Gerrit

1. Definition

A. Gerrit is a web-based tool that is used for code review.

B. Its main features are side-by-side difference viewing and inline commenting. These make code reviews a quick and simple task.

C. It is used together with the Git version control system.

D. Gerrit allows authorized contributors to merge changes to the Git repository after reviews are done.

E. Contributors can get their code reviewed with little effort, and get their changes quickly through the system.

F. Gerrit is a Git server that provides access control for the hosted Git repositories and a web front-end for doing code review.

G. Code review is a core functionality of Gerrit, but still, it is optional and teams can decide to work without code review.

H. Code review

   a. With Gerrit *Code Review* means to review every commit before it is accepted into the codebase.

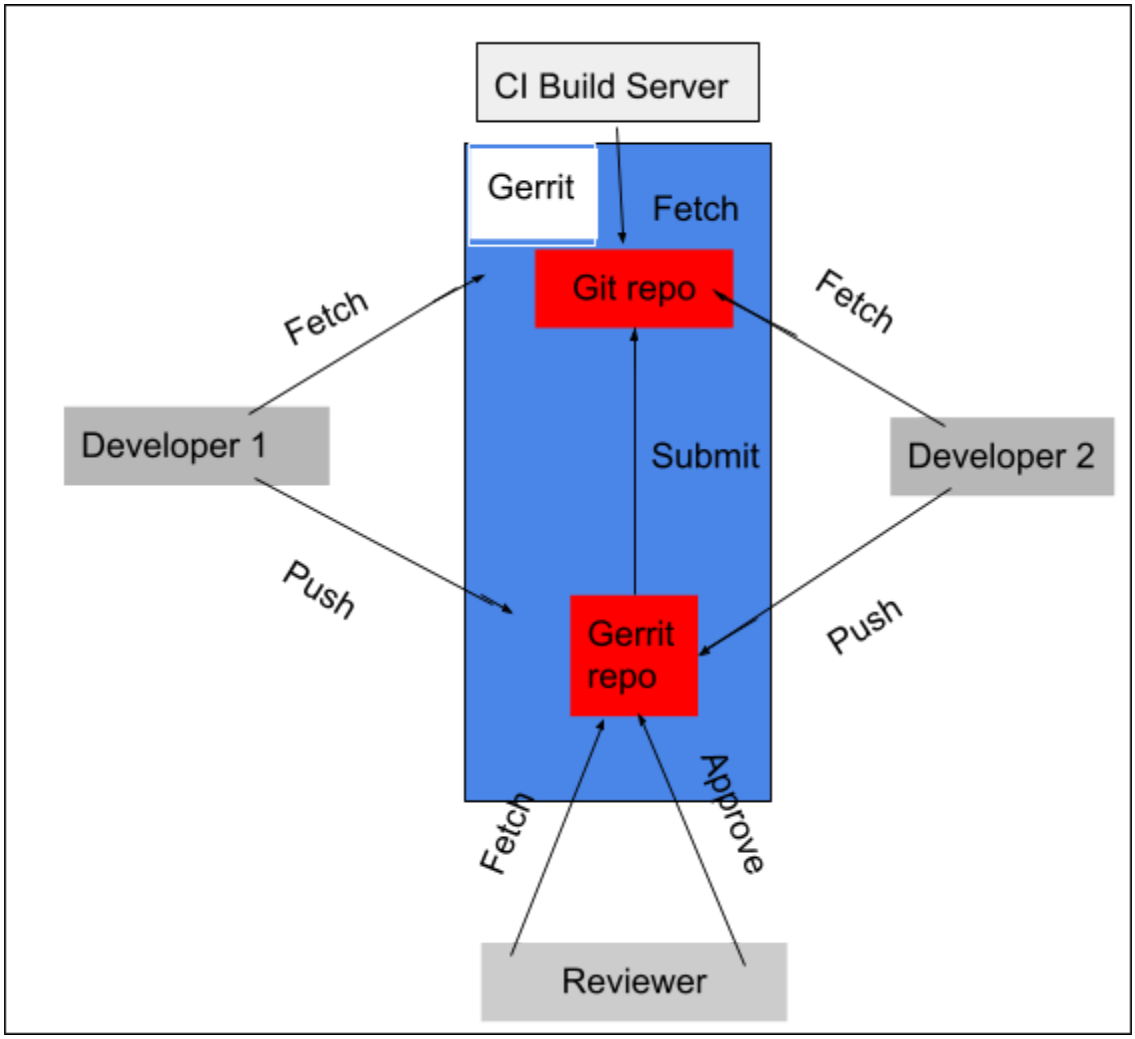
   b. The author of a code modification uploads a commit as a change to Gerrit.

   c. In Gerrit, each change is stored in a staging area where it can be checked and reviewed.

   d. Only when it is approved and submitted it gets applied to the code base. If there is feedback on a change, the author can improve the code modification by amending the commit and uploading the new commit as a new patch set.

   e. This way a change is improved iteratively and it is applied to the code base only when it is ready.

2. Gerrit workflow



3. Gerrit Installation

1. Make an account on Wikimedia developer account
2. https://www.mediawiki.org/wiki/Developer_account
3. The same username and password will be used to log into Gerrit below.
4. Install git
    a. On Linux

        ```
        sudo apt install git
        ```

b. On windows
https://phoenixnap.com/kb/how-to-install-git-windows

4. Configure Git

Git tracks who makes each commit by checking the user's name and email. In addition, this info is used to associate your commits with your Gerrit account.

Enter the two commands below to set your username and email address. Replace gerrituser with your own Gerrit username and replace gerrituser@example.com with your own email address:

```
git config --global user.email "gerrituser@example.com"
```

```
git config --global user.name "gerrituser"
```

To see your current configuration variables which control how Git behaves, use

```
git config -l
```

5. SSH key Setting in Gerrit

For making a connection between your computer and Gerrit we establish an SSH key
For checking if the key is already present in your server

Run this command in a terminal:

```
ls ~/.ssh
```

The command will list the files that are in the (hidden) .ssh directory. If the directory already exists on your system and if the output lists a file called id_rsa.pub then no need to generate any a new ssh key

a. Generate a new SSH key

To generate a new SSH key, enter the command below and replace gerrituser@example.com with your own email address. We want the default settings so when asked to enter a file in which to save the key, just press enter.

```
ssh-keygen -t rsa -C "gerrituser@example.com"
```

The `ssh-keygen` command will create 2 files in `~/.ssh` directory:

`~/.ssh/id_rsa` : your private SSH key (for identification)

`~/.ssh/id_rsa.pub` : your public SSH key

b. Copy your SSH Public key

Get the content of your public key file (e.g. `id_rsa.pub`) to copy it to your clipboard:

- On Linux, run `cat ~/.ssh/id_rsa.pub` and manually copy the output to the clipboard.
- On Windows, you can open Git GUI, go to Help □ Show Key, and then press "Copy To Clipboard" to copy your public key to your clipboard.

c. Add SSH Public key to your Gerrit account

- Log in to the web interface for Gerrit. The username and password for your Gerrit are the same as for your Wikimedia Developer account.
- Click on your username in the top right corner, then choose "Settings".
- Click "SSH Keys" in the menu on the left.

- Paste your SSH Public Key into the corresponding field and click "Add".

d. Add SSH Private key to use with Git

Start the Git Bash command line.

- Get ssh-agent running using `eval `ssh-agent``
- Add your private key to the agent. If you followed the steps above and your key has the default name id_rsa, then the command is: `ssh-add ~/.ssh/id_rsa`
- Add your private key to the agent. If you followed the steps above and your key has the default name `id_rsa`, then the command is: `ssh-add ~/.ssh/id_rsa`
- Be paranoid and compare that the "RSA key fingerprint" is the same as the SSH fingerprint for gerrit.wikimedia.org:29418. If it is the same, answer "Yes" to "Are you sure you want to continue connecting?". Then enter the passphrase for your key.
- You should get a message "Welcome to Gerrit Code Review". The last line should show "Connection to gerrit.wikimedia.org closed."
- If you run into problems, use `ssh -p 29418 -v gerrituser@gerrit.wikimedia.org`
- (replace Gerrit user by your username). The -v will provide verbose output to help find problems. Then read Gerrit Troubleshooting.
- Be paranoid and compare that the "RSA key fingerprint" is the same as the SSH fingerprint for gerrit.wikimedia.org:29418. If it is

the same, answer "Yes" to "Are you sure you want to continue connecting?". Then enter the passphrase for your key.

- You should get a message "Welcome to Gerrit Code Review". The last line should show "Connection to gerrit.wikimedia.org closed."
- If you run into problems, use `ssh -p 29418 -v gerrituser@gerrit.wikimedia.org` (replace gerrituser with your username). The `-v` will provide verbose output to help find problems. Then read Gerrit Troubleshooting.

```
gerrituser@machine:/mw/sandbox$ ssh -p 29418
gerrituser@gerrit.wikimedia.org
The authenticity of host '[gerrit.wikimedia.org]:29418
([208.80.154.85]:29418)' can't be established.
RSA key fingerprint is
dc:e9:68:7b:99:1b:27:d0:f9:fd:ce:6a:2e:bf:92:e1.
Are you sure you want to continue connecting (yes/no)? yes
Warning permanently added '[gerrit.wikimedia.org]:29418
([208.80.154.85]:29418)' (RSA) to the list of known hosts.
Enter passphrase for key '/home/gerrituser/.ssh/id_rsa':

  ****    Welcome to Gerrit Code Review    ****

  Hi gerrituser, you have successfully connected over SSH.

  Unfortunately, interactive shells are disabled.
  To clone a hosted Git repository, use:

  git clone
ssh://gerrituser@gerrit.wikimedia.org:29418/REPOSITORY_NAME.
git

Connection to gerrit.wikimedia.org closed.

 gerrituser@machine:/mw/sandbox$
```

6. Practice

a. Cloning

Clone the sandbox repo using the following command on git bash cmd

`gerrituser@gerrit.wikimedia.org:29418/sandbox`

Change Gerrit user to your username

b. Preparation for Working with gerrit

Gerrit requires that your commit message must have a "change ID". They look like `Change-Id: I`bd3be19ed1a23c8638144b4a1d32f544ca1b5f97 starting with an *I* (capital i). Each time you amend a commit to improve an existing patch in Gerrit, this change ID stays the same, so Gerrit understands it as a new "patch set" to address the same code change.

There's a git add-on called git-review that adds a Change-ID line to your commits. Using git-review is recommended. It makes it easier to configure your Git clone, submit a change, or to fetch an existing one.

Installing git-review

Note that Wikimedia Gerrit requires git-review version 1.27 or later.

Linux

```
sudo apt-get install git-review
```

If git-review has not been packaged by your distribution, check git-review for other options such as installing git-review by using the pip Python package installer.

Windows

Please see git-review Windows.

Configuring git-review

Git's default remote hostname is "origin". This name is also used by Wikimedia projects. We need to tell git-review to use that host. Replace gerrituser with your Gerrit username:

```
git config --global gitreview.remote origin
git config --global gitreview.username gerrituser
```

Setting up git-review

After downloading ("cloning") a repository, you need to set it up for git-review. This will automatically happen the first time you try to submit a commit, but it's generally better to do it right after cloning. Make sure that you are in the directory of the project that you cloned (otherwise you will get an error "fatal: Not a git repository"). Then run this command:

```
git review -s --verbose
```

c. Submit a patch

Make sure that you cloned the code repository that you are interested in

Make sure that you are in the directory of the code repository (the command `pwd` tells you where exactly you are).

Update master

Make sure that your master branch (the branch created when you initially cloned the repository) is up to date:

```
git pull origin master
```

However, note that a few repositories use different terms (for example, the `operations/puppet` repository has a `production` instead of a `master` branch).

Create a branch

First, create a local branch for your new change. Replace BRANCHNAME below with a short but reasonably descriptive name (e.g. `T1234` if a

corresponding Phabricator task exists for your changes, `cleanup/some-thing`, or `badtitle-error`). Other people will also use this name to identify your branch.

```
git checkout -b BRANCHNAME origin/master
```

This will create a new branch (called BRANCHNAME) from the latest 'master' and check it out for you.

Make your changes

Make changes to your local code. Use your preferred text editor and modify a file. In the example below, we edit the file `README.md` and add a word.

Then close your text editor and check the changes you have made since the last commit, within the file(s) and within the directory:

```
git diff
```

git diff displays your changes in unified diff format: Removed lines have a minus (-) prefix and added lines have a plus (+) prefix. These changes are not yet "staged" (via git add) for the next commit.

Stage your changes for a commit

Run git status to decide which of your changes should become part of your commit. It will display a list of all file(s) that you have changed within the directory. At this point, the output will display "no changes added to commit" as the last line.

Use git add to make your changed file(s) become part of your next commit. In the example above we modified the file README.md, so the command would be:

git add README.md Any files you've changed that you have not passed to git add will be ignored when running git commit in the next step.

Commit your staged changes

Once you are happy with the list of changes added via `git add`, you can turn these changes into a commit in your local repository by using

`git commit`

You will then be asked in your text editor to add a descriptive summary for your commit. You must follow the Commit message guidelines. This is what other people will see when looking at the history of changes in the code repository.

Save the commit message and close your text editor. A summary (the commit ID, your subject line, the files and lines changed) will be displayed.

Push your commit to Gerrit

Uploading a change to Gerrit is done by pushing a commit to Gerrit. The commit must be pushed to a ref in the refs/for/ namespace which defines the target branch: refs/for/<target-branch>. The magic refs/for/ prefix allows Gerrit to differentiate commits that are pushed for review from commits that are pushed directly into the repository, bypassing code review. For the target branch, it is sufficient to specify the short name, e.g. master, but you can also specify the fully qualified branch name, e.g. refs/heads/master.

Push for Code Review

```
$ git commit
$ git push origin HEAD:refs/for/master

// this is the same as:
$ git commit
$ git push origin HEAD:refs/for/refs/heads/master
```

Push with bypassing Code Review

```
$ git commit
```

```
$ git push origin HEAD:master

// this is the same as:
$ git commit
$ git push origin HEAD:refs/heads/master
```

When a commit is pushed for review, Gerrit stores it in a staging area which is a branch in the special `refs/changes/` namespace. Understanding the format of this ref is not required for working with Gerrit, 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, the ref would be `refs/changes/70/263270/2` for the second patch set.

Using the change ref git clients can fetch the corresponding commit, e.g. for local verification.

Fetch Change

`$ git fetch https://gerrithost/myProject refs/changes/74/67374/2 && git checkout FETCH_HEAD`

The `refs/for/` prefix is used to map the Gerrit concept of "Pushing for Review" to the git protocol. For the git client, it looks like every push goes to the same branch, e.g. `refs/for/master` but in fact for each commit that is pushed to this ref Gerrit creates a new branch under the `refs/changes/` namespace. In addition, Gerrit creates an open change.

A change consists of a Change-Id, metadata (owner, project, target branch, etc.), one or more patch sets, comments, and votes. 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.

The Change-Id is important for Gerrit to know whether a commit that is pushed for code review should create a new change or whether it should create a new patch set for an existing change.

The Change-Id is an SHA-1 that is prefixed with an uppercase I. It is specified as a footer in the commit message (last paragraph):

```
Improve the foo widget by attaching a bar.

We want a bar, because it improves the foo by providing more
wizbangery to the dowhatimeanery.

Bug: #42
Change-Id: Ic8aaa0728a43936cd4c6e1ed590e01ba8f0fbf5b
Signed-off-by: A. U. Thor <author@example.com>
```

If a commit that has a Change-Id in its commit message is pushed for review, Gerrit checks if a change with this Change-Id already exists for this project and target branch and if yes, Gerrit creates a new patch set for this change. If not, a new change with the given Change-Id is created.

If a commit without Change-Id is pushed for review, Gerrit creates a new change and generates a Change-Id for it. Since in this case the Change-Id is not included in the commit message, it must be manually inserted when a new patch set should be uploaded. Most projects already require a 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.

Amending and rebasing a commit preserves the Change-Id so that the new commit automatically becomes a new patch set of the existing change when it is pushed for review.

Push new Patch Set

```
$ git commit --amend
$ git push origin HEAD:refs/for/master
```

Change-Ids are unique for a branch of a project. E.g. commits that fix the same issue in different branches should have the same Change-Id, which happens automatically if a commit is cherry-picked to another branch. This way you can search by the Change-Id in the Gerrit web UI to find a fix in all branches.

Change-Ids can be created automatically by installing the `commit-msg` hook as described in the Change-Id documentation.

Instead of manually installing the `commit-msg` hook for each git repository, you can copy it into the git template. Then it is automatically copied to every newly cloned repository.

d. View the Change / Next Steps

Open the link to your Gerrit changeset in a web browser.

Under "Files", after you clicked the down arrow at the very right of any file in the list, you can see a diff of your changes per file: The old lines are shown in red color and your new lines are shown in green color.

Gerrit's diff algorithm (jGit) is slightly different from git's default diff algorithm. The differences displayed by Gerrit might not look like the differences displayed by Git on your machine.

If your commit addresses a ticket in Phabricator, a comment will be automatically added in the Phabricator task if you followed the Commit message guidelines. If you did not, you could either fix your commit message (by creating an updated patchset), or manually add a comment on that Phabricator ticket which includes a link to your changeset in Gerrit.

e. Upload a new Patch Set

If there is feedback from code review and a change should be improved a new patch set with the reworked code should be uploaded.

This is done by amending the commit of the last patch set. If needed this commit can be fetched from Gerrit by using the fetch command from the download commands in the change screen.

It is important that the commit message contains the Change-Id of the change that should be updated as a footer (last paragraph). Normally the commit message already contains the correct Change-Id and the Change-Id is preserved when the commit is amended.

Push Patch Set

```
// 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

  // rework the change
  $ git add <path-of-reworked-file>
  ...
```

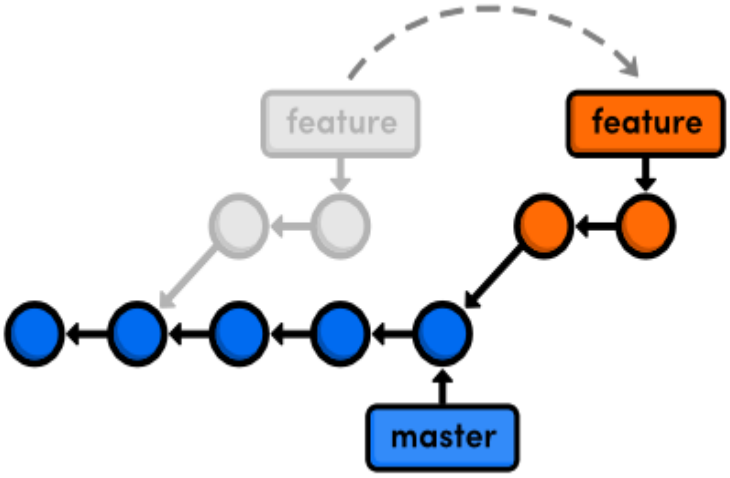
```
// amend commit
$ git commit --amend

// push patch set
$ git push origin HEAD:refs/for/master
```

`note:-`Never amend a commit that is already part of a central branch.

Pushing a new patch set triggers email notification to the reviewers.

f. Rebasing

- Rebasing is the process of moving or combining a sequence of commits to a new base commit.



- From a content perspective, rebasing is changing the base of your branch from one commit to another making it appear as if you'd created your branch from a different commit.

- Internally, Git accomplishes this by creating new commits and applying them to the specified base.
- It's very important to understand that even though the branch looks the same, it's composed of entirely new commits.
- you should never rebase commits once they've been pushed to a public repository.
- The rebase would replace the old commits with new ones and it would look like that part of your project history abruptly vanished.
- https://gist.github.com/nicholashagen/2855167

7. Basic flow

- go to Gerrit code
- Settings
- Copy command for cloning
- Go to terminal
- Git clone ssh..cloning link
- Go to file
- Make changes
- Come again on terminal
- Git status
- Git diff file_path //to see the changes
- Git add file_path //to add the file
- Git commit //to open one file to write a message
- Git push origin HEAD:ref/for/master //HEAD->is a pointer that point to the last commit
- If it will give failure
- So in output, you will get one hook command copy that, and run
- Now for commit again
- Commit -amend //to make a change in the last commit message instead of the new commit message
- again run the following command
- Git push origin HEAD:ref/for/master
- Now go to Gerrit and add reviewer and add reply message

- If you make changes in someone else code then gerrit->download->cherry pick
- If you want to just checkout in someone else code then gerrit->download->checkout

- Ssh:

SSH stands for *Secure Shell* or sometimes *Secure Socket Shell* protocol used for accessing network services securely from a remote computer. You can set the SSH keys to provide a reliable connection between the computer and Gerrit.

8. Links

https://www.mediawiki.org/wiki/Gerrit/Tutorial

https://gerrit-review.googlesource.com/Documentation/intro-user.html

https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase#:~:text=What%20is%20git%20rebase%3F,of%20a%20feature%20branching%20workflow.