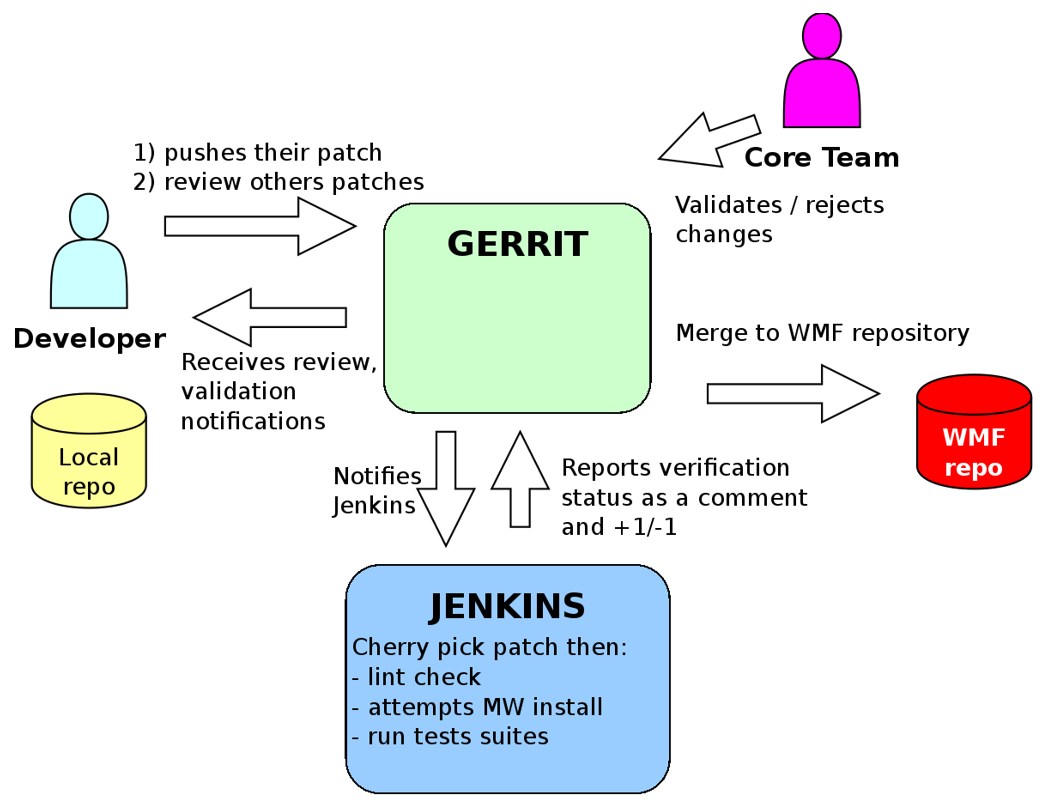
* **Process Ensurance**: Teamwork becomes easy as Gerrit provides step by step of what team members have to complete before a commit is finally done.
* **Ease of Teamwork**: Collaborating between many devs on a specific feature is easy as Gerrit manages each commit on a branch automatically. Devs don't need to manage any feature branch, but just focus on the commit and the target branch.
* **Genkin Integration**: By having Genkin integrated with Gerrit, your code compiling and testcase execution can be done automatically right after a commit is pushed.

**Refs:**

* Official guideline: <https://gerrit-review.googlesource.com/Documentation/>
* Official manual: <https://gerrit-review.googlesource.com/Documentation/user-review-ui.html>
* Gerrit demo: <https://gerrit-review.googlesource.com/q/project:gerrit>
* Slide: <https://docs.google.com/presentation/d/1C73UgQdzZDw0gzpaEqIC6SPujZJhqamyqO1XOHjH-uk/edit#slide=id.g4d6c16487b_1_46>

# Workflow

## Diagram



## Real Life Example

In this example, we'll follow 2 developers, Max (junior dev) and Hannah (senior dev). Max is the one who writes code, and Hannah is responsible for reviewing and approving Max's code. We'll follow their work through these steps:

1. **Make the change (by Max)**

Max has decided to make a change to the *RecipeBook* project he works on. His first step is to clone the source code:

$ git clone ssh://gerrithost:29418/RecipeBook.git RecipeBook

Then, he adds a Change-Id to his commits:

$ scp -p -P 29418 gerrithost:hooks/commit-msg RecipeBook/.git/hooks/

$ chmod u+x .git/hooks/commit-msg

**Note**:

Here, the Gerrit server is configured as follows:

* Hostname: *gerrithost*
* HTTP interface port: *80*
* SSH interface port: *29418*

1. **Create the review (by Max)**

Max pushes his change to Gerrit so Hannah (or any other) can review it. He runs:

$ git commit -m "Change to a proper, yeast based pizza dough."

[master 3cc9e62] Change to a proper, yeast based pizza dough.

1 file changed, 10 insertions(+), 5 deletions(-)

$ git push origin HEAD:refs/for/master

Counting objects: 3, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 532 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

remote: Processing changes: new: 1, done

remote:

remote: New Changes:

remote: http://gerrithost/#/c/RecipeBook/+/702 Change to a proper, yeast based pizza dough.

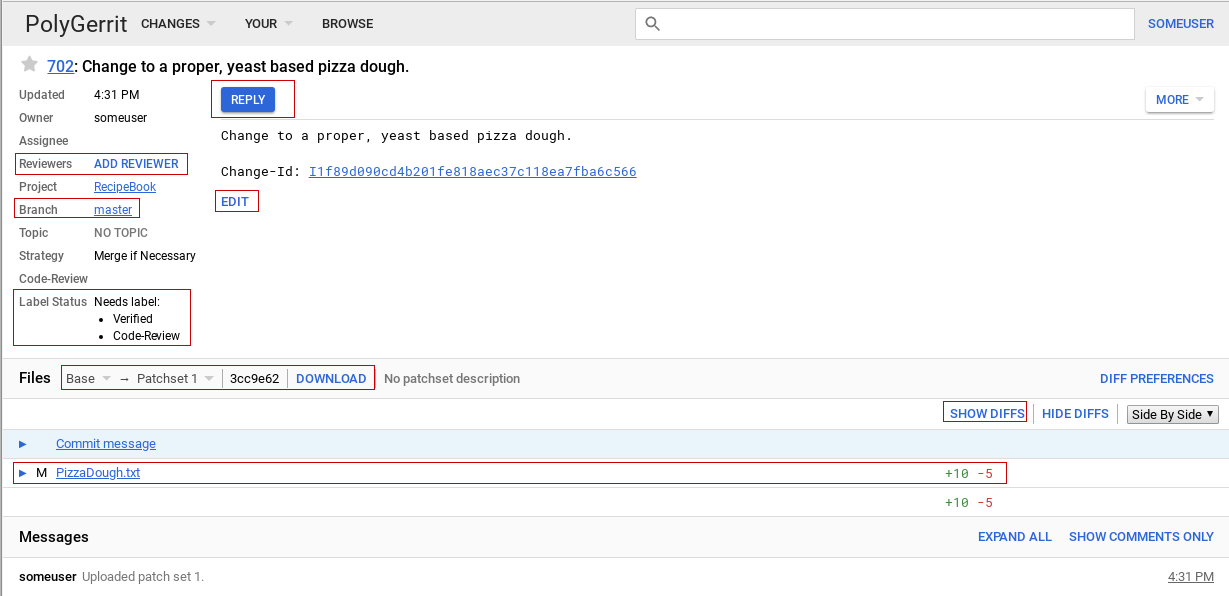
remote:

To ssh://gerrithost:29418/RecipeBook

\* [new branch] HEAD -> refs/for/master

**Notes**:

* Gerrit **uses refs/for/master branch to create reviews** for the master branch.
* The output contains **a link to a web page** which takes Max to a Gerrit code review screen:



Here, Max can:

* Look at the [diff](https://gerrit-review.googlesource.com/Documentation/user-review-ui.html#diff-preferences) of his change.
* Write inline or summary [comments](#_Comment) to ask reviewers for advice on particular aspects of the change.
* [Add reviewers](#_Adding_Reviewers) for the change. In this case, Max manually adds Hannah to review his change.

1. **Review the change (by Hannah)**

Gerrit offers several ways for reviewers to find changes:

* Using the [search feature](#_Search_Change) that to find changes.
* Selecting *Change* > *Open*.
* Setting up [email notifications](https://gerrit-review.googlesource.com/Documentation/user-notify.html) to stay informed of changes even if you are not added as a reviewer.

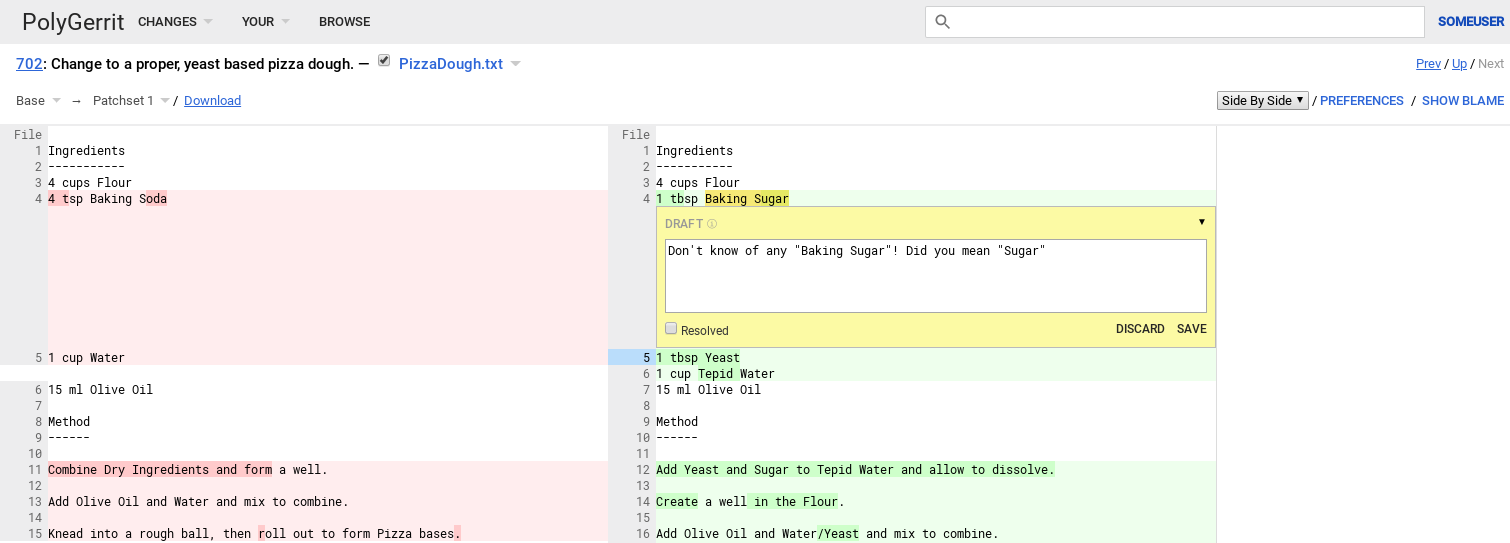
Because Max added Hannah as a reviewer, she receives an email telling her about his change. She opens up the code review screen and selects Max's change.

Notice the *Label status* section above, *Code-Review* and *Verified*. They **indicate what checks must be completed before the change is accepted**. The default Gerrit workflow requires two checks:

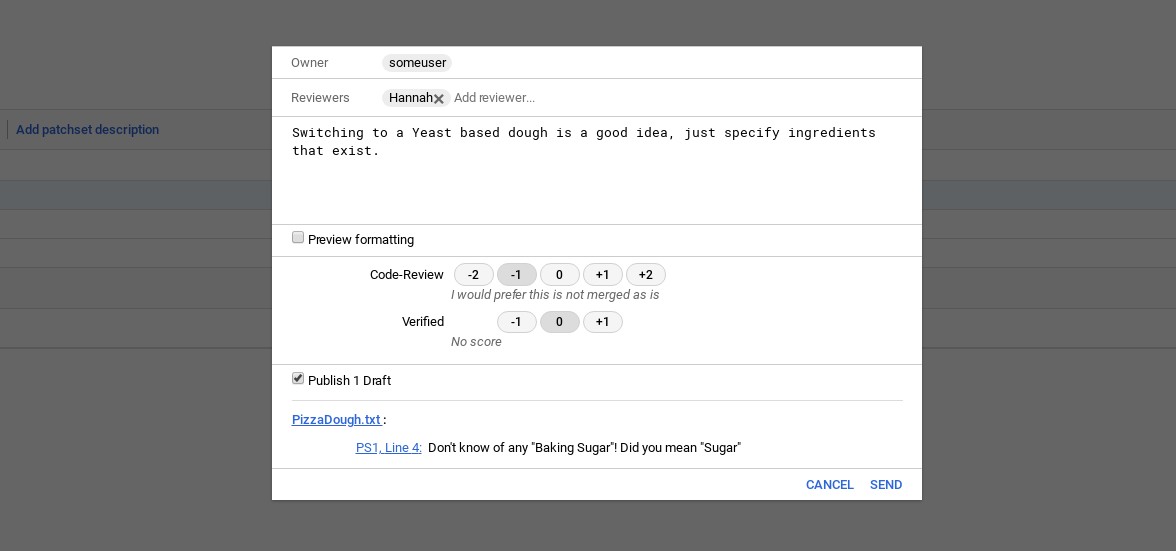
* *Code-Review*: This check requires that **the code meets project guidelines, styles, and other criteria**. In general, it requires an individual to look at the code.
* *Verified*: This check means that **the code actually compiles, passes any unit tests, and performs as expected**. In general, it is done by an automated build server, through a mechanism such as the [Gerrit Trigger](https://plugins.jenkins.io/gerrit-trigger/).

With the Code Review Screen open, Hannah can review Max's change either *unified* or *side-by-side*.

Hannah chooses side-by-side view and adds an inline comment:



Then she votes Max's change by clicking the *REPLY* button:



The code review vote is a numerical score between -2 and 2. The possible options are:

+2 Looks good to me, approved

+1 Looks good to me, but someone else must approve

0 No score

-1 I would prefer this is not submitted as is

-2 This shall not be submitted

**A change must have at least one +2 vote and no -2 votes before it can be submitted**. These numerical values do not accumulate, so two +1 votes do not equate to a +2.

**Tip**: These settings are enabled by default. To learn about how to customize them for your own workflow, see the [Project Configuration File Format](https://gerrit-review.googlesource.com/Documentation/config-project-config.html) topic.

Hannah notices a possible issue with Max's change, so she selects a -1 vote. She uses the Cover Message text box to provide Max with some additional feedback. Once done, she clicks the *SEND* button. At this point, her vote and cover message become visible to to all users.

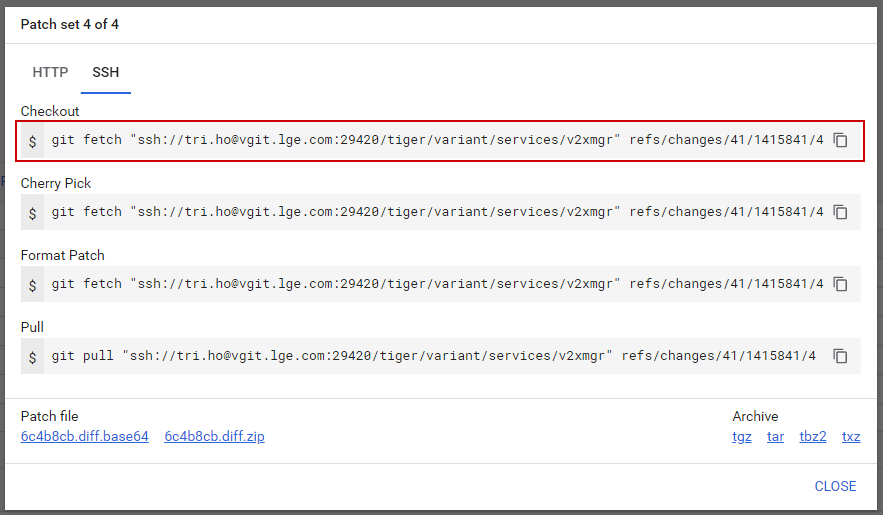
1. **Rework the change (by Max)**

Later in the day, Max checks and notices Hannah's feedback. All he has to do is to follow the typical Git workflow for updating a commit:

1. Check out the commit (if needed)

Because Max's commit has not been approved and merged to the master branch, Max cannot work directly on the master branch.

But he has to work in the so-called [detached HEAD](https://www.git-tower.com/learn/git/faq/detached-head-when-checkout-commit) mode. He selects the latest patch set, then uses the *Download* button to copy a command that fetches and checks out the commit underlying your change.

**

*Note: This image is a random image. Don’t care about its real content*

1. Amend the commit

$ git commit --amend

[master 30a6f44] Change to a proper, yeast based pizza dough.

Date: Fri Jun 8 16:28:23 2018 +0200

1 file changed, 10 insertions(+), 5 deletions(-)

**Notes**:

* Gerrit will automatically **create a new patch set for each commit**. You can view each one by clicking on *Patch set XXX* dropdown list.
* Because Max amends to the previous commit instead of creating a completely new commit, his change now **contains two patch sets** (the first one is the original commit, the second one is the latest commit). The number of patch sets will be increased by 1 each time a commit is amended.

1. Rebase the commit (if needed)

Before doing any commit, Max wants to make sure that his change is on top of the branch.

$ git fetch

He got no output, which is good. The master branch has not progressed and there is no need for rebase.

1. Max is now ready to push his change

$ git push origin HEAD:refs/for/master

Counting objects: 3, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 528 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

remote: Processing changes: updated: 1, done

remote:

remote: Updated Changes:

remote: http://gerrithost/#/c/RecipeBook/+/702 Change to a proper, yeast based pizza dough.

remote:

To ssh://gerrithost:29418/RecipeBook

\* [new branch] HEAD -> refs/for/master

**Notes**:

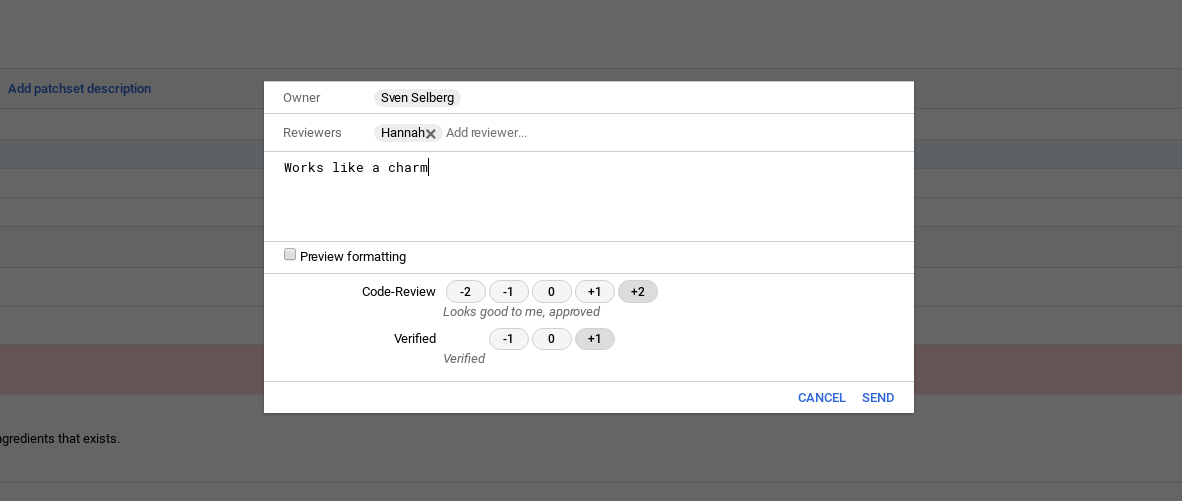
* The output is slightly different from Max's first commit. This time, the output verifies that the change was updated.
* Finally, Max can go back to the Gerrit web to **press the *DONE* button to let Hannah know** that she can review the changes.

1. **Verify the change (by Hannah)**

When Hannah confirms that Max's change incorporated her feedback, she changes her vote to a +2. This means the change also satisfies the *Review* check. It now just has to pass the *Verified* check before it can be accepted.

Verification is typically an automated process using the [Gerrit Trigger Jenkins Plugin](https://plugins.jenkins.io/gerrit-trigger/) or a similar mechanism. However, there are still times when a change requires manual verification, or a reviewer needs to check how or if a change works.

To manually verify a change, a reviewer must have the [Verified](https://gerrit-review.googlesource.com/Documentation/config-labels.html#label_Verified) permission. Then, the reviewer can fetch and checkout that branch from Gerrit. Hannah has this permission, so she is authorized to manually verify Max’s change.



Here Hannah can provide a score of either +1 or -1. **A change must have at least one +1 and no -1.**

Hannah selects a +1 for her verified check.

1. **Submit the change (by Max)**

Max can now submit his change. He opens up the change in the Code Review screen and **clicks the *SUBMIT* button**.

At this point, Max’s change is merged into the repository’s master branch and becomes an accepted part of the project.

**Notes**:

* Here Max is the one who submit the change. But depending on the project workflow, Hannah can be responsible for doing this as well.
* There can be conflicts while merging. In this case, Max or Hannah has to resolve all of the conflicts to finish mergining.

# Concepts

## Change

A change consists of following **components**:

* One or more patch sets
* Change-Id
* Meta data (owner, project, target branch etc.)
* Comments
* Votes

A change can have following **status**:

* Needs <label>: The change is in review and an approval on the shown label is still required to make the change submittable.
* Not <label>: The change is in review and a veto vote on the shown label is preventing the submit.
* Not Current: The currently viewed patch set is outdated. Please note that some operations, like voting, are not available on outdated patch sets, but only on the current patch set.
* Ready to Submit: The change has all necessary approvals and may be submitted.
* Merged: The change was successfully merged into the destination branch.
* Abandoned: The change was abandoned.

## Patch Set

A patch set is a **Git commit**. Each patch set in a change represents a new version of the change and replaces the previous patch set. Only the latest patch set is relevant. This means all failed iterations of a change will never be applied to the target branch, but only the last patch set that is approved is integrated.

## Change-Id

Change-Id is an unique **SHA1 hash of the first commit of the change**. It's prefixed with an I. It is specified as footer in the commit message (last paragraph). While the SHA1 hash of your change (the Git commit ID) might change, the Change-Id stays the same.

Change-Id is important for Gerrit to know **whether a commit should create a new change** or **whether it should create a new patch set for an existing change**. The mechanism is straightforward:

* If a commit has a Change-Id in its commit message, Gerrit checks if a change with this Change-Id already exists for this project and target branch. If yes, Gerrit creates a new patch set for this change. If not, Gerrit creates a new change with the given Change-Id.
* If a commit without Change-Id is pushed for review, Gerrit creates a new change and generates a Change-Id for it.

**Tips**:

* Most projects already [require a Change-Id](https://gerrit-review.googlesource.com/Documentation/project-configuration.html#require-change-id) when pushing the very first patch set. This reduces the risk of accidentally creating a new change instead of uploading a new patch set. **Any push without Change-Id then fails** with [missing Change-Id in commit message footer](https://gerrit-review.googlesource.com/Documentation/error-missing-changeid.html).
* Change-Ids are unique for a branch of a project. So, **commits that fix the same issue in different branches should have the same Change-Id**. This happens automatically if a commit is cherry-picked to another branch. This way you can [search](https://gerrit-review.googlesource.com/Documentation/user-search.html) by the Change-Id to find a fix in all branches.
* Change-Ids can be created automatically by installing the commit-msg hook. Check [here](https://gerrit-review.googlesource.com/Documentation/user-changeid.html#creation).

**Note**:

* Never amend a commit that is already part of a central branch.

## Action

Depending on the change status and the permissions of the user, different actions are available on the change:

### Submit

Submits the change and **adds it to the merge queue**. If possible the change is merged into the destination branch.

The *Submit* button is only available if the change is submittable and the Submit access right is assigned.

### Revert

Reverts the change via **creating a new one**.

The *Revert* button is only available if the change has been submitted.

### Abandon

Abandons the change.

The *Abandon* button is only available if the change is open and the Abandon access right is assigned.

### Restore

Restores the change.

The *Restore* button is only available if the change is abandoned and the Abandon and the Push access right is assigned.

### Rebase

Rebases the change.

The *Rebase* button is only available if the Rebase access right is assigned.

If the rebase is successful, **a new patch set with the rebased commit is created**. Otherwise, there are conflicts that have to be resolved manually.

If the change does not depend on another open change, it's rebased onto the tip of the destination branch. Otherwise, it's rebased onto the current patch set of that open change.

### Cherry-Pick

Allows to **cherry-pick the change to another branch**. The destination branch can be selected from a dialog. Cherry-picking a change creates a new open change on the selected destination branch.

It's possible to cherry-pick a change to the same branch. This is effectively the same as rebasing it to the current tip of the destination branch. This can be used to remove dependencies on other open changes.

Users can only cherry-pick changes to branches for which they are allowed to upload changes for review.

### Delete Change / Delete Revision

For open or abandoned changes,

The *Delete Change* button will be available if the user is the change owner and is granted the Delete Own Changes permission, or if they are granted the Delete Changes permission, or if they are an administrator.

* *Further actions may be available if plugins are installed.*

## Related Changes

They are changes related to the currently viewed change. They're displayed in the third column of the change screen.

Related changes can be labeled as:

* Relation chain
  + Not current
  + Indirect descendant
  + Merged
  + Submittable
  + Abandoned
* Merge conflicts
* Submitted together
* Cherry-picks
* Same topic

## Inline comment

The comments **support markdown**. It follows the CommonMark spec, except inline images and direct HTML are not rendered and kept as plaintext.

Comments are **first saved as drafts**, and you can revisit the drafts as you read through code review. Finally, they will be **published by clicking the *Reply* button**.

Comments can be unresolved (something should be changed) or resolved (informational). If you have addressed an unresolved comment in a next patchset, you can quickly resolve the comment by clicking *Done* (if it was resolved in a next patchset) or *Ack* (if you acknowledge the comment, but don’t want to make changes).

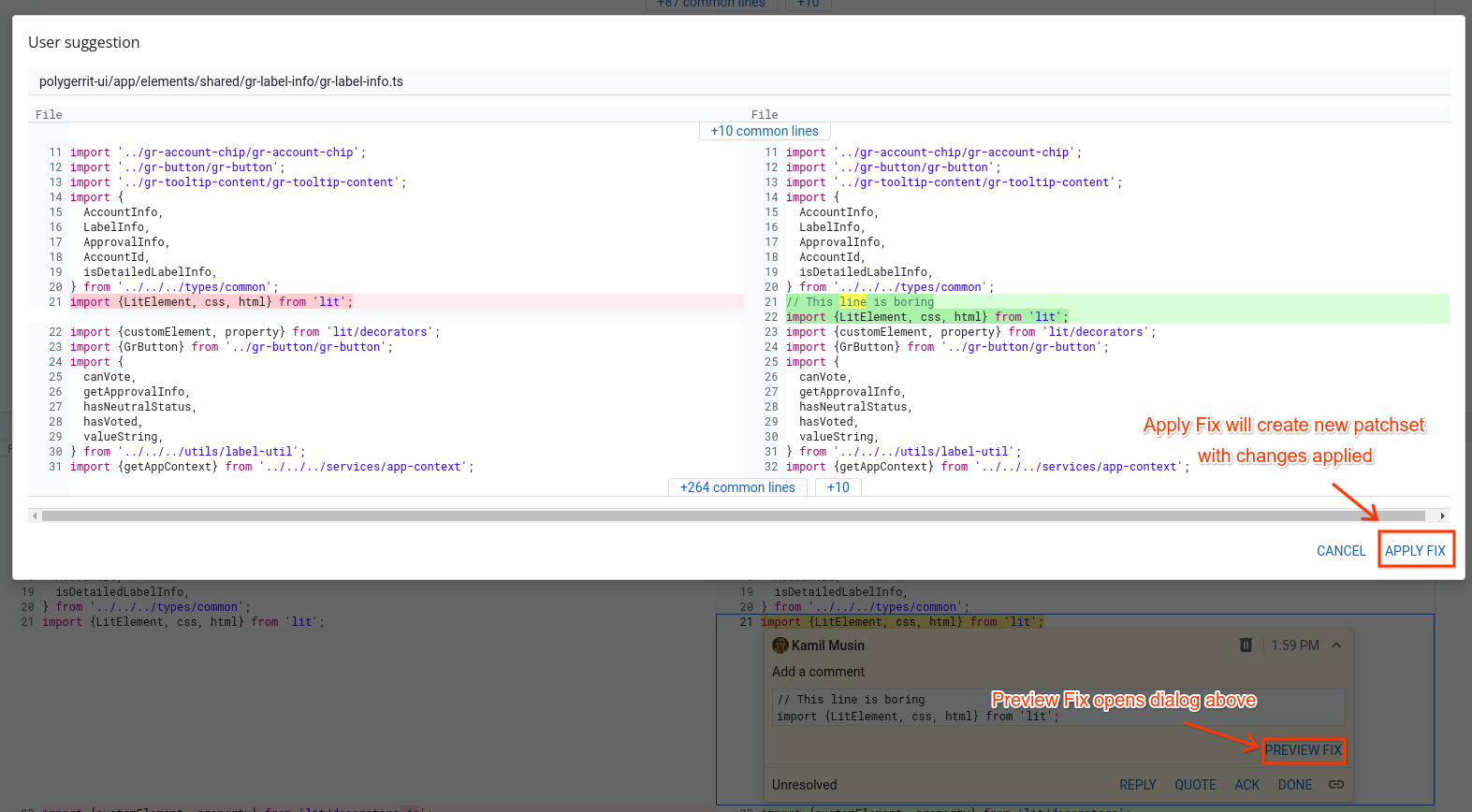
### Suggest Fix (WIP)

Comments can contain suggested fixes.

Clicking *Suggest Fix* will insert a special code-block in the text of the comment. The contents of this code block will replace the lines the comment is attached to (what gets highlighted when hovering over comment).

[](https://gerrit-review.googlesource.com/Documentation/images/user-review-ui-suggest-fix.png)

The author of the change can then preview and apply the change. This will **created a new patchset with changes applied**.

[](https://gerrit-review.googlesource.com/Documentation/images/user-review-ui-apply-fix.png)

# Clone Repo

Cloning a Gerrit project is done the same way as cloning any other Git repository by using the git clone command.

For example:

$ git clone ssh://gerrithost:29418/RecipeBook.git RecipeBook

Cloning into RecipeBook...

**Notes**:

* The URL can be found in the Gerrit web under *Projects*> *List*> *<project-name>* > *General*.
* Gerrit supports the [SSH](https://gerrit-review.googlesource.com/Documentation/user-upload.html#ssh) and the [HTTP/HTTPS](https://gerrit-review.googlesource.com/Documentation/user-upload.html#http) protocols.

To use SSH you may need to [configure your SSH public key in your Settings](https://gerrit-review.googlesource.com/Documentation/user-upload.html#ssh).

# Commit

## Review every commit

**Every single commit has to be reviewed** before it's accepted into the code base.

If there is comment, the author can improve the code **modification by amending the commit** and uploading it as a new patch set.

How is this different from GitHub?

With GitHub, you can commit to your feature branch as much as you like. The sum of all your commits on that branch will get reviewed.

By contrast, as a single commit gets reviewed in Gerrit, you need to run git commit --amend when you iterate on the same change. This is opposed to using git commit with GitHub.

## Must merge first commit before submitting second commit

You can, of course, add another commit on top of your existing commit if you want. This will create a second change (and thus another review). Gerrit will show the relationship between these two changes as a so-called relation chain. This also means that your second change can only be submitted after the first was successfully merged. In many basic use cases, this situation is however not what you want.

# Push

## Upload a change to Gerrit

This is done by pushing a commit to Gerrit. The commit must be pushed to a ref:

$ git commit --amend --no-edit

$ git push origin HEAD:refs/for/<branch\_name>

Gerrit accepts pushes to refs/for/<branch\_name> for every branch that it tracks.

## Bypass code review

The above magic syntax allows Gerrit to differentiate commits pushed for review from commits pushed directly into the repository, bypassing code review.

That's mean you can ignore code review if you want by running:

$ git push origin HEAD:<branch\_name>

**Notes**:

* If you bypass code review, you always need to merge/rebase manually if the tip of the destination branch has moved. Remember that if you choose to not work with code review because you think it’s easier to avoid the additional complexity of the review workflow, it might actually not be easier.

## What happens behind a push?

When a commit is pushed for review, Gerrit stores it in a staging area which is a branch in the refs/changes/ namespace.

The ref is used to map the Gerrit concept of "Pushing for Review" to the Git. For the Git client, it looks like every push goes to the same branch. But in fact, **for each commit pushed to this ref, Gerrit creates a new branch** under the refs/changes/ namespace.

Understanding the format of this ref is not required, but it is explained below.

A change ref has the format refs/changes/X/Y/Z where X is the last two digits of the change number, Y is the entire change number, and Z is the patch set. For example, if the change number is [263270](https://gerrit-review.googlesource.com/c/gerrit/+/263270), the ref would be refs/changes/70/263270/2 for the second patch set.

# Pull

If needed this commit can be fetched from Gerrit by using the fetch command from the [download commands](https://gerrit-review.googlesource.com/Documentation/user-review-ui.html#download) in the change screen.

# Merge

## Submit a change

This occurs when the change can be **merged to the target branch**. Submit requires the Submit access right and is done on the change screen by clicking on the *Submit* button.

In order to be submittable, **changes must first be approved by**[**voting on the review labels**](https://gerrit-review.googlesource.com/Documentation/user-review-ui.html#vote). By default a change can only be submitted if it has a vote with the highest value on each review label and no vote with the lowest value (veto vote).

**Notes**:

* Submitting a change may fail with conflicts. In this case, you need to [rebase the change](#_Rebase) locally, resolve the conflicts and upload the commit with the conflict resolution as new patch set.
* If a change cannot be merged due to path conflicts this is highlighted on the change screen by a bold red Cannot Merge label.

**Tips**:

* Projects can configure [custom labels](https://gerrit-review.googlesource.com/Documentation/intro-project-owner.html#labels) and [custom submit rules](https://gerrit-review.googlesource.com/Documentation/intro-project-owner.html#submit-rules) to control when a change becomes submittable.
* How the change is merged to the target branch is controlled by the [submit type](https://gerrit-review.googlesource.com/Documentation/config-project-config.html#submit-type) which can be [configured on project-level](https://gerrit-review.googlesource.com/Documentation/intro-project-owner.html#submit-type).

How is this different from GitHub?

After resolving, with GitHub, you end up with another merge commit and push it to the server.

With Gerrit, you resolve the conflict through rebasing your commit/change results in another amended version of that same commit. Then, you upload it to Gerrit, resulting in a new patch set. This new patch set will usually require another round of reviewer votes, as Gerrit will not copy votes from a previous patch set by default.

## [TIP] Auto merge

The project owner may enable [auto-merge on push](https://gerrit-review.googlesource.com/Documentation/user-upload.html#auto_merge) to benefit from the automatic merge/rebase on server side while pushing directly into the repository.

# Rebase Changes

While your change is in review, the HEAD of the target branch can evolve. In this case, the change should be rebased onto the new HEAD.

**If there are no conflicts, the rebase can be done directly from the change screen. Otherwise, it must be done locally**.

## Rebase a change from Gerrit web

TODO

## Rebase a change locally

// update the remote tracking branches

$ git fetch

// fetch and checkout the change (checkout command copied from change screen)

$ git fetch https://gerrithost/myProject refs/changes/74/67374/2 && git checkout FETCH\_HEAD

// do the rebase

$ git rebase origin/master

// resolve conflicts if needed and stage the conflict resolution

...

$ git add <path-of-file-with-conflicts-resolved>

// continue the rebase

$ git rebase --continue

// push the commit with the conflict resolution as new patch set

$ git push origin HEAD:refs/for/master

**Notes**:

* Rebasing change **creates a new patch set**.
* Never rebase commits that are already part of a central branch.

# Search Changes

## How?

<https://gerrit-review.googlesource.com/Documentation/user-search.html>

## Tips

You can make your search easier by following additional steps:

### Using Topics

**Why?**

Changes can be grouped by topics. This is useful because it **allows you to easily find related changes** by using the [topic search operator](https://gerrit-review.googlesource.com/Documentation/user-search.html#topic). Also, **changes with the same topic are displayed** so that you can easily navigate between them.

Often changes that together implement **a feature** or **a user story** are group by a topic.

**How?**

Assigning a topic to a change can be done by one of following ways:

* Using the change screen:



* Using CL:

$ git push origin HEAD:refs/for/master%topic=multi-master

// this is the same as:

$ git push origin HEAD:refs/heads/master -o topic=multi-master

**Tips**:

* Gerrit may be [configured](https://gerrit-review.googlesource.com/Documentation/config-gerrit.html#change.submitWholeTopic) to submit all changes in a topic together with a single click, even when topics span multiple projects.
* For more information about using topics, see the user guide: [Submitting Changes Across Repositories by using Topics](https://gerrit-review.googlesource.com/Documentation/cross-repository-changes.html).

### Using Hashtags

**Why?**

Similar to topics, hashtags can be **used to group related changes together**, and to search using the [hashtag:](https://gerrit-review.googlesource.com/Documentation/user-search.html#hashtag) operator. Unlike topics, a change can have multiple hashtags, and they are only used for informational grouping; changes with the same hashtags are not necessarily submitted together.

**How?**

Set hashtags to a change can be done by one of following ways:

* Using the change screen:



* Using CL:

$ git push origin HEAD:refs/for/master%t=stable-bugfix

// this is the same as:

$ git push origin HEAD:refs/heads/master -o t=stable-bugfix

# Mark Changes

## Work-in-Progress Changes

### What?

Work-in-Progress (WIP) changes are visible to anyone, but **they don't notify or require an action from a reviewer**.

When you mark a change as *Work-in-Progress*:

* Reviewers are not notified for most operations, such as adding or removing, posting comments, etc.
* The change does not show in reviewers' dashboards.

### When?

WIP changes are useful when:

* **You have implemented only part of a change**, but want to push your change to the server to run tests or perform other actions before requesting reviewer feedback.
* During a review, you realize you need to rework your change, and you want to stop notifying reviewers of the change until you finish your update.

### How to set WIP?

To set the status of a change to *Work-in-Progress*, you can:

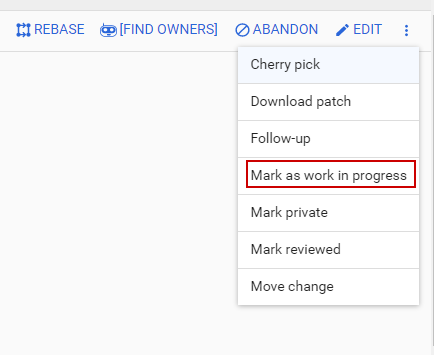
**With CL**

Append %wip to your push request:

$ git push origin HEAD:refs/for/master%wip

**With web UI**

Click *More > Mark as work in progress* on the change screen. The Change screen updates with a yellow header, indicating that the change is in a Work-in-Progress state.



### How to change WIP to Ready?

**With CL**

To mark the change as ready for review, append %ready to your push request.

$ git push origin HEAD:refs/for/master%ready

**With web UI**

You choose one of following options:

1. From the change screen, click button *Start Review*. This will open the reply-modal and allow you to add reviewers and/or CC before you start review.
2. Click button *Mark As Active* to change the state from WIP to ready, without opening the reply-modal.

## Private Changes

### What?

Private changes are changes that are only visible to their owners, reviewers and users with the View Private Changes access right.

### When?

Private changes are useful when:

* You want a set of collaborators to review the change before formal review starts. By creating a Private change and adding only a selected few as reviewers, you can control who can see the change and get a first opinion before opening up for all reviewers.
* You want to check what the change looks like before formal review starts. By marking the change private without reviewers, nobody can prematurely comment on your changes.
* You want to use Gerrit to sync data between different devices. By creating a private throwaway change without reviewers, you can push from one device, and fetch to another device.

**Notes**:

* Do **not** use private changes for making security fixes (see [pitfalls](https://gerrit-review.googlesource.com/Documentation/intro-user.html#private-changes-pitfalls) ). How to make security fixes is explained [here](https://gerrit-review.googlesource.com/Documentation/intro-user.html#security-fixes).

### How?

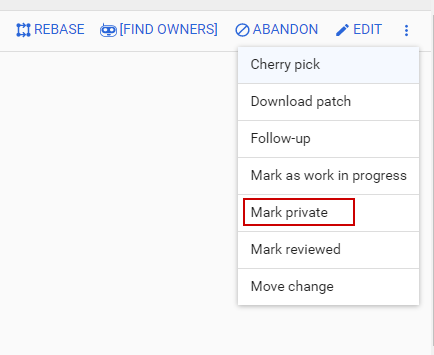
**With CL**

Push a change with the %private option.

$ git push origin HEAD:refs/for/master%private

**With web UI**

Click *More > Mark private* on the change screen.

****

**Notes**:

* The change will remain private on subsequent pushes until you specify the remove-private option.
* When pushing a private change with a commit that is authored by another user, the other user will not be automatically added as a reviewer and must be explicitly added.

# Access Control

<https://vlab.noaa.gov/code-review/Documentation/access-control.html>

# Adding Reviewers

In the change screen, **reviewers can be added explicitly to a change**. The added reviewer will then be **notified by email** about the review request.

Normally, it's not needed to add reviewers on every change, but you rather rely on the project team to watch their project and to process the incoming changes by importance, interest, time etc.

There are also [plugins which can add reviewers automatically](https://gerrit-review.googlesource.com/admin/repos/plugins/reviewers,general) (e.g. by configuration or based on git blame annotations). If this functionality is required it should be discussed with the project owners and the Gerrit administrators.

# Preferences

<https://gerrit-review.googlesource.com/Documentation/intro-user.html#preferences>