Gerrit
Concepts and Workflows
(for Googlers: go/gerrit-explained)

Edwin Kempin
Google Munich
ekempin@google.com

This presentation is based on a [Git/Gerrit workshop] that was developed by SAP.
Credits go to sasa.zivkov@sap.com, matthias.sohn@sap.com and christian.halstrick@sap.com

PUBLIC - SHARED WITH GERRIT COMMUNITY

Target Audience

This presentation is for:

- New Gerrit users
- Advanced Gerrit users that want to consolidate their Gerrit knowledge

Required pre-knowledge:

- Good knowledge about Git (see presentation about Git - Concepts and Workflows)

Content

YES

- Gerrit terminology
- Gerrit concepts
- Gerrit workflows
- Upstream Gerrit (2.16, open source version)

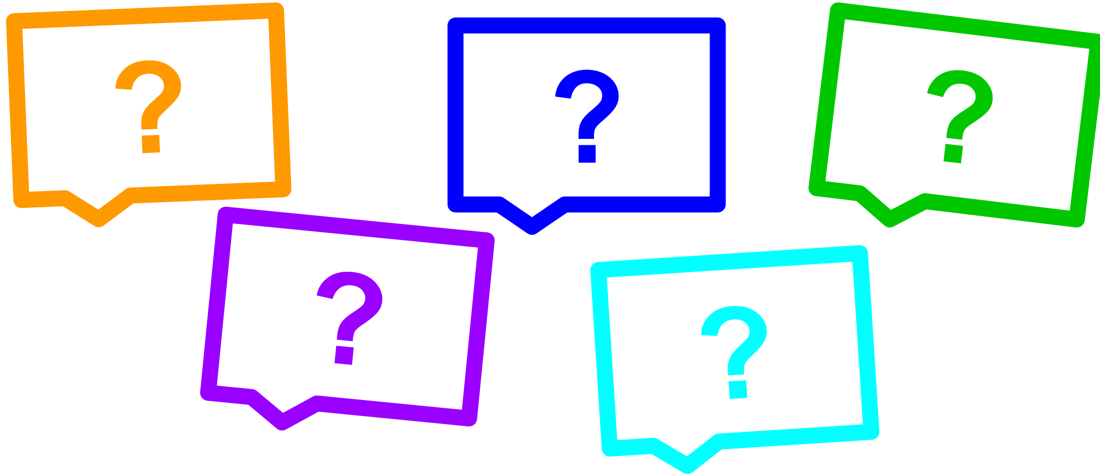
NO

- Git (covered by Git - Concepts and Workflows presentation)
- Gerrit internals (e.g. NoteDb storage format)
- Access Rights
- Repository administration
- Gerrit setup
- Plugins & tool integration
- REST API
- Change queries
- Google specifics

Agenda

- What’s Gerrit?
- What’s Code Review?
- Gerrit Concepts
  - Pushing for code review
  - Pushing new patch sets
- Submit, including submit strategies
- Rebasing Changes with Conflict Resolution
- Standard Workflow
- Working with Change Series
- Topics
- Working with Stable Branches
- How to make security fixes

Welcome



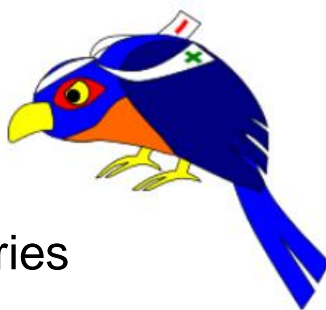
- Please ask questions immediately!
- To make the presentation more interactive you will also be asked questions :-)

Q: What is Gerrit?

Gerrit

Gerrit is a ***Git server*** that provides

- Code Review
- Access Control on the Git repositories



Code Review:

- Gerrit allows to ***review commits before*** they are integrated into a target branch.
- 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 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.

NOT Gerrit

Gerrit does **not** provide

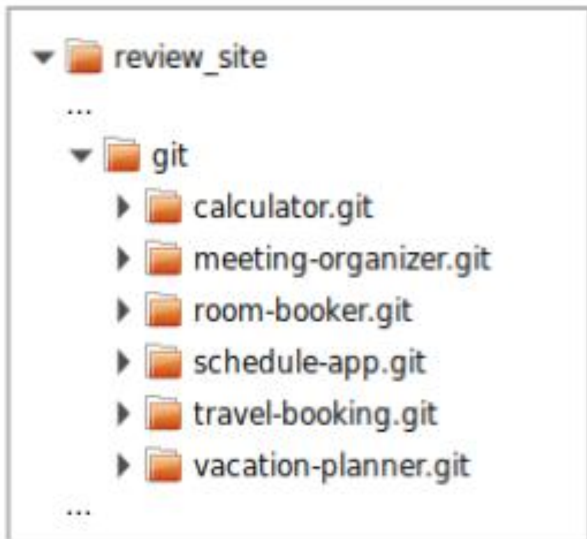
- Code Browsing
- Code Search
- Project Wiki
- Issue Tracking
- Continuous Builds
- Code Analyzers
- Style Checkers

Gerrit allows **integration** with third-party tools that provide additional functionalities (see list on the left).

Gerrit

Gerrit is built on top of Git

- it manages standard Git repositories and
- controls access and updates to them



In the Gerrit installation folder on the server (`review_site`) you can find a `git` folder that contains all Git repositories that are managed by Gerrit:

- these are ***bare repositories*** (this means they don't have a working dir)
- the repositories may be hierarchically structured in subfolders

At Google the repositories are not stored in the file system, but in a database.

Modern Code Review

- One developer writes code, another developer is asked to review that code
- A careful line-by-line critique
- Happens in a non-threatening context
- Goal is cooperation, not fault-finding
- Integral part of the coding process

Code Review Benefits

- Four eyes catch more bugs
  - Catching bugs early can save hours of debugging later
- Mentoring of new developers / contributors
  - Learn from mistakes without breaking stuff
- Establish trust relationships
  - Prepare for more delegation
- Good alternative to pair programming
  - Asynchronous and across locations
- Coding standards
  - Keep overall readability and code quality high

- Even self-reviewing your code in a review tool often let you spot bugs that you wouldn't have noticed otherwise.

Gerrit Concepts

- Gerrit “speaks” the ***Git protocol***
  ⇒ users only need a ***Git client***
  (there is no need to install a “Gerrit client”)
  ⇒ this means Gerrit must somehow map its concepts onto Git
- Gerrit allows to ***review commits before*** they are integrated into the target branch, but ***code review is optional***
- commits are pushed to Gerrit by using the `git push` command
- Git is a toolbox (“Swiss army knife”) which allows many workflows, Gerrit defines one workflow for working with Git

GitHub ***Pull Requests*** is another workflow for working with Git (not supported by Gerrit).

Q: Since code review is optional, how does Gerrit know if you push directly to Git or for code review?

Push for Code Review

Push for code review:

- Same command as pushing to Git with one Gerrit speciality: The target branch is prefixed with `refs/for/`
- `git push origin HEAD:refs/for/<branch-name>`
- Example:
  `git push origin HEAD:refs/for/master`
  same as
  `git push origin HEAD:refs/for/refs/heads/master`

Push directly to Git (bypassing code review):

- `git push origin HEAD:<branch-name>`
- Example:
  `git push origin HEAD:master`
  same as
  `git push origin HEAD:refs/heads/master`

Whether pushing directly to Git, and hence bypassing code review, is allowed can be controlled by access rights (direct push requires the `Push` permission on `refs/heads/*`).

Using `HEAD` in the push command means that the current commit/branch is pushed. Instead you can also specify a branch or SHA1.

Further details about the `git push` command are discussed in the Git - Concepts and Workflows presentation.

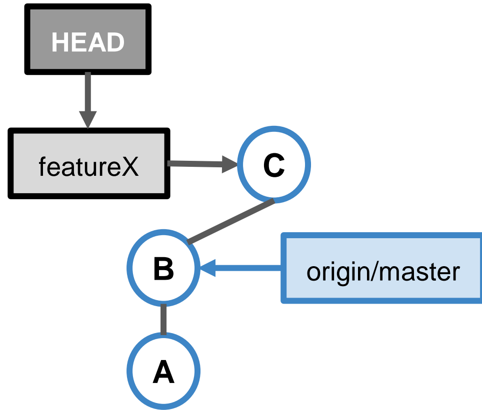
Push for Code Review

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

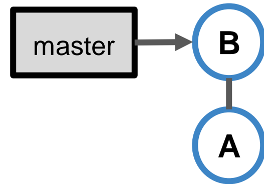
- From the Git clients perspective it looks like every push for code review goes to the same branch: `refs/for/master`
- However Gerrit tricks the Git client:
  - it creates a new ref for the commit(s) that are pushed
  - it creates or updates an open Gerrit change for each pushed commit

Push for Code Review - Case 1

local repository



remote repository



git push origin HEAD:refs/for/master

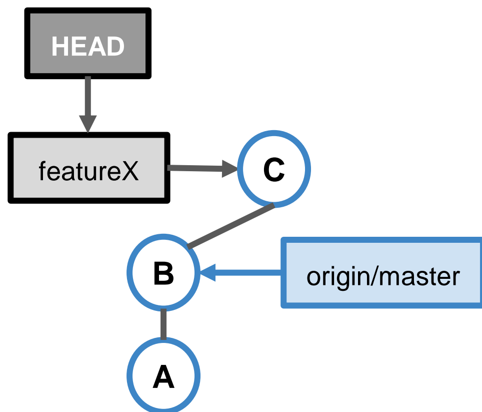
Situation:

- The remote repository was cloned, a local *featureX* branch was created and in this branch a commit ***C*** was created.

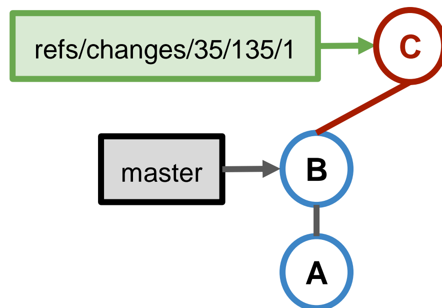
Q: What happens on push for code review?

Push for Code Review - Case 1

local repository



remote repository



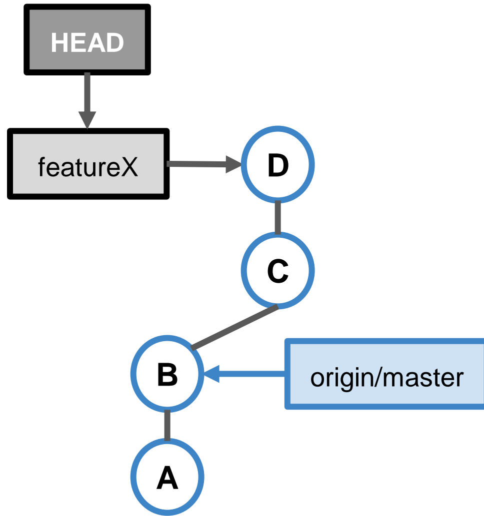

```
git push origin HEAD:refs/for/master
```

Push for Code Review:

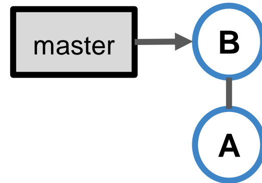
- pushes commit ***C*** to the remote repository
- Gerrit creates a new ***change ref*** that points to the new commit (`refs/changes/35/135/1`)
- Gerrit creates a new ***change*** object in its database
- does **not** update the `master` branch in the remote repository (the target branch is only updated once code review was done and the change is approved and submitted)

Push for Code Review - Case 2

local repository



remote repository




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

Situation:

- The remote repository was cloned, a local *featureX* branch was created and in this branch two commits, ***C*** and ***D***, were created.

Q: Which commits get pushed?

Push for Code Review - Case 2

local repository

HEAD
featureX
D
C
B
origin/master
A

remote repository

refs/changes/36/136/1
D
depends on
refs/changes/35/135/1
C
master
B
A

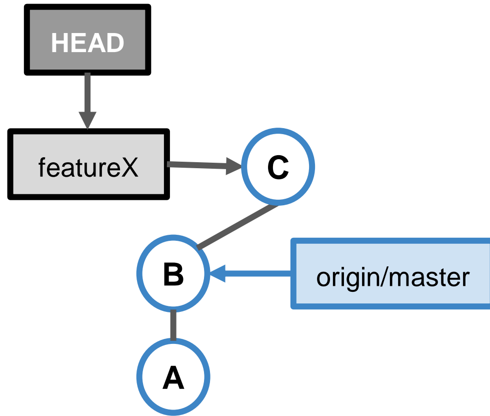
```
git push origin HEAD:refs/for/master
```

Push for Code Review:

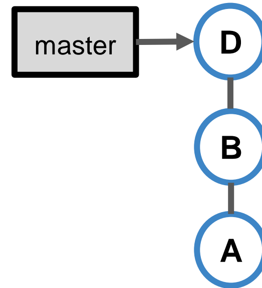
- pushes **all** commits which are reachable from the pushed commit and which are not available in the remote repository
- for each pushed commit Gerrit creates a change ref and a Gerrit change in its database
- The change for commit ***D*** depends on the change for commit ***C*** (since commit ***D*** depends on commit ***C***).

Push for Code Review - Case 3

local repository



remote repository




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

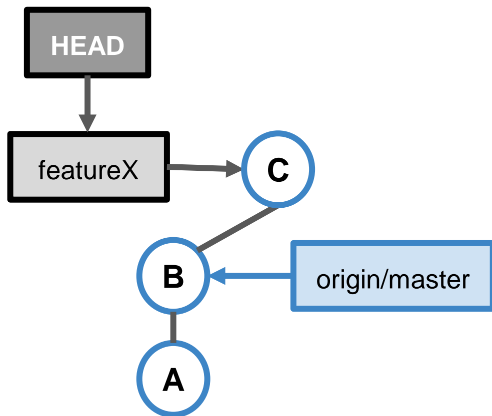
Situation:

- The remote repository was cloned, a local *featureX* branch was created and in this branch a commit ***C*** was created. In the meantime the remote branch *master* was updated to a commit ***D***.

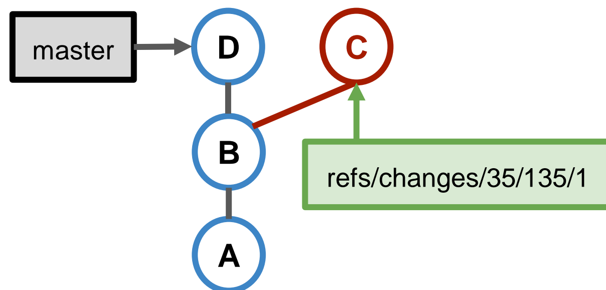
Q: What happens on push for code review?

Push for Code Review - Case 3

local repository



remote repository

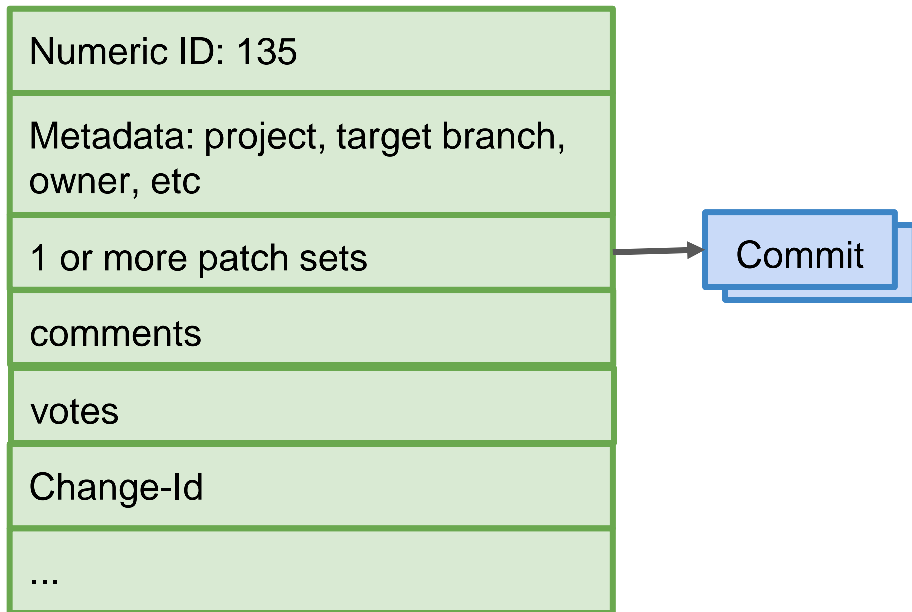


git push origin HEAD:refs/for/master

The push succeeds:

- Gerrit accepts commit ***C*** and creates a new change for it.
- The push succeeds even if commit ***C*** and ***D*** would be conflicting.
- If the push would have been done directly to Git this push would have failed since `master` cannot be fast-forwarded to the pushed commit.
- Submitting the change may or may not succeed (depends on the submit strategy which is explained later).

Change



- the ***numeric ID*** uniquely identifies a change on a Gerrit server
- the ***change owner*** is the user that uploaded the change (can differ from committer and commit author)
- ***patch sets*** correspond to Git commits (more about patch sets later)
- ***Change-Id*** (unique per repository and branch, explained later)

Review and Vote

Changes can be inspected in the Gerrit WebUI:

- The ***change screen*** shows you all information about a change, including which files have been changed.

Review and Vote

Changes can be inspected in the Gerrit WebUI.

- The ***change screen*** shows you all information about a change, including which files have been changed.
- For each modified file you can review the ***file diff*** and comment inline on it, which creates ***unpublished draft comments***. You can also reply to existing comments.

refs/changes/92/208892/1

C

master

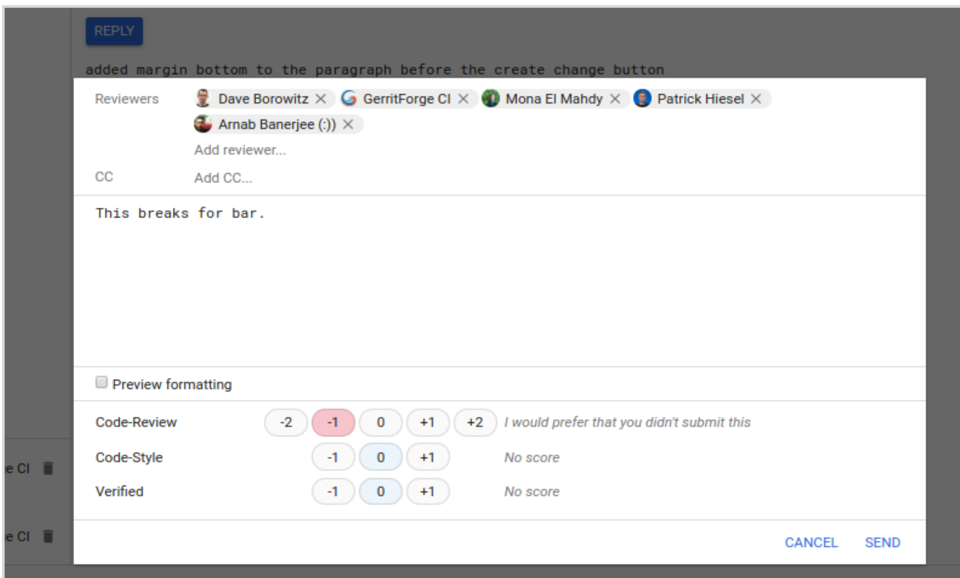
B

A

Compared versions:

- left side: commit ***B*** (***base/parent version***)
- right side: commit ***C***

Review and Vote



Changes can be inspected in the Gerrit WebUI.

- The ***change screen*** shows you all information about a change, including which files have been changed.
- For each modified file you can review the ***file diff*** and comment inline on it, which creates ***unpublished draft comments***. You can also reply to existing comments.
- The comments are published by ***replying*** on the change. The reply can include a general ***change message*** and you can give a ***voting*** on the change.

Voting

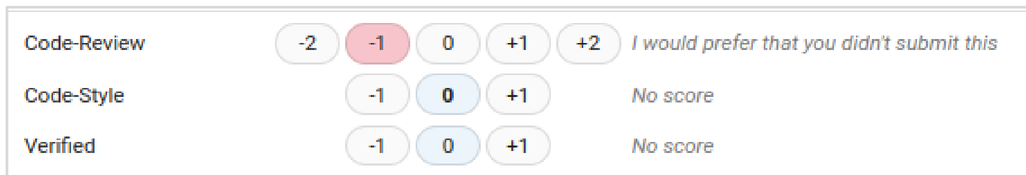
Code-Review	-2	-1	0	+1	+2	I would prefer that you didn't submit this
Code-Style		-1	0	+1		No score
Verified		-1	0	+1		No score

Voting is done on ***review labels***:

- Which ***review labels*** and ***voting values*** are available can be configured per repository, by default there is only the `Code-Review` label.
- Usually a change requires an ***approval*** (highest possible vote) for each label in order to become submittable.
- ***Veto votes*** (lowest possible value) block the submit of a change.
- ***Access rights*** control which user is allowed to vote on which review label and with which values
- Votings on some labels may be done automatically by bots (e.g. voting on the `Verified` label is often done by CI servers depending on the build and test results)

Q: What can the change owner do if a negative vote is received?

Voting



- Rework the change and upload a new version of the change.
  OR
- ***Abandon*** the change.

Abandoning a change means that the modifications are discarded and the change doesn't get submitted. Abandoned changes are still accessible and can be ***restored*** if needed.

Q: When you push a commit for code review how does Gerrit know if you push a new change or a new version of an existing change?

Change-Id

- ***Change-Id***:
  ID of a ***change*** that is set as footer in the commit message.
- Automatically generated and inserted on commit by a commit hook.
- If a commit is pushed that contains a `Change-Id` in the commit message Gerrit checks if a change with this `Change-Id` already exists.
  - If yes, this change is updated.
  - If not, a new change with that `Change-Id` is created.

The ***Gerrit commit-msg hook*** that generates and inserts ***Change-Ids*** on `git commit` must be installed once in a repository after it was cloned:

- The clone command that is offered by Gerrit in the WebUI includes the command to install the `commit-msg` hook.

Change-Id

```
First line is the subject, should be shorter than 70 chars

Separate the body from the subject by an empty line. The
commit message should describe why you are doing the
change. That's what typically helps best to understand
what the change is about. The details of what you changed
are visible from the file diffs.

The body can have as many paragraphs as you want. Lines
shouldn't exceed 80 chars. This helps command line tools
to render it nicely. Paragraphs are separated by empty
lines.

Change-Id: I3eefa6661010058d3684b2983f5b38bf3233d0f7
Bug: Issue 123
```

`Change-Id`:

- Format: 'I' + SHA1
- To be recognized by Gerrit the `Change-Id` must be contained in the **last paragraph** of the commit message (as all Git footers)

Q: What is a patch set?

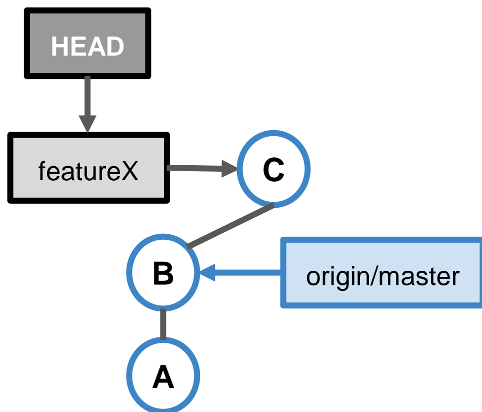
Patch Set

- ***Patch Set***:
  A version of a ***change***
- Correlates to a ***Git commit***.
- A change contains one or more patch sets.
- Each new patch set **replaces** the previous patch set, only **the latest patch set is relevant**.

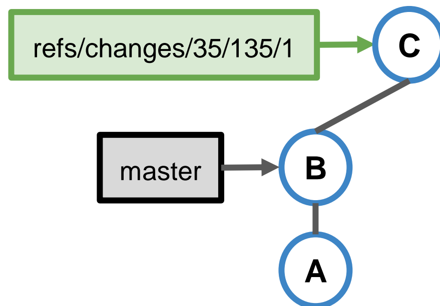
Often the term ***revision*** is used as synonym for ***patch set,*** or the Git commit of a patch set.

Push new Patch Set

local repository



remote repository



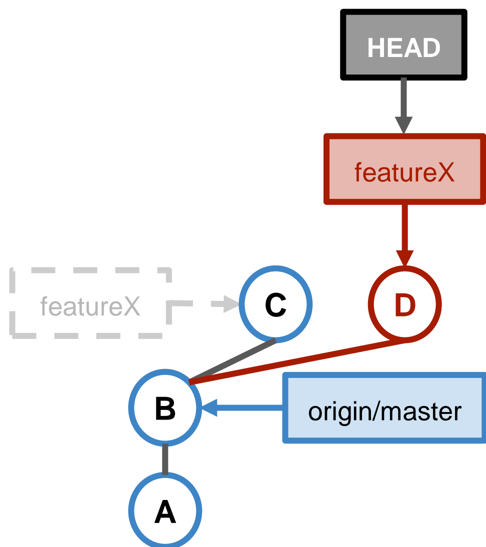
Situation:

- In the local `featureX` branch a commit *C* was done that was pushed for code review. During code review an issue was detected and the change should be reworked. The user has already checked out the `featureX` branch.

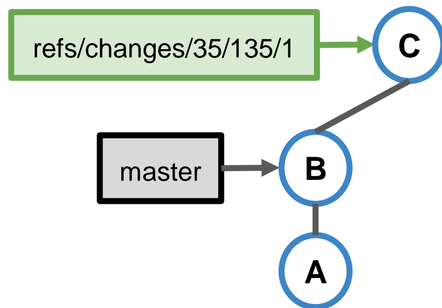
Q: How is a new a patch set created?

Push new Patch Set

local repository



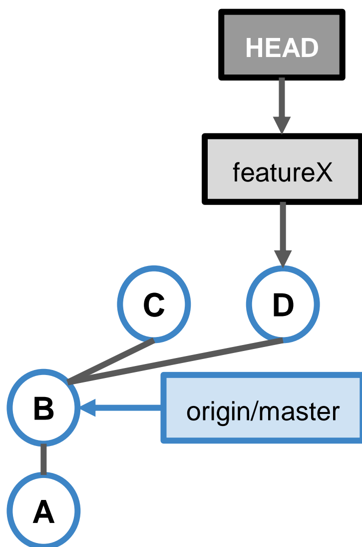
remote repository

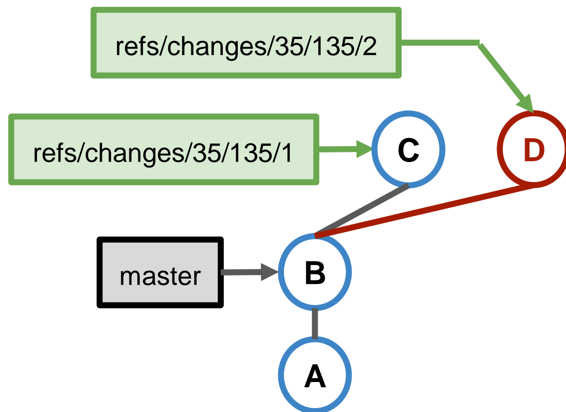


The user fixes the code and create a new commit by using *git commit --amend*:

- A new commit ***D*** is created that is a sibling of the old commit ***C***.
- The commit message, including the *Change-Id*, is preserved.

Push new Patch Set



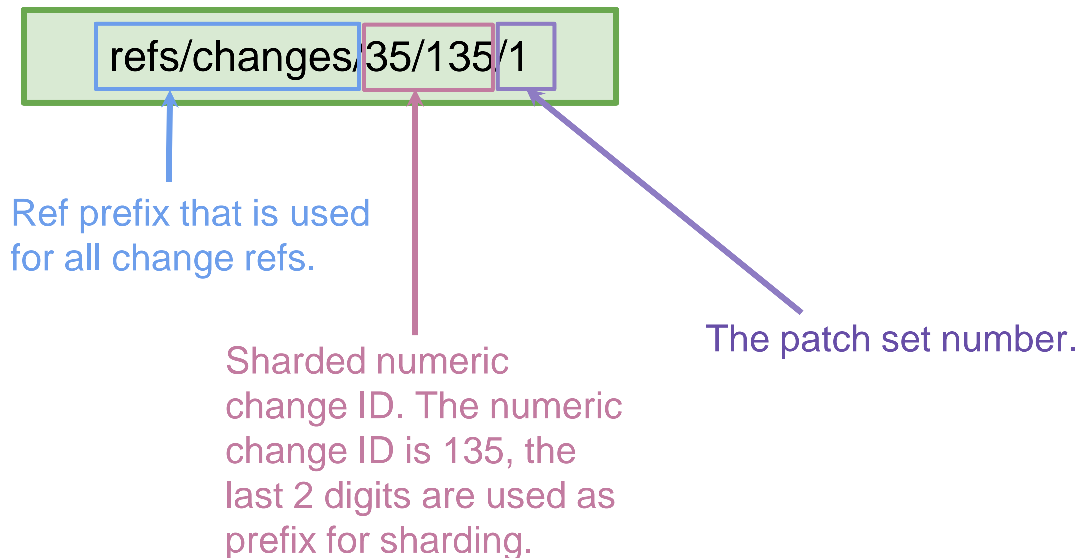


`git push origin HEAD:refs/for/master`

The new commit ***D*** is pushed for code review:

- Gerrit inspects commit ***D*** and finds the `Change-Id` in its commit message.
- Gerrit checks if for the target branch a change with that `Change-Id` already exists.
- Since a change with this `Change-Id` already exist Gerrit accepts commit ***D*** as a new patch set for this change and creates a new change ref for the second patch.
- The new patch set **replaces** the old patch set.
- The latest patch set on a change is called ***current patch set***.

Change ref



- All ***change refs*** share the same *`refs/changes/`* namespace. Refs in this namespace are not automatically fetched on *`git clone`* and *`git fetch`*.
- Can be fetched on need. Gerrit offers the fetch command on the change screen so that patch sets can be easily downloaded (e.g. to amend them and create a new patch set).

Review of new Patch Set

Patch Set Selector:

- Reviewers can see the full diff by comparing the new patch set against base version.

master

B

A

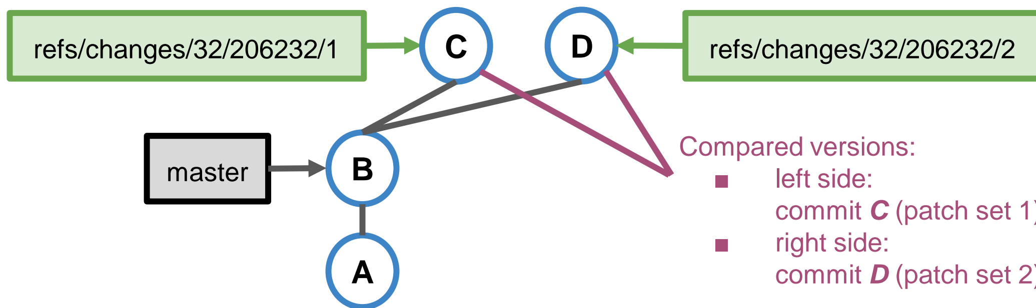
Compared versions:

- left side:
  commit ***B***
  (***base/parent version***)
- right side:
  commit ***D*** (patch set 2)

Comparing Patch Sets

Patch Set Selector:

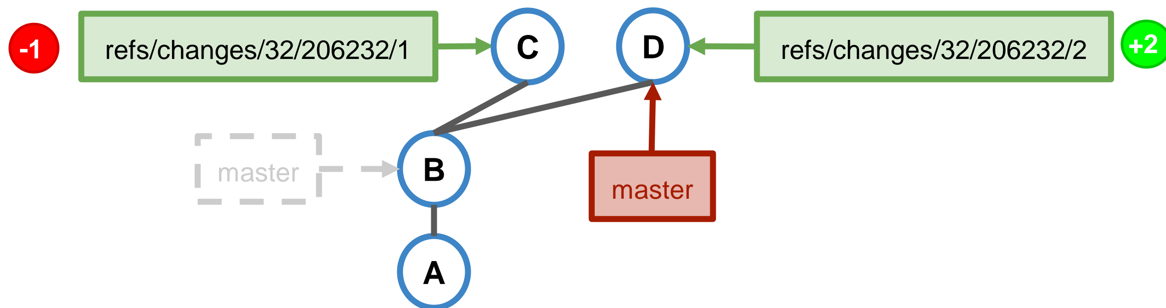
- Users that have previously reviewed the old patch set likely want to see what has changed with the new patch set. They can do so by comparing the old and new patch set.



Submit

Pre-conditions for submit:

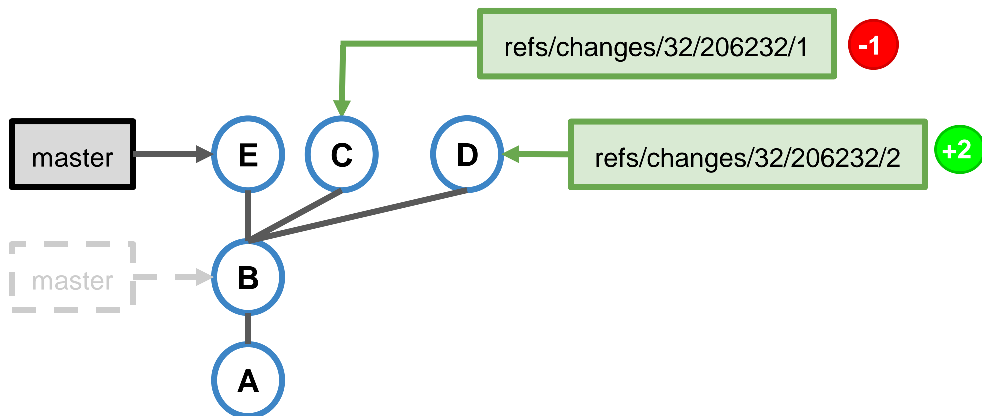
- The change has an ***approval*** (highest possible vote) for each ***review label***.
- None of the ***review labels*** has a ***veto vote*** (lowest possible vote)
- The change doesn't depend on other changes that are non-submittable.
- The user is allowed to submit.



Submit integrates a change into its target branch (more precisely integrates the current patch set into the target branch):

- The `master` branch is ***fast-forwarded*** to the commit that represents the current patch set.

Submit



Situation:

- A change has two patch sets (commit ***C*** and commit ***D***) which are both based on commit ***B***. The current patch set (commit ***D***) was approved and is submittable. In the meantime the `master` branch was updated to commit ***E***. Fast-forwarding `master` to the current patch set is not possible.

Q: What happens on submit if fast-forwarding the target branch is not possible?

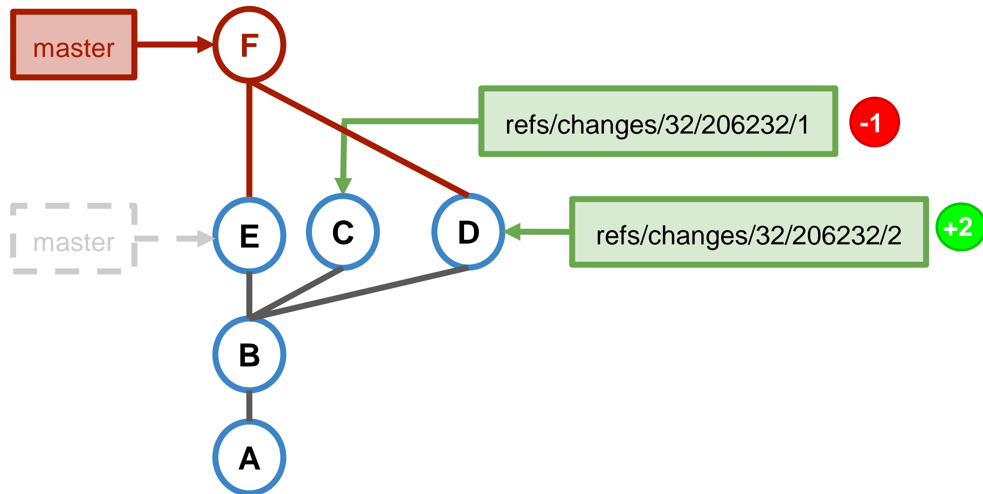
Submit Type / Submit Strategy

The behaviour on submit is configurable per repository:

- ***Submit Type*** / ***Submit Strategy***:
  - *FAST_FORWARD_ONLY*:
    Submit fails if fast-forward is not possible.
  - *MERGE_IF_NECESSARY*:
    If fast-forward is not possible, a merge commit is created.
  - *REBASE_IF_NECESSARY*:
    If fast-forward is not possible, the current patch set is automatically rebased (creates a new patch set which is submitted).
  - *MERGE_ALWAYS*:
    A merge commit is always created, even if fast-forward is possible.
  - *REBASE_ALWAYS*:
    The current patch set is always rebased, even if fast-forward is possible. For all rebased commits some additional footers will be added (`Reviewed-On`, `Reviewed-By`, `Tested-By`).
  - *CHERRY_PICK*:
    The change is cherry-picked. This ignores change dependencies. For all cherry-picked commits some additional footers will be added (`Reviewed-On`, `Reviewed-By`, `Tested-By`).
- *Allow content merges*:
  - whether Gerrit should do a content merge if the same files have been touched

- Recommended setting:
  - Submit type: *MERGE_IF_NECESSARY* or *REBASE_IF_NECESSARY*
  - *Allow content merges: true*
- Safest setting:
  - *FAST_FORWARD_ONLY*
  - With all other submit types there is a possibility that the submitted commit is broken.
  - Requires users to rebase changes whenever any change is submitted
  - Only feasible if there is a low frequency of incoming changes.

Submit - MERGE_IF_NECESSARY

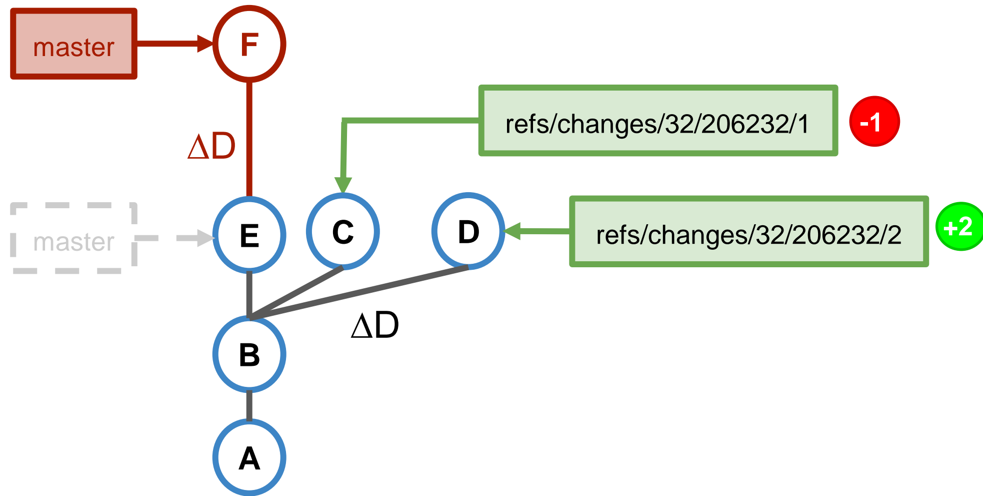


Since a fast-forward of the target branch is not possible a merge commit is created :

- The target branch is then fast-forwarded to the merge commit.
- The merge may fail due to conflicts.

Q: How would the result look like with REBASE_IF_NECESSARY?

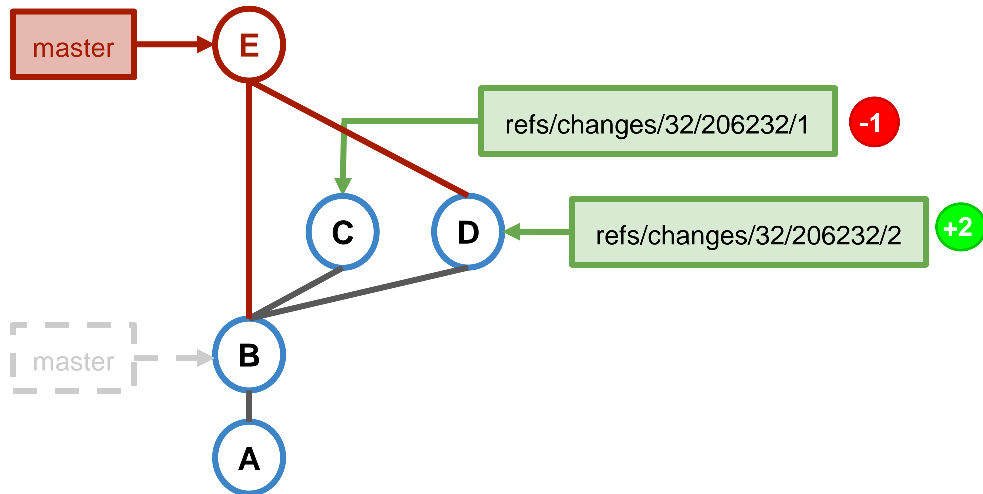
Submit - REBASE_IF_NECESSARY



Since a fast-forward of the target branch is not possible the current patch set, commit ***D***, is rebased which creates patch set ***F***:

- The target branch is then fast-forwarded to the merge commit.
- The rebase may fail due to conflicts.
- Results in linear history.

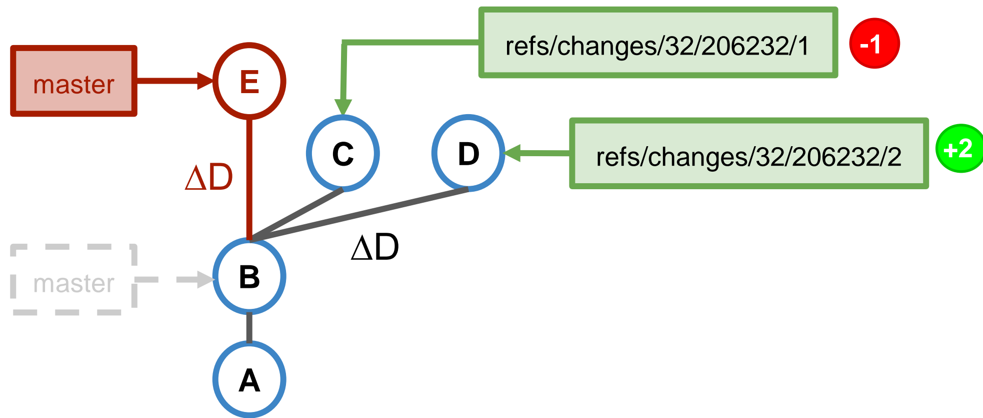
Submit - MERGE_ALWAYS



Although a fast-forward of the target branch to the current patch set, commit ***D***, is possible a merge commit ***F*** is created:

- The target branch is then fast-forwarded to the merge commit.
- Since a merge commit is always created you can always see from the version graph when a change got submitted (commit timestamp of the merge commit).

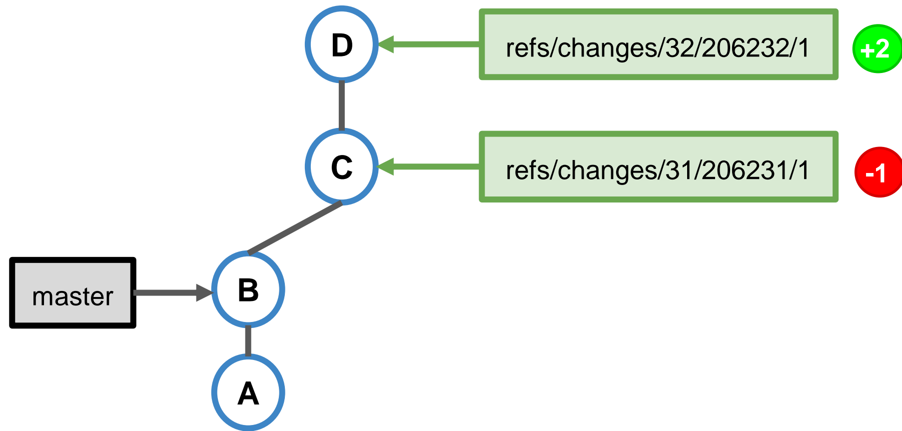
Submit - REBASE_ALWAYS



Although a fast-forward of the target branch to the current patch set, commit ***D***, is possible a rebase is done which creates commit ***E***:

- The target branch is then fast-forwarded to the rebased commit.
- Since the commit is always rebased you can see from the commit timestamp when the change got submitted.
- Guarantees that all submitted commits have additional footers which are inserted on rebase (e.g. *`Reviewed-On`*, *`Reviewed-By`*, *`Tested-By`*).

Submit - CHERRY_PICK

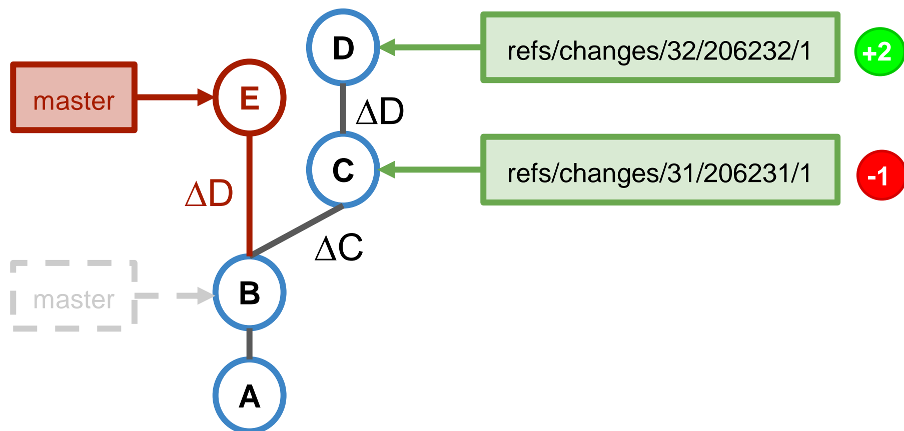


Situation:

- There is a change series with two changes, the change for commit ***D*** depends on the change for commit ***C***.
- The change for commit ***D*** was approved, the change for commit ***C*** got a negative review and needs to be reworked. Hence the change for commit ***C*** is not submittable.

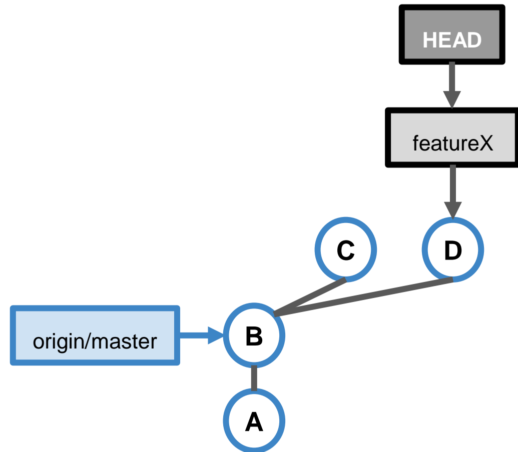
Q: What happens on submit of the change for commit D?

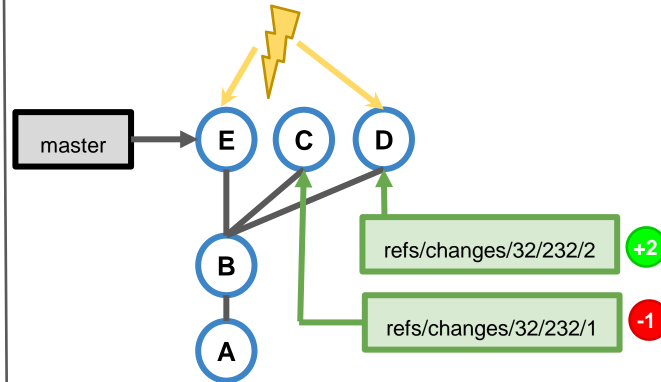
Submit - CHERRY_PICK



- The change for commit ***D*** is submittable since the `CHERRY_PICK` submit strategy **ignores change dependencies** (with all other submit strategies the change would be non-submittable because it depends on a non-submittable change).
- The current patch set of the submitted change is cherry-picked and the target branch is fast-forwarded to it.
- The cherry-pick may fail with conflicts.
- Guarantees that all submitted commits have additional footers which are inserted on cherry-pick (e.g. `Reviewed-On`, `Reviewed-By`, `Tested-By`).

Conflict Resolution





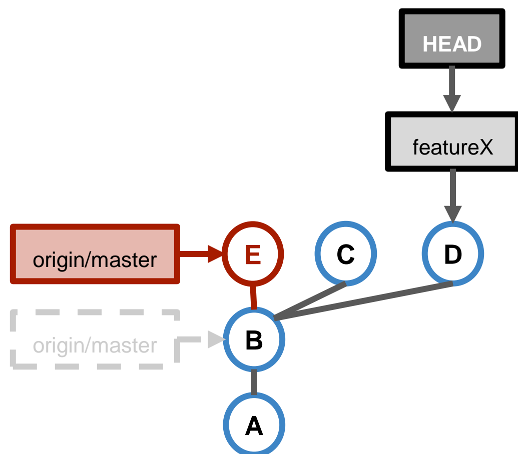
Situation:

- A change has two patch sets (commit ***C*** and commit ***D***) which are both based on commit ***B***. The current patch set (commit ***D***) was approved. In the meantime the `master` branch was updated to commit ***E***. There is a conflict between commit ***E*** (tip of the target branch) and commit ***D*** (current patch set) which prevents the submit of the change. The user has already checked out the `featureX` branch.

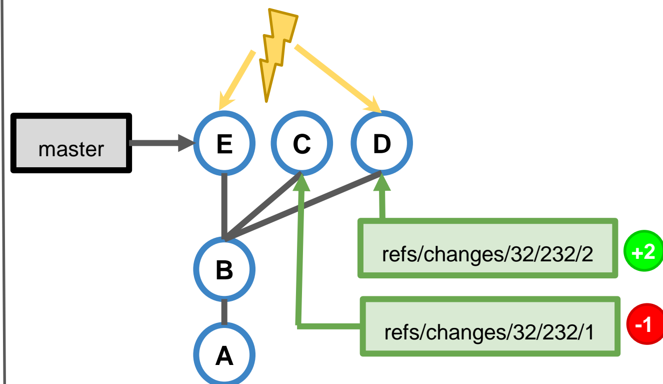
Q: How is the conflict resolved so that the change becomes submittable?

Conflict Resolution - Rebase Change

local repository



remote repository

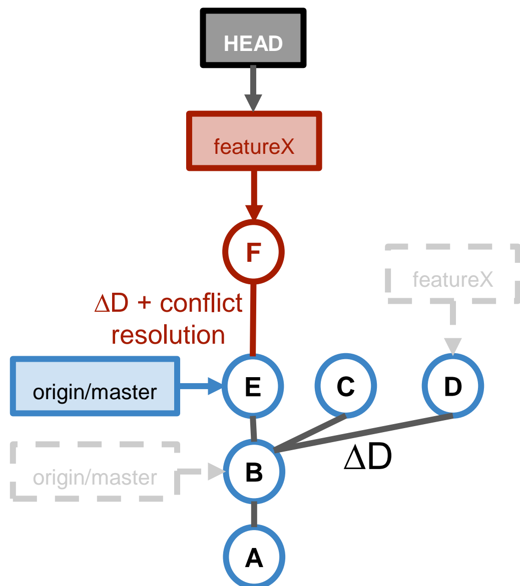


git fetch origin

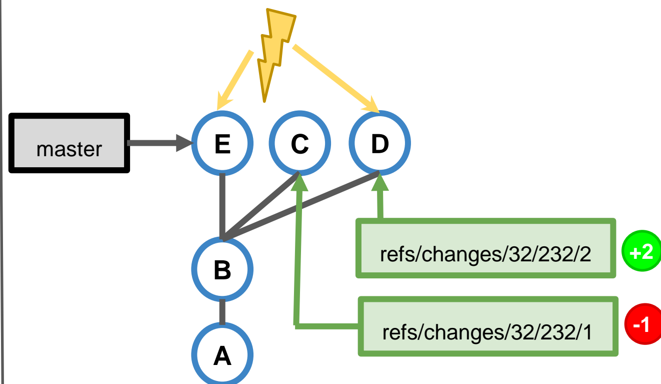
1. `git fetch origin`:
   - Brings commit ***E*** into the local repository and updates the remote tracking branch.

Conflict Resolution - Rebase Change

local repository



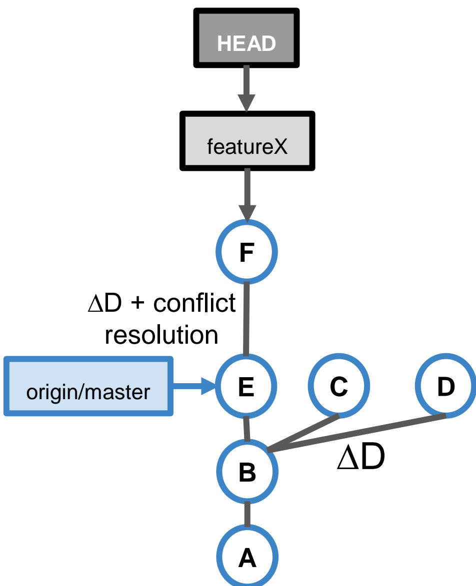
remote repository



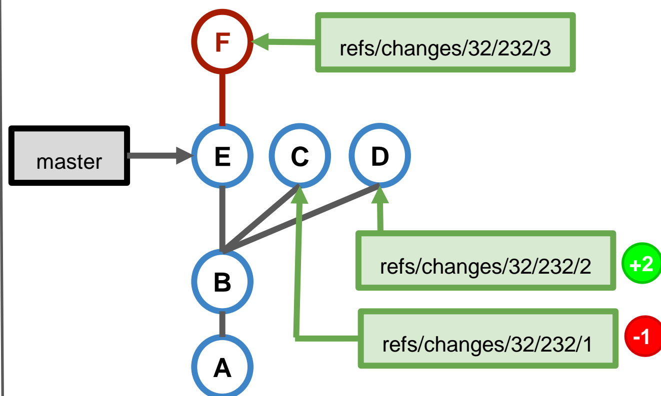
1. *git fetch origin*:
   - Brings commit E into the local repository and updates the remote tracking branch.
2. *git rebase origin/master*:
   - Rebases the *featureX* branch (commit ***D***) onto the remote tracking branch *origin/master*. During the rebase the conflicts need to be resolved.
   - On rebase the commit message, including the *Change-Id*, is preserved.

Conflict Resolution - Rebase Change

local repository



remote repository



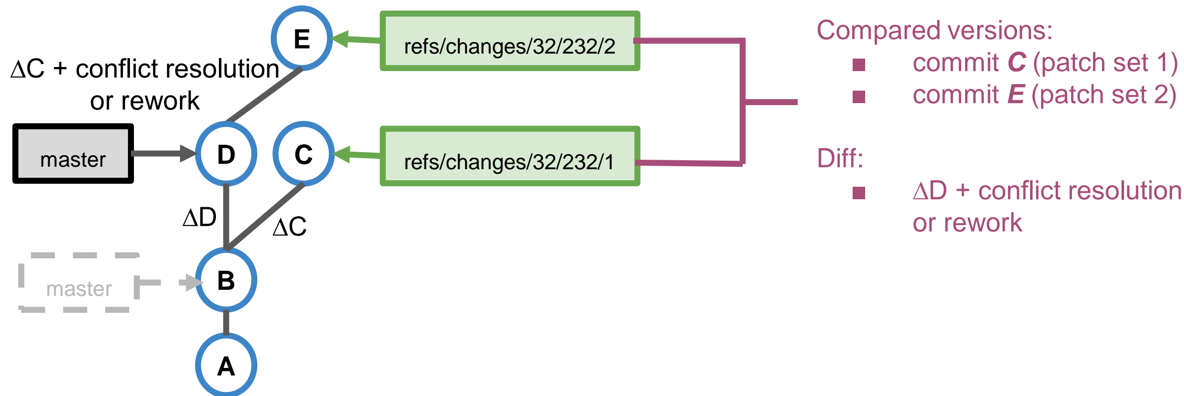
git push origin HEAD:refs/for/master

1. `git fetch origin`:
   - Brings commit E into the local repository and updates the remote tracking branch.
2. `git rebase origin/master`:
   - Rebases the `featureX` branch (commit ***D***) onto the remote tracking branch `origin/master`. During the rebase the conflicts need to be resolved.
   - On rebase the commit message, including the `Change-Id`, is preserved.
3. `git push origin HEAD:refs/for/master`:
   - Commit ***F*** is pushed and becomes a new patch set of the change.
   - `master` can now be fast-forwarded to commit ***F*** (current patch set), hence the change is submittable now

Comparing Patch Sets after Rebase

If patch sets are compared after a rebase was done the diff includes:

- modifications that have been done for the new patch set (conflict resolution or rework)
- modifications between the old and new base



Situation:

- The first patch set of a change (commit ***C***) was implemented based on commit ***B***. Then the `master` branch advanced to commit ***D*** and the change was rebased, which created commit ***E***.

When comparing patch sets after rebase

- the file list filters out files that only changed due to the rebase but which are not touched by the change
- the diff view show modifications that can be clearly attributed to the rebase in different colors

Comparing Patch Sets after Rebase

208035: Update most documentation for NoteDb — Documentation/config-gerrit.txt

File 2 of 3 | Prev | Up | Next

Patchset 3 gitiles → Patchset 6 gitiles | Download

SHOW BLAME | Diff view:

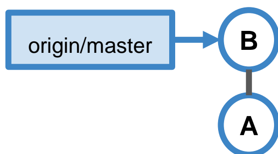
Patchset 3	Patchset 6
818 requires two HTTP requests, and this cache tries to carry state from	818 requires two HTTP requests, and this cache tries to carry state from
819 the first request into the second to ensure it can complete.	819 the first request into the second to ensure it can complete.
820	820
821 cache `"changes"`::	821 cache `"change_refs"`::
	822 +
	823 Cache entries are used to compute change ref visibility efficiently. Entries
	824 contain the minimal required information for this task.
822 +	825 +
823 The size of `memoryLimit` determines the number of projects for which	826 The size of `memoryLimit` determines the number of projects for which
824 all changes will be cached. If the cache is set to 1024, this means all	827 all changes will be cached. If the cache is set to 1024, this means all
825 changes for up to 1024 projects can be held in the cache.	828 at maximum 1024 changes can be held in the cache.
826 +	829 +
827 Default value is 0 (disabled). It is disabled by default due to the fact	830 Default value is 10.000.
828 that change updates are not communicated between Gerrit servers. Hence	
829 this cache should be disabled in an multi-master/multi-slave setup.	
830 +	831 +
831 The cache should be flushed whenever NoteDb change metadata is modified	832 If the size is set to 0, the cache is disabled. The change index will not
832 outside of Gerrit.	833 be used at all. Instead, the change notes cache is used directly.
	834 +
	835 A good size for this cache is twice the number of changes that the Gerrit
	836 instance has to allow it to hold all current changes and account for
	837 growth.
833	838
834 cache `"diff"`::	839 cache `"diff"`::
835 +	840 +
836 Each item caches the differences between two commits, at both the	841 Each item caches the differences between two commits, at both the
837 directory and file levels. Gerrit uses this cache to accelerate	842 directory and file levels. Gerrit uses this cache to accelerate
838 the display of affected file names, as well as file contents.	843 the display of affected file names, as well as file contents.
839 +	844 +
840 Entries in this cache are relatively large, so memoryLimit is an	845 Entries in this cache are relatively large, so memoryLimit is an
841 estimate in bytes of memory used. Administrators should try to target	846 estimate in bytes of memory used. Administrators should try to target
842 cache.diff.memoryLimit to fit all changes users will view in a 1 or 2	847 cache.diff.memoryLimit to fit all changes users will view in a 1 or 2
@@ +10↑ - Show 2026 common lines - +10↓	@@ +10↑ - Show 2026 common lines - +10↓
2869	2874

The example on the left side shows the diff of a file that changed between two patch sets both by rework and by rebase:

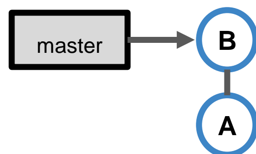
- modifications that have been done by reworking the change are shown in red and green
- modifications that were included by rebasing the change are shown in orange/yellow and blue/purple.

Standard Workflow (Summary)

local repository



remote repository

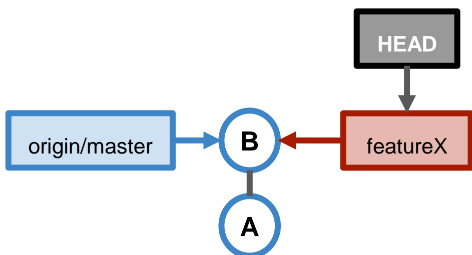


Situation:

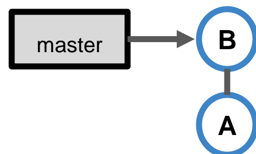
- The `master` branch in the remote repository contains two commits, commit ***A*** and commit ***B***. Both commits have been fetched into the local repository.

Standard Workflow (Summary)

local repository



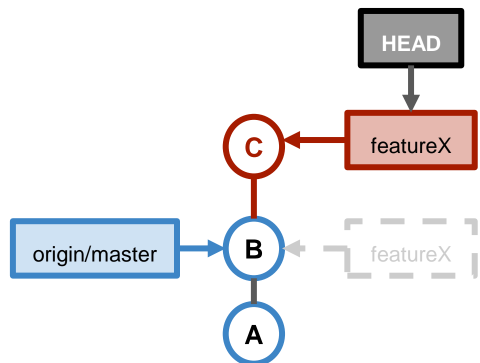
remote repository



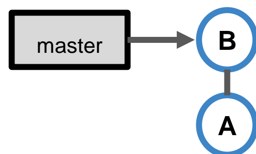
1. Create and checkout a local feature branch which is based on *origin/master*:
   `git checkout -b featureX origin/master`

Standard Workflow (Summary)

local repository



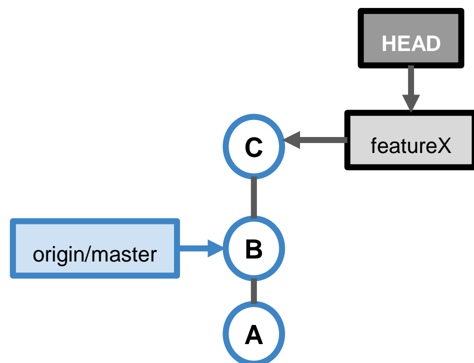
remote repository



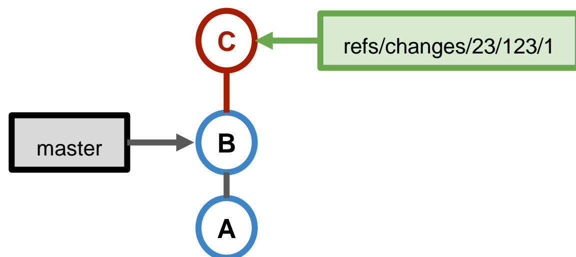
1. Create and checkout a local feature branch which is based on *origin/master*:
   git checkout -b featureX origin/master
2. Make a new commit that implements the feature.

Standard Workflow (Summary)

local repository



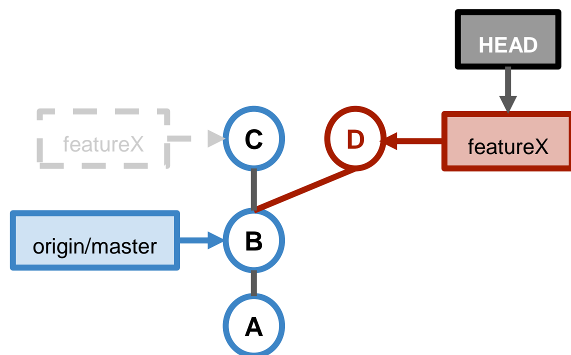
remote repository



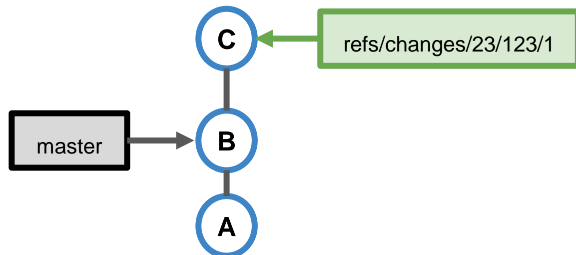
1. Create and checkout a local feature branch which is based on *origin/master*:
   git checkout -b featureX origin/master
2. Make a new commit that implements the feature.
3. Push the commit for code review:
   git push origin HEAD:refs/for/master

Standard Workflow (Summary)

local repository

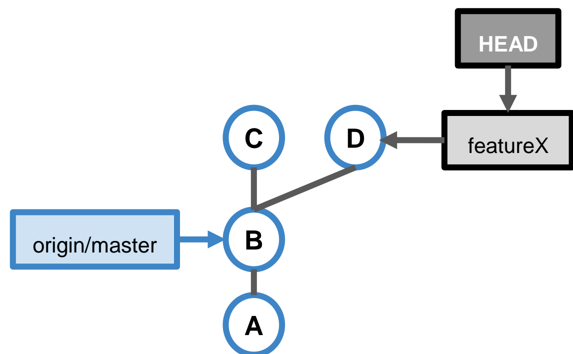


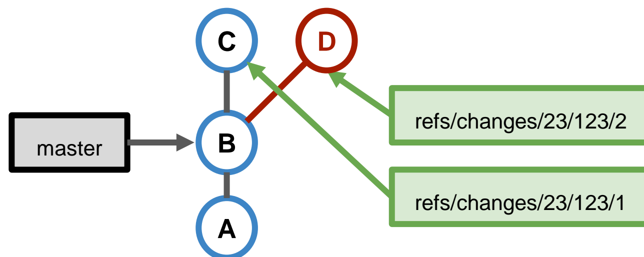
remote repository



1. Create and checkout a local feature branch which is based on *`origin/master`*:
   `git checkout -b featureX origin/master`
2. Make a new commit that implements the feature.
3. Push the commit for code review:
   `git push origin HEAD:refs/for/master`
4. If rework is needed:
   a. Checkout the *`featureX`* branch and amend the commit.

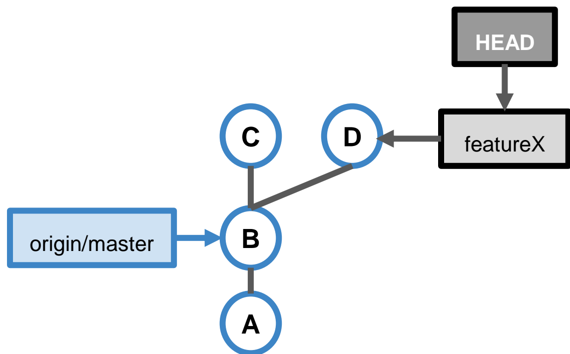
Standard Workflow (Summary)

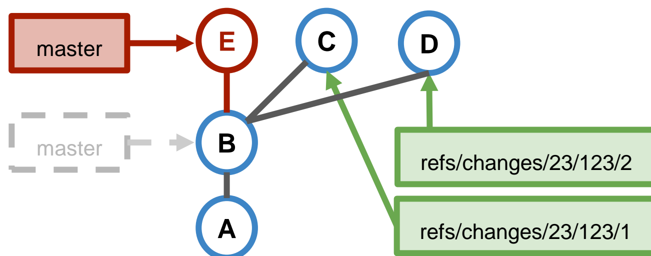




1. Create and checkout a local feature branch which is based on *origin/master*:
   git checkout -b featureX origin/master
2. Make a new commit that implements the feature.
3. Push the commit for code review:
   git push origin HEAD:refs/for/master
4. If rework is needed:
   a. Checkout the *featureX* branch and amend the commit.
   b. Push the new commit for review to create a second patch set

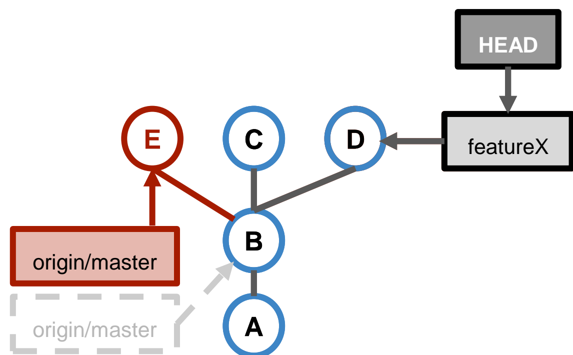
Standard Workflow (Summary)

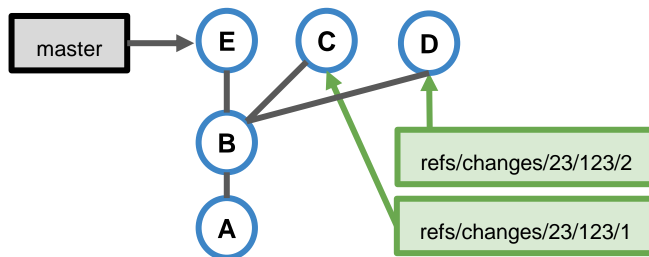




1. ...
2. Make a new commit that implements the feature.
3. Push the commit for code review:
   git push origin HEAD:refs/for/master
4. If rework is needed:
   a. Checkout the *featureX* branch and amend the commit.
   b. Push the new commit for review to create a second patch set
5. If rebase is needed

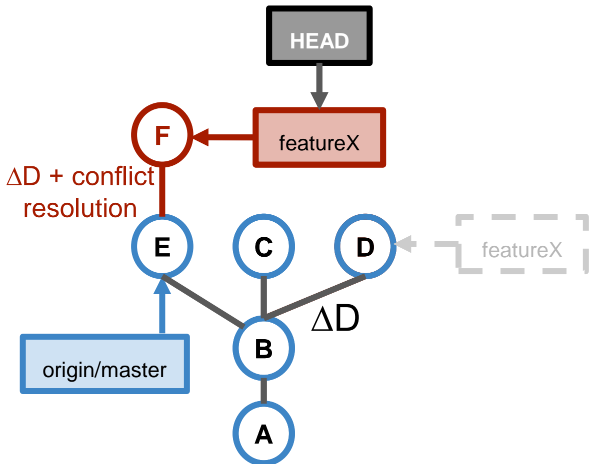
Standard Workflow (Summary)

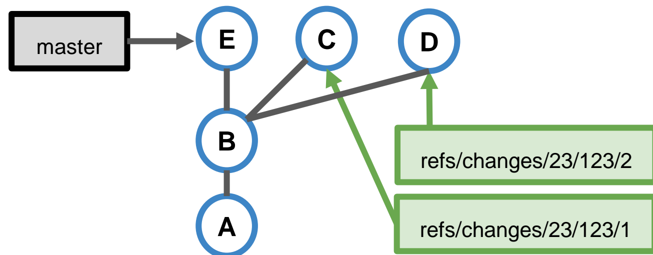




1. ...
2. Make a new commit that implements the feature.
3. Push the commit for code review:
   git push origin HEAD:refs/for/master
4. If rework is needed:
   a. Checkout the *featureX* branch and amend the commit.
   b. Push the new commit for review to create a second patch set
5. If rebase is needed:
   a. Fetch the updates from the remote repository

Standard Workflow (Summary)

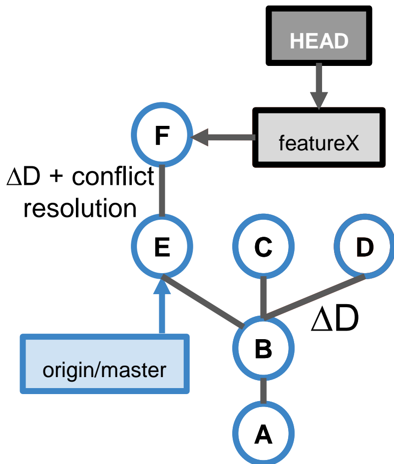




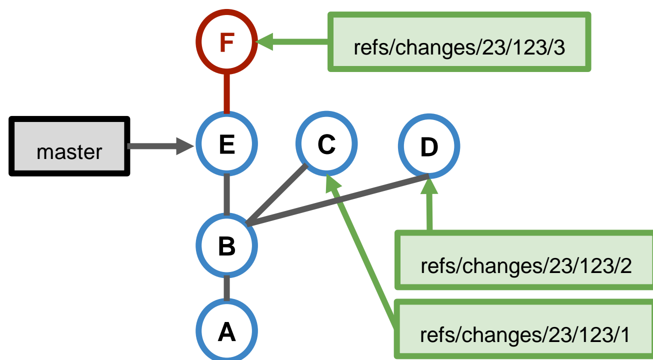
1. ...
2. Make a new commit that implements the feature.
3. Push the commit for code review: `git push origin HEAD:refs/for/master`
4. If rework is needed:
   a. Checkout the `featureX` branch and amend the commit.
   b. Push the new commit for review to create a second patch set
5. If rebase is needed:
   a. Fetch the updates from the remote repository
   b. Checkout the `featureX` branch and rebase it onto `origin/master`

Standard Workflow (Summary)

local repository



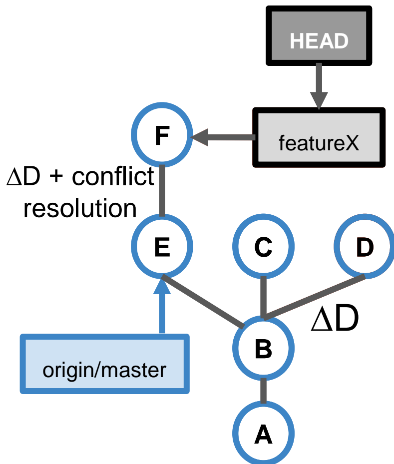
remote repository



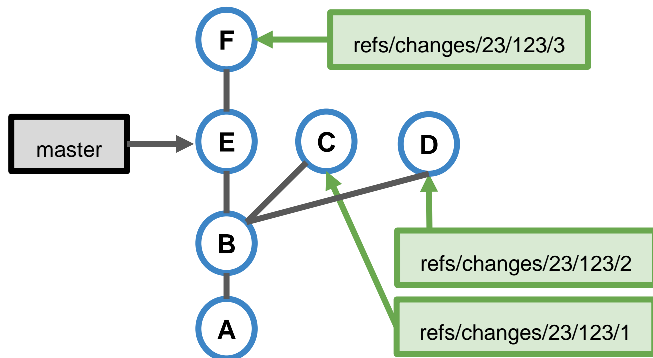
1. ...
2. …
3. ...
4. If rework is needed:
   a. Checkout the *featureX* branch and amend the commit.
   b. Push the new commit for review to create a second patch set
5. If rebase is needed:
   a. Fetch the updates from the remote repository
   b. Checkout the *featureX* branch and rebase it onto *origin/master*
   c. Push the new commit for review to create a third patch set

Standard Workflow (Summary)

local repository



remote repository



Note that we didn't use any **local** `master` branch. In fact a `master` branch in the local repository is not needed when working with Gerrit. It's recommended to delete the **local** `master` branch to avoid any confusion with it.

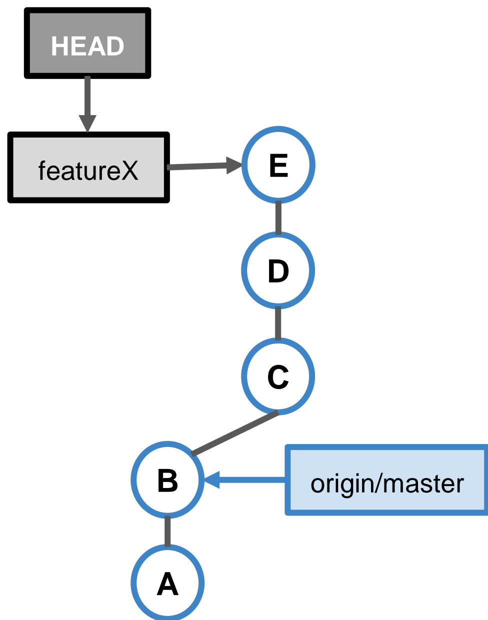
Alternative Workflow

Same as standard workflow, but don’t create local feature branches and work with ***detached HEAD*** instead.

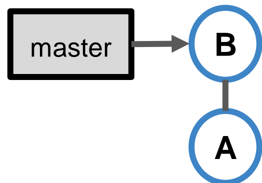
- After pushing a commit for code review it is available on Gerrit and it can be fetched from there whenever the change needs to be reworked or rebased (copy the fetch command from the change screen).
- Working with detached HEAD has the risk of losing reference to commits when you checkout another state to work on something else (e.g. if you forgot to push new commits to Gerrit)
- If you need to checkout another state but your current work is not ready for push yet, create a commit and a local feature branch for it. You can then resume the work later by checking out this local feature branch (same as standard workflow).

Working with Change Series

local repository



remote repository



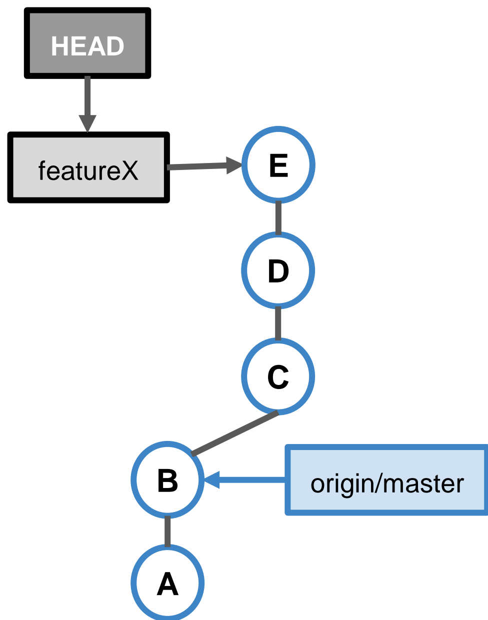
It is common that features are implemented by **multiple self-contained commits** that are based on each other.

Situation:

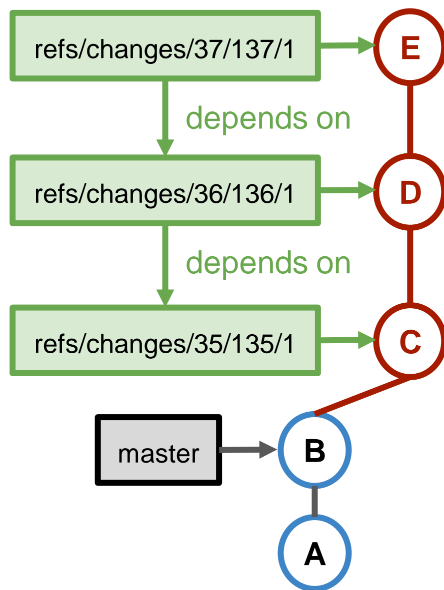
- The remote repository was cloned, a local `featureX` branch was created and in this branch three commits, ***C***, ***D*** and ***E***, were created.

Working with Change Series

local repository

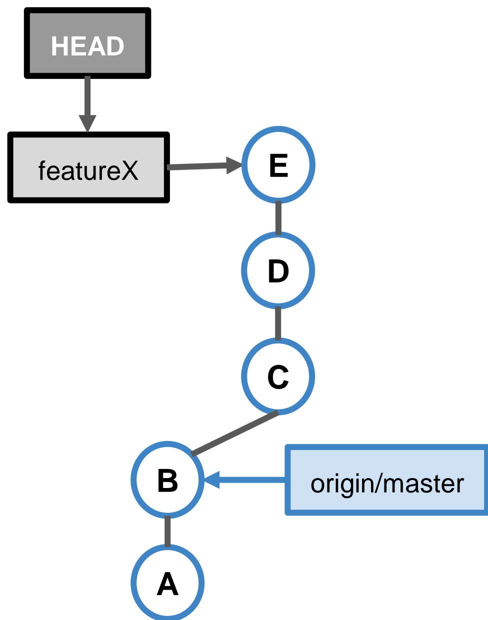


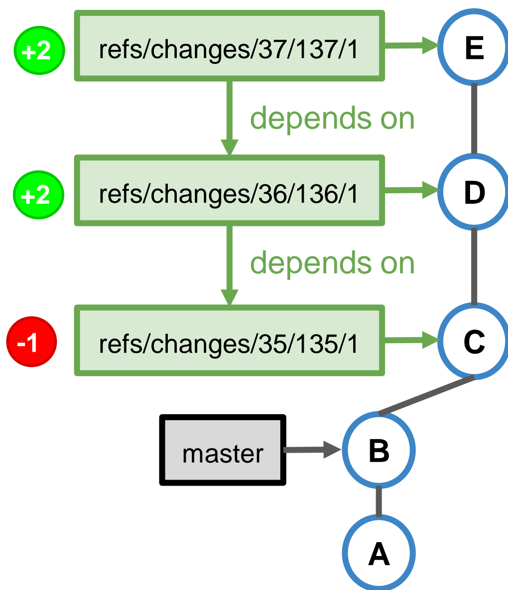
remote repository



All commits in the `featureX` branch can be pushed for code review by a single `git push` command. For each of the pushed commits a change is created. The changes depend on each other the same way as the commits depend on each other.

Working with Change Series



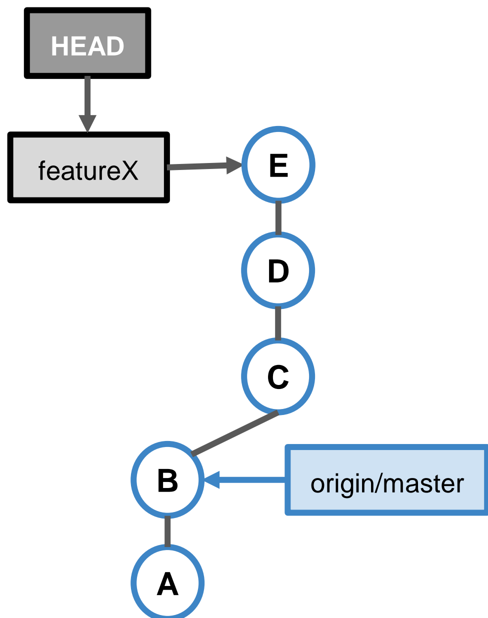


If a change in the series needs to be reworked checkout the *featureX* branch and use interactive rebase to rewrite the commits in the *featureX* branch:

```
git rebase -i origin/master
```

Working with Change Series

local repository



git rebase -i origin/master

Opens editor with rewrite plan

```
pick 7888debe88 C
pick 0a12290e7b D
pick deb7e4d61f E
```

Use `edit` command for commit C that needs rework

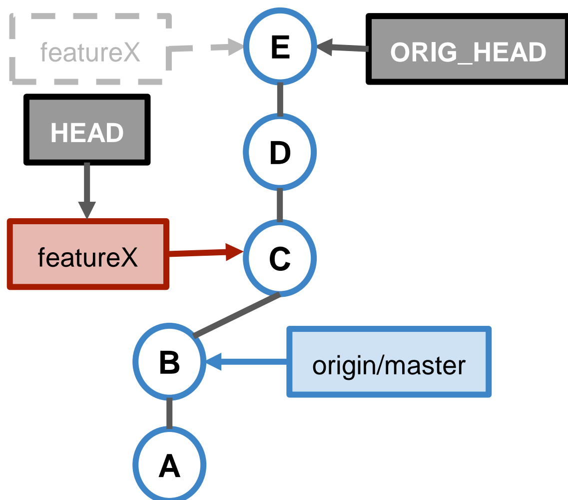
```
edit 7888debe88 C
pick 0a12290e7b D
pick deb7e4d61f E
```

If a change in the series needs to be reworked checkout the *featureX* branch and use interactive rebase to rewrite the commits in the *featureX* branch:

git rebase -i origin/master

Working with Change Series

local repository



git rebase -i origin/master

Opens editor
with rewrite plan

```
pick 7888debe88 C
pick 0a12290e7b D
pick deb7e4d61f E
```

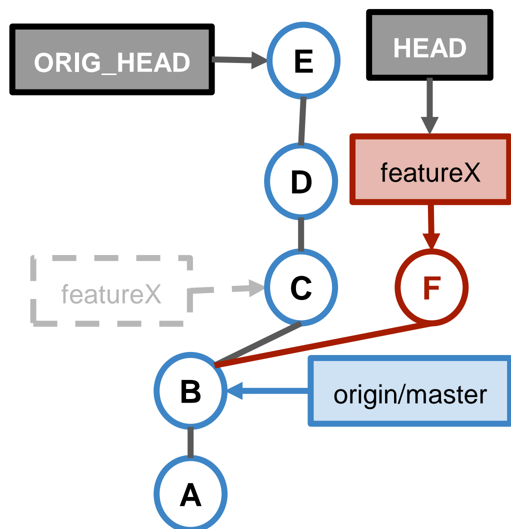
Use *edit* command
for commit C that
needs rework

```
edit 7888debe88 C
pick 0a12290e7b D
pick deb7e4d61f E
```

This rewinds the *featureX* branch to commit ***C*** where the interactive rebase stops so that it can be edited.

Working with Change Series

local repository



git rebase -i origin/master

Opens editor with rewrite plan

```
pick 7888debe88 C
pick 0a12290e7b D
pick deb7e4d61f E
```

Use *edit* command for commit C that needs rework

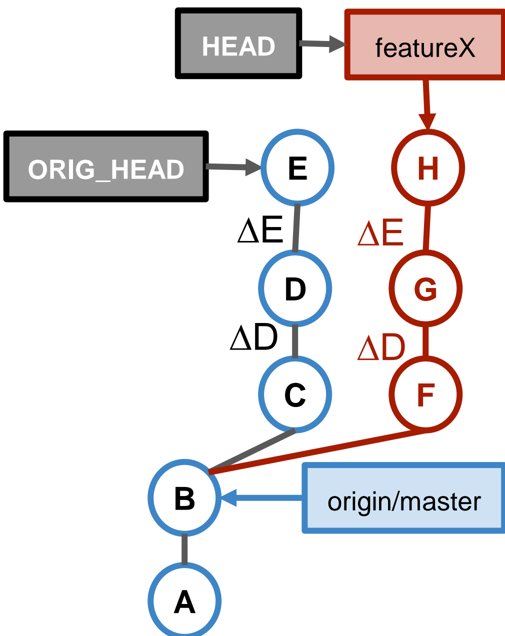
```
edit 7888debe88 C
pick 0a12290e7b D
pick deb7e4d61f E
```

This rewinds the *featureX* branch to commit ***C*** where the interactive rebase stops so that it can be edited:

1. Edit commit ***C*** by amending it.

Working with Change Series

local repository



git rebase -i origin/master

Opens editor with rewrite plan

```
pick 7888debe88 C
pick 0a12290e7b D
pick deb7e4d61f E
```

Use *edit* command for commit C that needs rework

```
edit 7888debe88 C
pick 0a12290e7b D
pick deb7e4d61f E
```

This rewinds the *featureX* branch to commit ***C*** where the interactive rebase stops so that it can be edited:

1. Edit commit ***C*** by amending it.
2. Continue the rebase by *git rebase --continue*, this will ***cherry-pick*** commit ***D*** and commit ***E*** on top of the amended commit ***F***.

Working with Change Series

local repository

HEAD
featureX
ORIG_HEAD
E
H
ΔE
ΔE
D
G
ΔD
ΔD
C
F
B
origin/master
A

remote repository

+2
.../137/1
E
H
.../137/2
depends on
depends on
+2
.../136/1
D
G
.../136/2
depends on
depends on
-1
.../135/1
C
F
.../135/2
master
B
A

Once the rebase is done the new commits can be push for code review by a single `git push` command. This creates new patch sets for all 3 changes (remember that the `Change-Ids` in the commit messages are preserved on rebase).

Q: What happens with the votes on the changes?

Sticky Votes

On upload of a new patch set the current votes are either removed or copied to the new patch set:

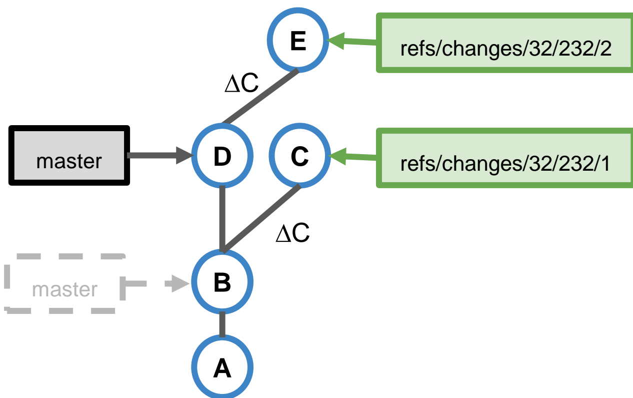
Whether a vote is copied depends on the **label configuration**:

- `copyMinScore`:
  The vote is copied if it was the lowest possible vote of the review label.
- `copyMaxScore`:
  The vote is copied if it was the highest possible vote of the review label.
- `copyAllScoresOnTrivialRebase`:
  The vote is copied if the new patch set is a trivial rebase of the current patch set. A ***trivial rebase*** is a rebase that doesn't include conflict resolution or rework.
- `copyAllScoresIfNoCodeChange`:
  The vote is copied if the new patch set only modified the commit message.
- `copyAllScoresIfNoChange`:
  The vote is copied if the only difference between the current patch set and the new patch set is the SHA1.
- `copyAllScoresOnMergeFirstParentUpdate`:
  Only relevant for merge commits. The vote is copied forward if a merge commit is rebased.

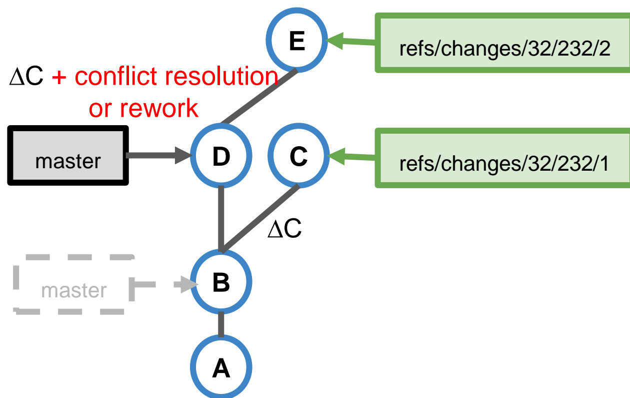
- ***review labels*** can be defined and configured per repository (see next slides)
- if a vote or approval is copied forward to new patch sets it is called ***sticky***
- `copyMinScore` must be set to enable ***veto votes***
- Review labels that are set by CI systems often use `copyAllScoresIfNoCodeChange` since the CI verification depends only on the code but not on the commit message.
- In the API the kind of changes that are done by a patch set in comparison to the predecessor patch set is exposed as ***change kind***.

Trivial Rebase

Trivial Rebase



No Trivial Rebase



Situation:

- The first patch set of a change (commit ***C***) was implemented based on commit ***B***. Then the `master` branch advanced to commit ***D*** and the change was rebased, which created commit ***E***.

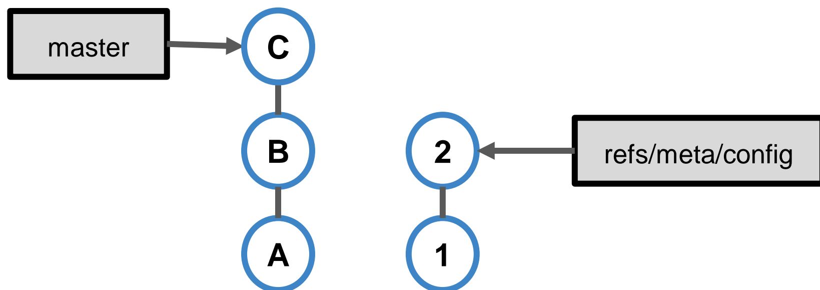
A new patch set is considered as ***trivial rebase*** if the commit message is the same as in the previous patch set and if it has the same code delta as the previous patch set.

Repository Configuration

Gerrit stores a configuration per repository.

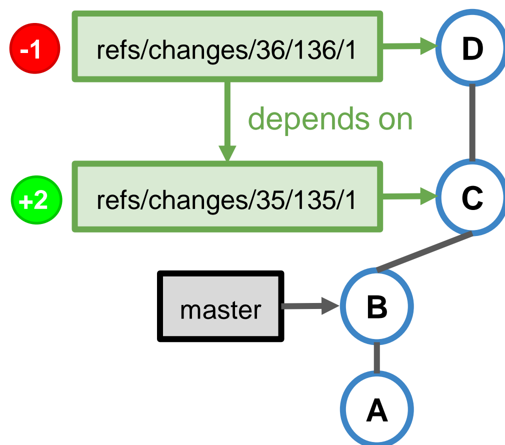
This configuration contains:

- a repository description
- the parent repository
- the access rights
- project options (e.g. submit type, the `Allow content merges` setting etc.)
- the definition of the ***review labels***, including the possible voting values



- Gerrit uses the term ***project*** as synonym for ***repository***.
- Every repository has a special `refs/meta/config` branch that contains a `project.config` file with the Gerrit configuration for the repository.
- The `refs/meta/config` branch is completely disconnected from the branches that are used for development (they don’t share any common ancestor commit).
- By default only repository owners have access to the `refs/meta/config` branch.
- Many settings in the repository configuration are **inheritable**. This means settings on a parent repository apply to all child repositories unless they are overwritten in the child repositories. Examples for inheritable settings are: access rights, submit type and review label definitions

Working with Topics

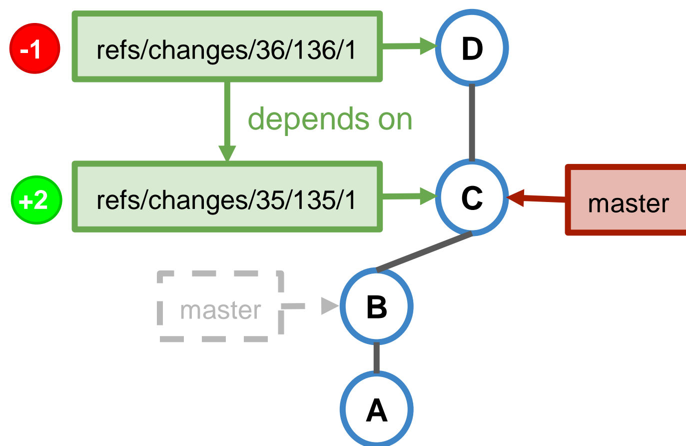


Situation:

- Two commits, commit ***C*** and ***D***, have been pushed for code review. This created two changes that depend on each other. The bottom change was approved and is submittable, the top change got a negative review and hence cannot be submitted.

Q: What happens on submit of the bottom change (change for commit C)?

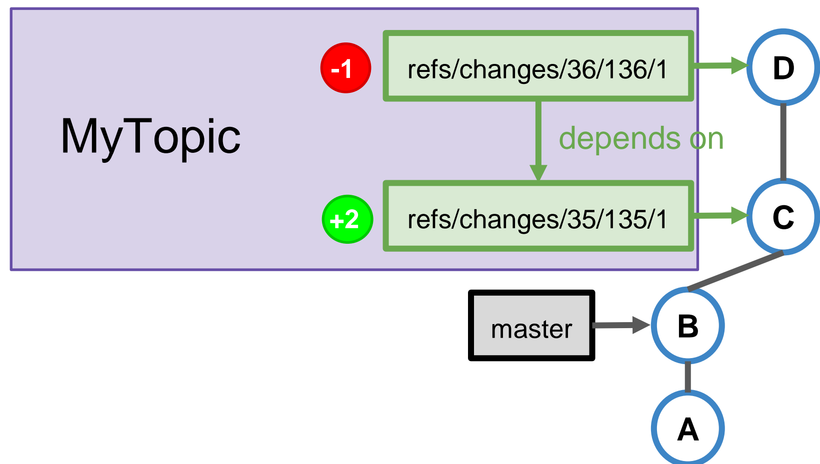
Working with Topics



The bottom change can be submitted since it was approved and doesn't depend on any non-submittable change. The top change stays open.

Q: Can it be enforced that both changes are only submittable together? If yes, how?

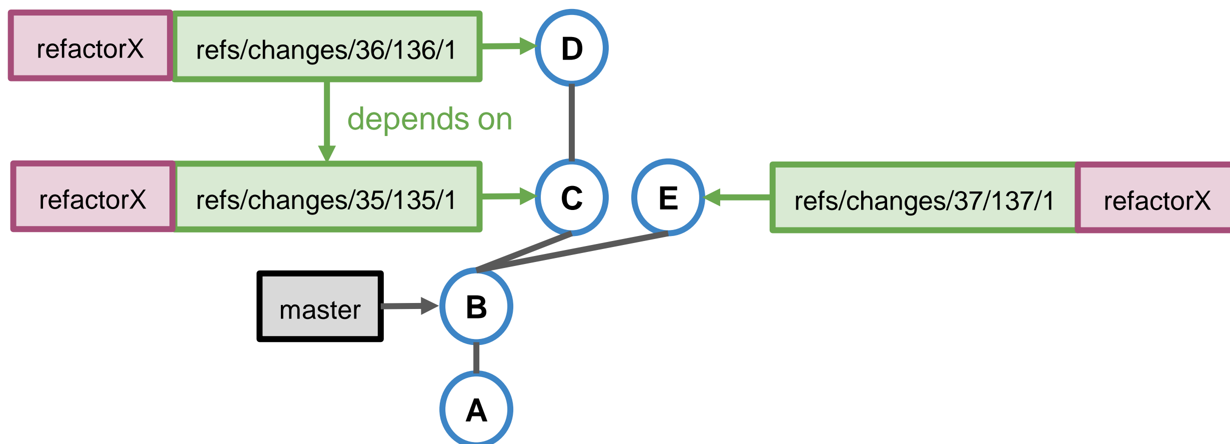
Working with Topics



Changes can be grouped by ***topics*** to enforce that they can only be **submitted together**. If both changes share the same topic, the topic can only be submitted when both changes are approved and submittable.

- A change can have at most one topic.
- Also changes that don't depend on each other can have the same topic and be submitted together.
- Topics can also be used across repositories. This is useful if there is a dependency between two repositories, e.g. one repository defines an API that is used in another repository. If now the API is changed you can assign the same topic to the API change and the change that adapts the other repository to the new API to enforce that both changes are submitted together.
- Changes of topics that are limited to one repository are guaranteed to be submitted atomically.
- Topics that span repositories are not submitted atomically (it's rather like clicking submit on the tip change of all branches in a fast sequence)
- Whether topics enforce that the changes are submitted together is configurable on server level.
- Topics can be set on push:
  `git push origin HEAD:refs/for/master -o topic=MyTopic`

Hashtags

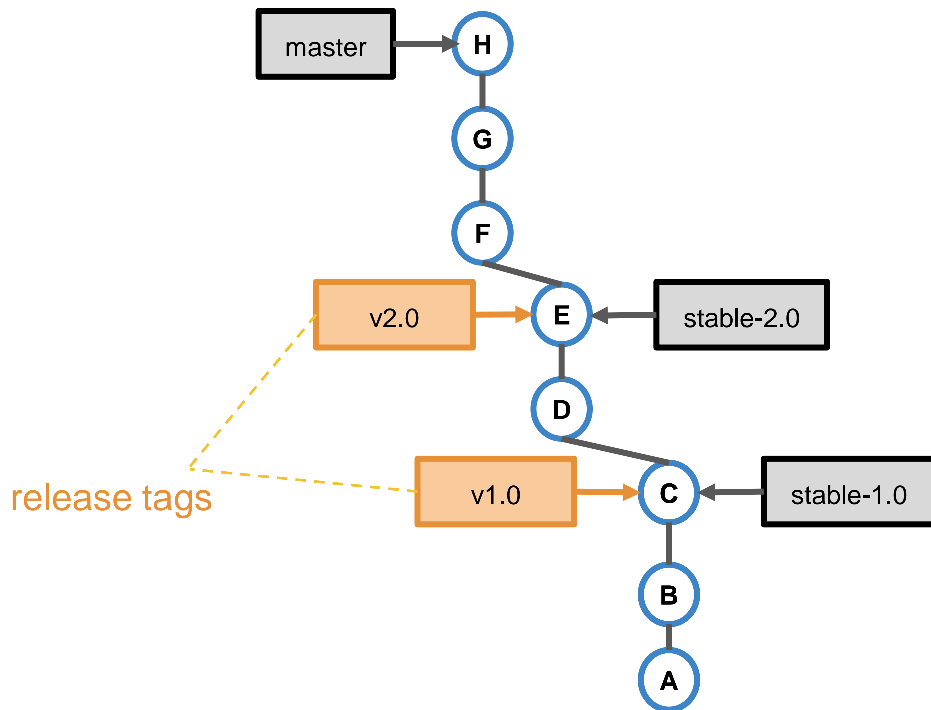


Hashtags allow to group arbitrary changes by common tags:

- Changes can have many hashtags.
- You can query for changes that have a certain hashtag and link to them.
- In contrast to ***topics***, changes with the same hashtag can be submitted independently.

Working with Stable Branches

remote repository



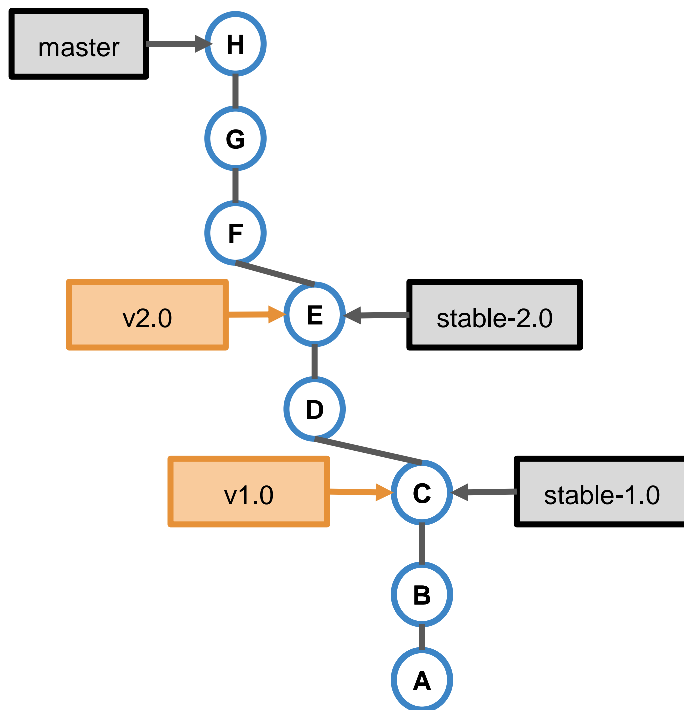
If **several versions of a project** need to be maintained it's a common practise to have one (central) branch for each major project version:

- The *master* branch is used to prepare the next major release. This means all new feature development should happen in this branch.
- *stable-<version>* branches are used to prepare bug-fix releases. Normally you only do bug-fixes in these branches, and no feature development. Stable branches can be created pro-actively when a new major version is released, or on need when the first bug-fix for a released version is needed (in this case you create the stable branch from the release tag).
- Often a stable branch is already created at the moment when the next major release should be done. Because then you can use it for stabilization and creation of ***release candidates*** while the feature development in *master* is ongoing.

Q: If a bug is discovered where should it be fixed?

Working with Stable Branches

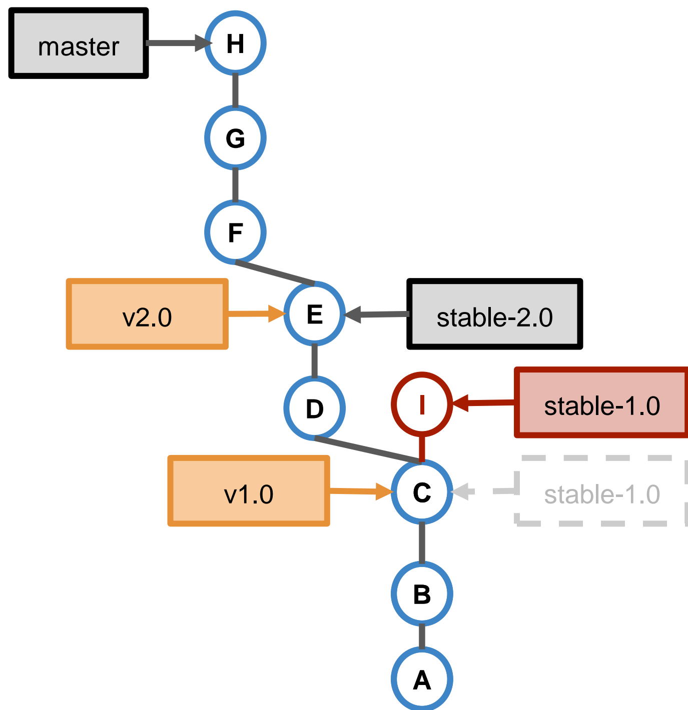
remote repository



There are 2 possibilities to do bug-fixes:

1. Do the bug-fix in the oldest stable branch that is effected and then merge the stable branches forward until the bug-fix reaches the `master` branch (preferred option).

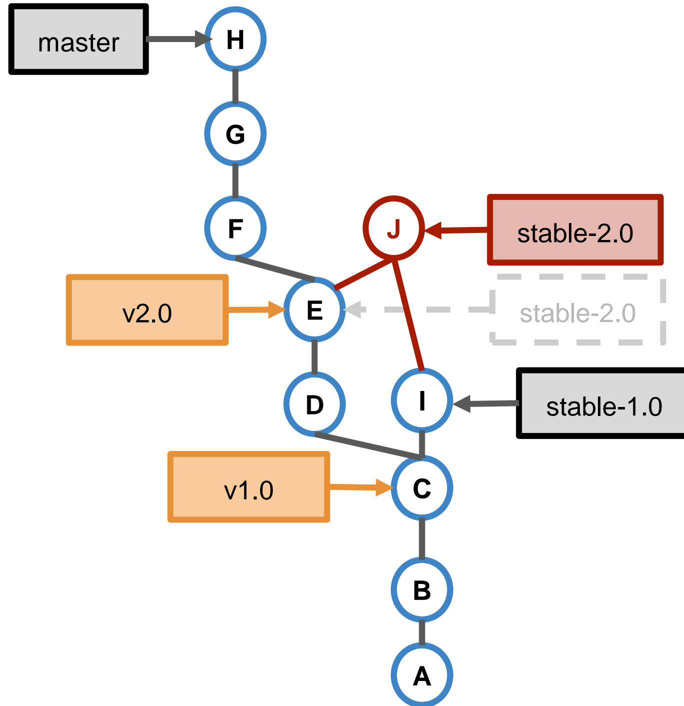
Working with Stable Branches - Option 1



There are 2 possibilities to do bug-fixes:

1. Do the bug-fix in the oldest stable branch that is effected and then merge the stable branches forward until the bug-fix reaches the `master` branch (preferred option).

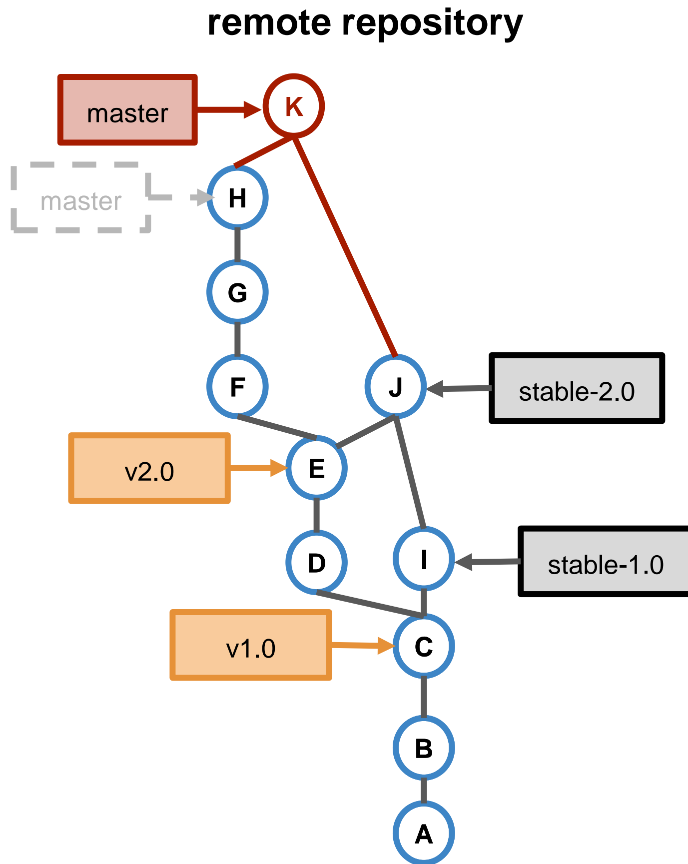
Working with Stable Branches - Option 1



There are 2 possibilities to do bug-fixes:

1. Do the bug-fix in the oldest stable branch that is effected and then merge the stable branches forward until the bug-fix reaches the `master` branch (preferred option).

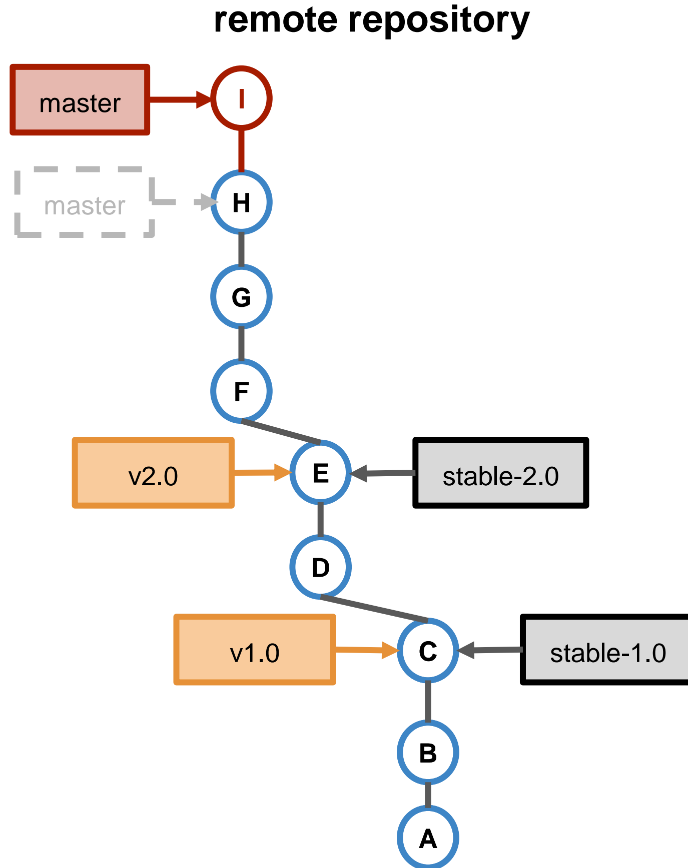
Working with Stable Branches - Option 1



There are 2 possibilities to do bug-fixes:

1. Do the bug-fix in the oldest stable branch that is effected and then merge the stable branches forward until the bug-fix reaches the `master` branch (preferred option).

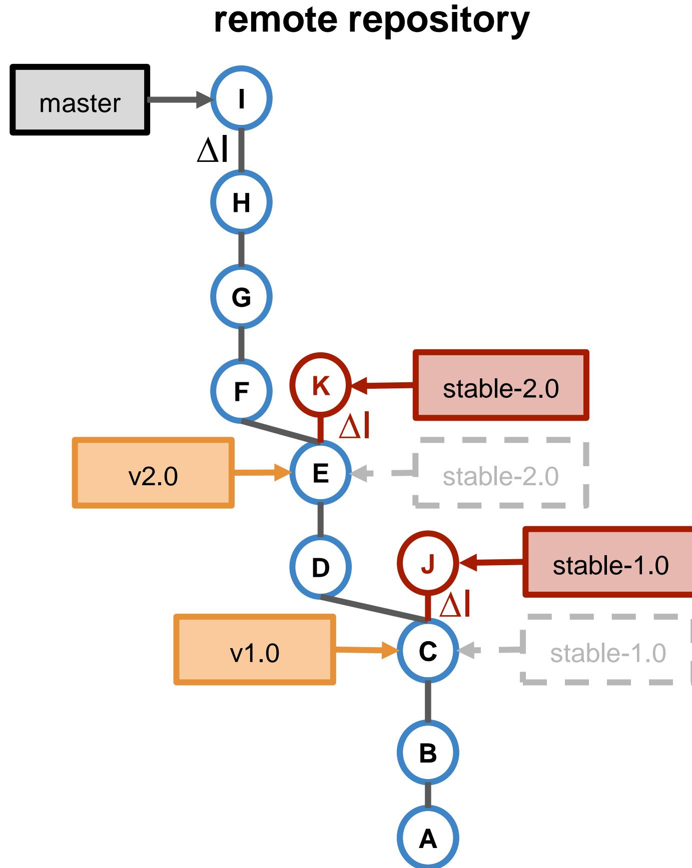
Working with Stable Branches - Option 2



There are 2 possibilities to do bug-fixes:

1. Do the bug-fix in the oldest stable branch that is effected and then merge the stable branches forward until the bug-fix reaches the `master` branch (preferred option).
2. Do the bug-fix in the `master` branch and then cherry-pick it to the stable branches.

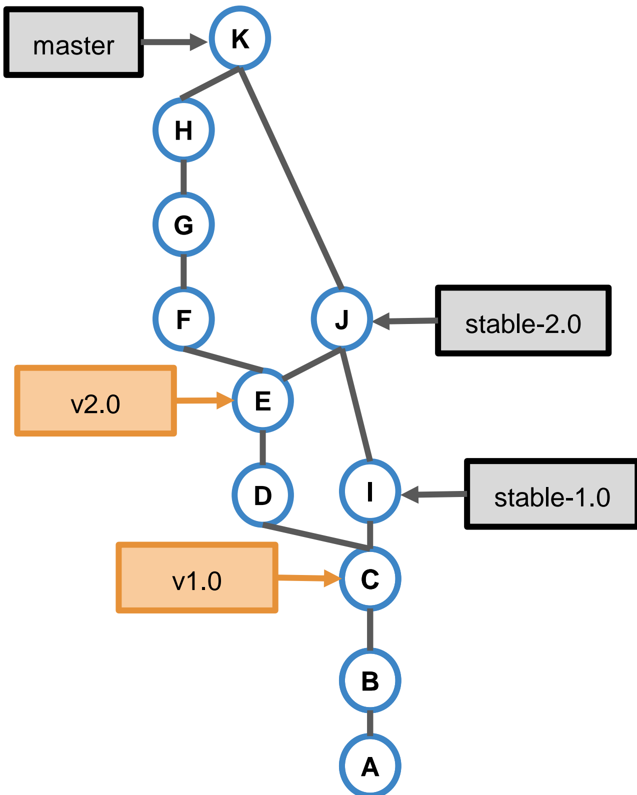
Working with Stable Branches - Option 2

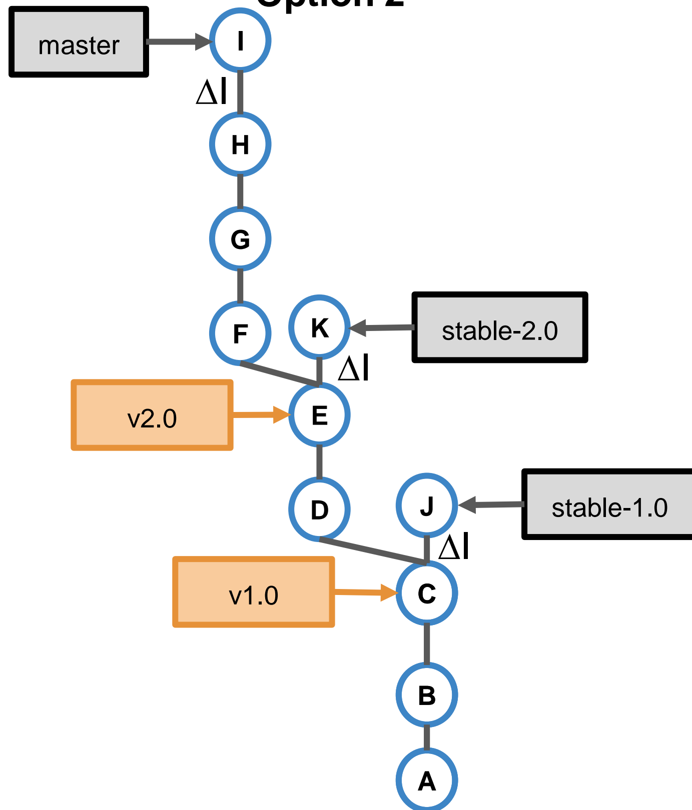


There are 2 possibilities to do bug-fixes:

1. Do the bug-fix in the oldest stable branch that is effected and then merge the stable branches forward until the bug-fix reaches the `master` branch (preferred option).
2. Do the bug-fix in the `master` branch and then cherry-pick it to the stable branches.

Working with Stable Branches



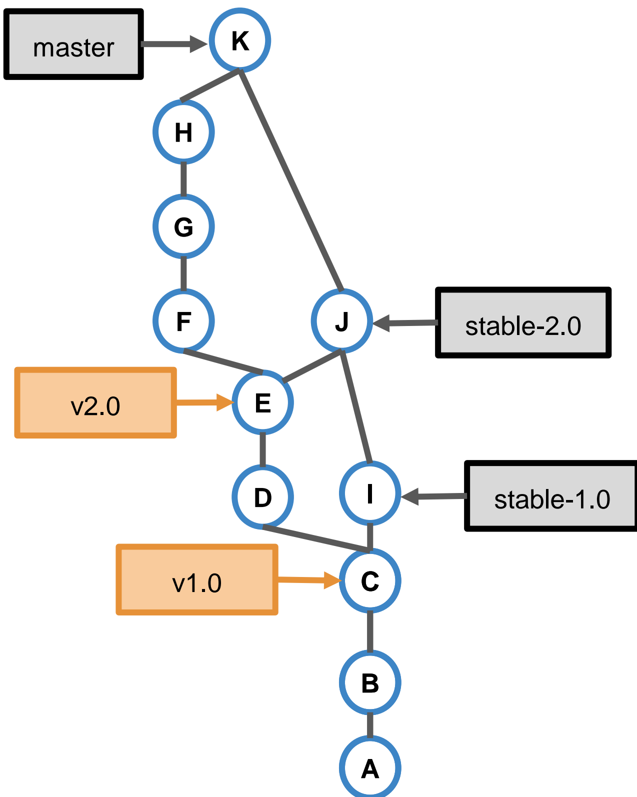


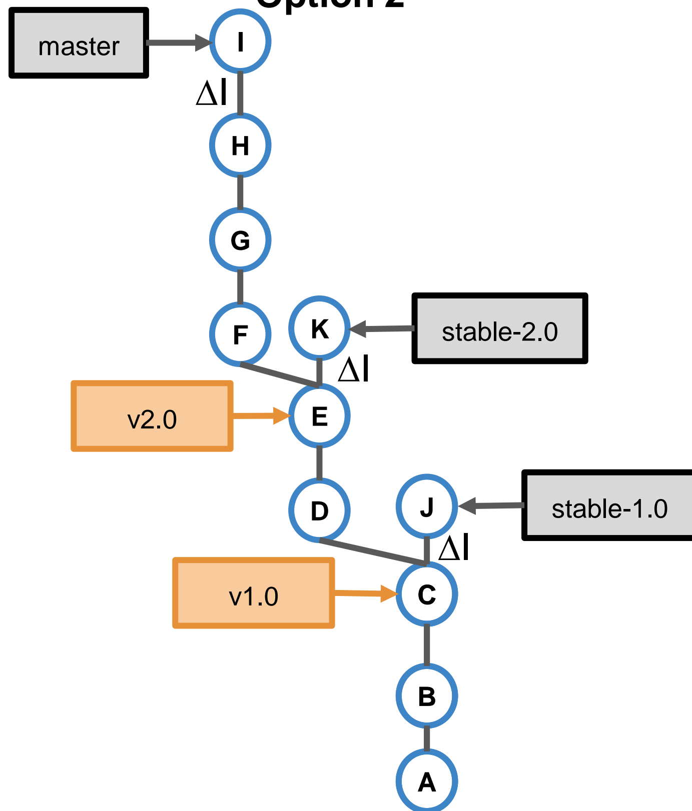
There are 2 possibilities to do bug-fixes:

1. Do the bug-fix in the oldest stable branch that is effected and then merge the stable branches forward until the bug-fix reaches the `master` branch (preferred option).
2. Do the bug-fix in the `master` branch and then cherry-pick it to the stable branches.

Q: Why is option 1 generally preferred? When would you use option 2?

Working with Stable Branches



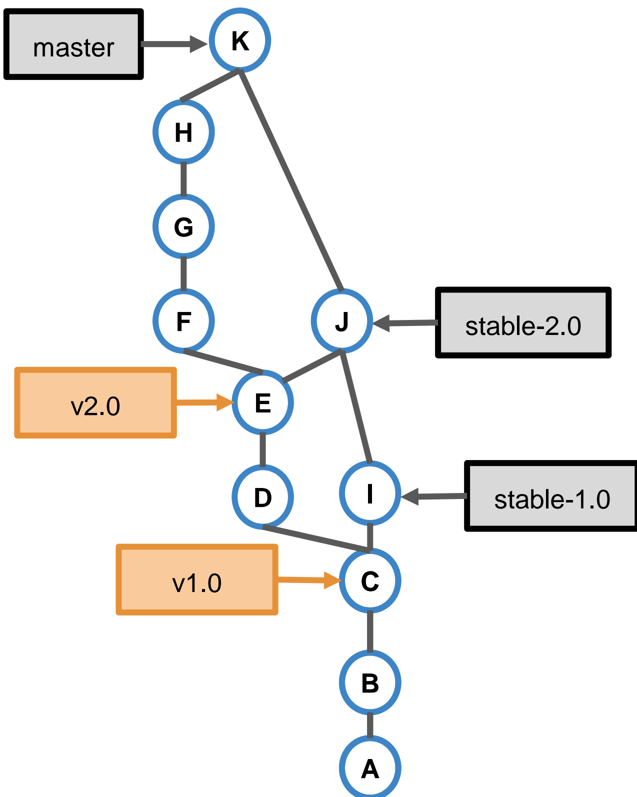


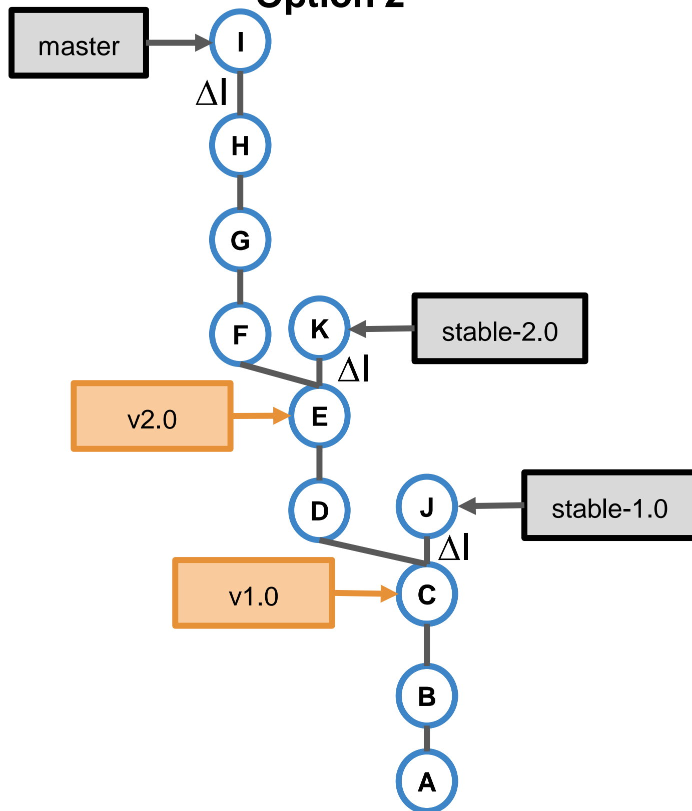
Option 1 is generally preferred because there is exactly one commit that implements the bug-fix (commit *I*). This means you can easily ask Git for all branches that contain the bug-fix:

```
git branch -r --contains I
```

Option 1 assumes that everything that is done in a stable branches should also be applied to `master` and newer stable branches. Note that merging into the other direction, e.g. `master` into `stable-2.0`, is bad since it would bring the features that have been implemented in `master` into the `stable-2.0` branch.

Working with Stable Branches



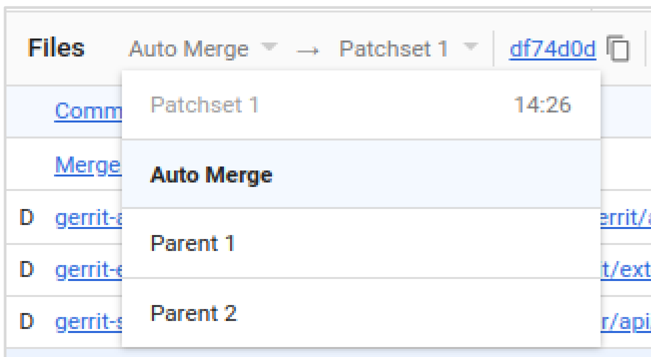


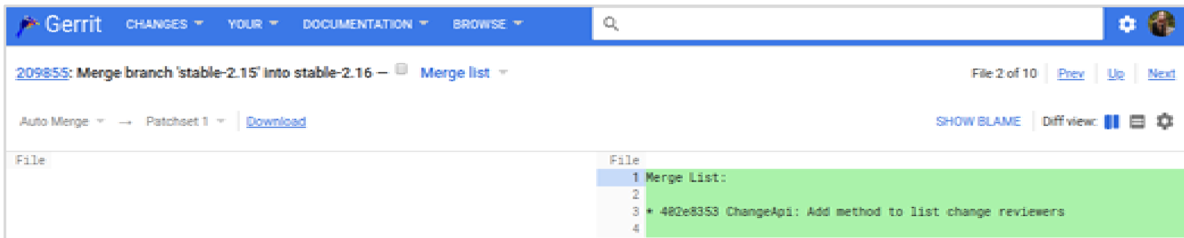
With **option 2** there are multiple commits that implement the bug-fix (commits ***I***, ***J***, ***K***) but from the version graph you can't see that they do the same modifications. However since all commits share the same `Change-Id` you can at least find all of them in Gerrit by searching by the `Change-Id`.

Option 2 is used if a bug-fix has already been done in `master` and only then it's discovered that it's also needed in a stable branch.

Option 2 is often preferred if it's known that the affected code has changed in such a way that there will be merge conflicts when merging the bug-fix up to the `master` branch, since resolving conflicts in a single cherry-pick is easier than resolving conflicts during merge. It also allows to rewrite the bug-fix entirely on stable branches.

Review Merge Commits

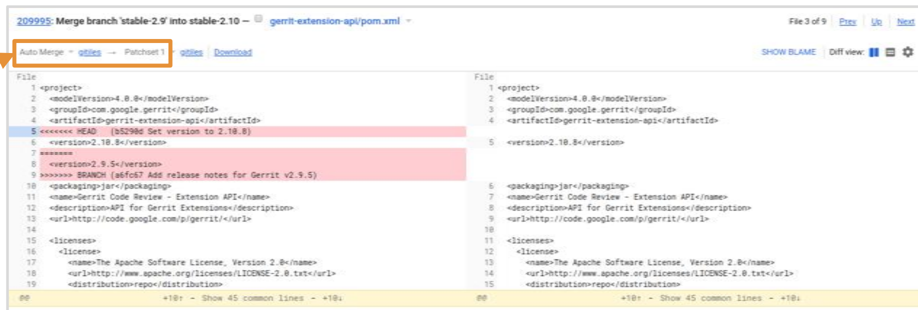


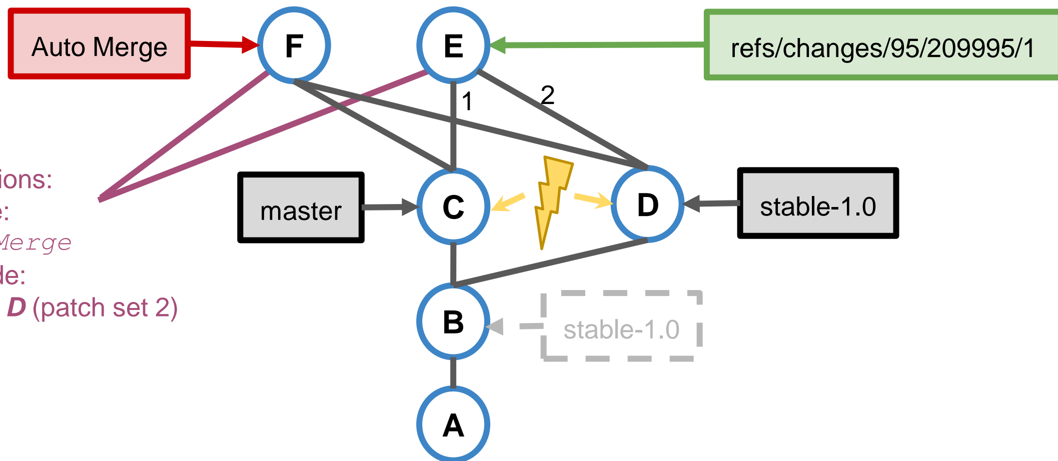


- ***Merge commits*** that are pushed for code review should have the tip of the target branch as first parent.
- By default patch sets of a merge commit are compared against the `Auto Merge` version, but you can also choose to compare against the first or second parent commit.
- The `Auto Merge` version is the result of automatically merging parent 2 into parent 1. This version may contain Git conflict markers.
- The file list of merge commits contains a magic `Merge list` file. This file contains a generated list of commits that are brought into the target branch by this merge. This information is shown as a file so that reviewers can comment on it.

Auto Merge

Patch Set Selector:





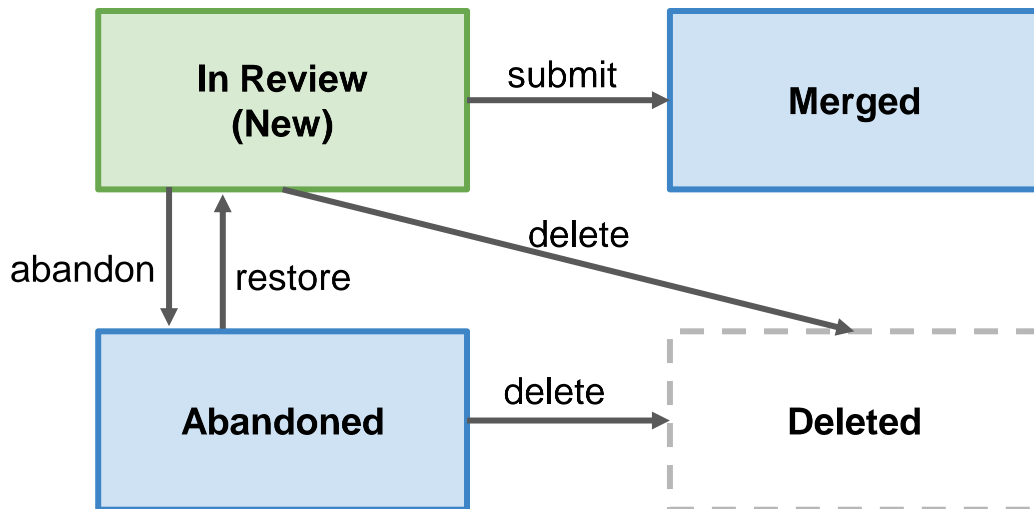
Compared versions:

- left side: `Auto Merge`
- right side: commit ***D*** (patch set 2)

Situation:

- A bug-fix, commit ***D***, was done on the `stable-1.0` branch. To bring the bug-fix into the `master` branch, the `stable-1.0` branch is merged into the `master` branch, which created the merge commit ***E***.
- During the merge there were conflicts which have been resolved in the merge commit ***E***.
- The merge commit has commit ***C*** which is tip of the target branch (`master`) as first parent. The second parent is commit ***D***, which is merged into `master`.
- The merge commit ***E*** was pushed for review, which created a change in Gerrit.
- In Gerrit the merge commit ***E*** is by default compared against the `Auto Merge` version. The `Auto Merge` version for a merge commit is the result of automatically merging parent 2 (commit ***D***) into parent 1 (commit ***D***). Conflicts are not resolved and are represented by Git conflict markers in the files.

Change States



- When a change is uploaded for code review it is in the *In Review* state (aka *New*).
- *Abandoned* means that the change has been given up and nobody is actively working on it.
- Abandoned changes can be **restored** any time.
- If a change gets approved and submitted it is in state *Merged*.
- Merged and abandoned changes are **closed**. This means no new patch sets can be uploaded.
- Changes that have not been submitted yet may be deleted, but this should only be done in exceptional cases (e.g. if the change leaks internal information that cannot be removed otherwise)

Private Changes

Changes can be marked as *private*.

- Private changes are only **visible to the change owner and reviewers** of the change.
- **Use cases:**
  - Backup unfinished work.
  - Collaborate with some reviewers on an experimental change in private.
- **Pitfalls:**
  - If a private change gets merged the corresponding commit gets visible for all users that can access the target branch (but the review discussion stays invisible).
  - If you push a non-private change on top of a private change the commit of the private change gets implicitly visible through the parent relationship of the follow-up change.
  - If you have a series of private changes and share one with reviewers, the reviewers can also see the commits of the predecessor private changes through the commit parent relationship.

- Changes can be marked as private on push:
  `git push origin HEAD:refs/for/master -o private`
- On the change screen the private flag can be toggled to make the change visible to other users.
- The global `View Private Changes` capability can grant users the permission to view all private changes.
- Do not use private changes for security fixes (see next slide).

Making Security Fixes

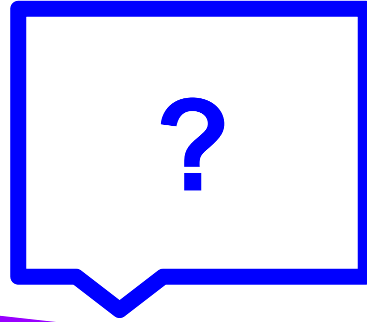
If a **security vulnerability** is discovered you normally want to have an **embargo** about it until fixed releases have been made available. This means you want to develop and review security fixes in private.

If your repository is public or grants broad read access it is recommended to fix security issues in a **copy of your repository which has very restricted read permissions** (e.g. `myproject-security-fixes`). You can then implement, review and submit the security fix in this repository, make and publish a new release and only then integrate the security fix back into the normal (public) repository.

Alternatively you can do the security fix in your normal repository in a branch with restricted read permissions. We don't recommend this because there is a risk of configuring the access rights wrongly and unintentionally grant read access to the wrong people.

Using ***private changes*** for security fixes is **not recommended** due to the pitfalls discussed on the previous slide. Especially you don't want the fix to become visible after submit and before you had a chance to make and publish a new release.

Thank You - Questions?



Go Links (for Googlers only)

TOPIC	GO LINK
Alternative Workflow	go/gerrit-explained@alternative-workflow
Auto Merge	go/gerrit-explained@auto-merge
CHERRY_PICK Submit Strategy	go/gerrit-explained@cherry_pick, go/gerrit-explained@cherry-pick-strategy
Change Ref	go/gerrit-explained@change-ref
Change States	go/gerrit-explained@change-states
Change-Id	go/gerrit-explained@change-id
Change	go/gerrit-explained@change
Code Review Benefits	go/gerrit-explained@code-review-benefits
Comparing Patch Sets after Rebase	go/gerrit-explained@comparing-patch-sets-after-rebase

TOPIC	GO LINK
Comparing Patch Sets	go/gerrit-explained@comparing-patch-sets
Conflict Resolution	go/gerrit-explained@conflict-resolution
Gerrit Concepts	go/gerrit-explained@concepts, go/gerrit-explained@gerrit-concepts
Gerrit	go/gerrit-explained@gerrit
Hashtags	go/gerrit-explained@hashtags
MERGE_ALWAYS Submit Strategy	go/gerrit-explained@merge_always, go/gerrit-explained@merge-always
MERGE_IF_NECESSARY Submit Strategy	go/gerrit-explained@merge_if_necessary, go/gerrit-explained@merge-if-necessary
Modern Code Review	go/gerrit-explained@modern-code-review
Not Gerrit	go/gerrit-explained@not-gerrit
Patch Set	go/gerrit-explained@patch-set

Go Links (for Googlers only)

TOPIC	GO LINK
Private Changes	go/gerrit-explained@private-changes
Push for Code Review	go/gerrit-explained@push-for-code-review, go/gerrit-explained@refs-for
Push for Code Review (Case 1)	go/gerrit-explained@push-for-code-review-case-1
Push for Code Review (Case 2)	go/gerrit-explained@push-for-code-review-case-2
Push for Code Review (Case 3)	go/gerrit-explained@push-for-code-review-case-3
Push new Patch Set	go/gerrit-explained@push-new-patch-set
REBASE_ALWYAS Submit Strategy	go/gerrit-explained@rebase_always, go/gerrit-explained@rebase-always
REBASE_IF_NECESSARY Submit Strategy	go/gerrit-explained@rebase_if_necessary, go/gerrit-explained@rebase-if-necessary

TOPIC	GO LINK
Repository Config	go/gerrit-explained@repo-config, go/gerrit-explained@repository-config, go/gerrit-explained@project-config
Review Merge Commits	go/gerrit-explained@review-merge-commits, go/gerrit-explained@merge-commits
Review and Vote	go/gerrit-explained@review-and-vote
Review new Patch Set	go/gerrit-explained@review-patch-set
Security Fixes	go/gerrit-explained@security-fixes
Standard Workflow	go/gerrit-explained@workflow, go/gerrit-explained@standard-workflow
Sticky Votes	go/gerrit-explained@sticky-votes, go/gerrit-explained@sticky-approvals
Submit Strategy / Submit Type	go/gerrit-explained@submit-strategy, go/gerrit-explained@submit-type

Go Links (for Googlers only)

TOPIC	*GO LINK*
Submit	go/gerrit-explained@submit
Trivial Rebase	go/gerrit-explained@trivial-rebase
Voting	go/gerrit-explained@voting
Working with Change Series	go/gerrit-explained@working-with-change-series, go/gerrit-explained@change-series, go/gerrit-explained@workflow-change-series
Working with Stable Branches	go/gerrit-explained@working-with-stable-branches, go/gerrit-explained@stable-branches, go/gerrit-explained@workflow-stable-branches
Working with Topic	go/gerrit-explained@working-with-topic, go/gerrit-explained@topics, go/gerrit-explained@workflow-topics

License

This presentation is part of the Gerrit Code Review project
and is published under the Apache License 2.0.