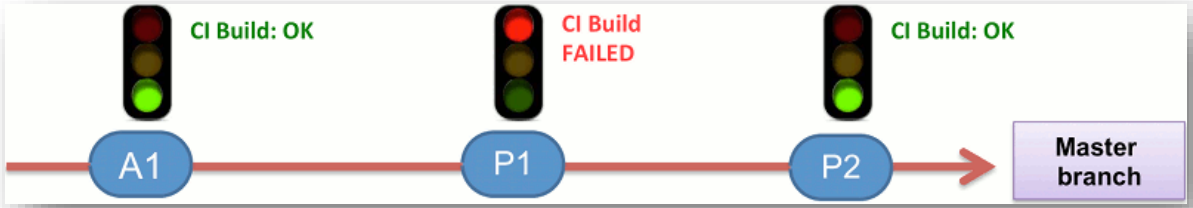
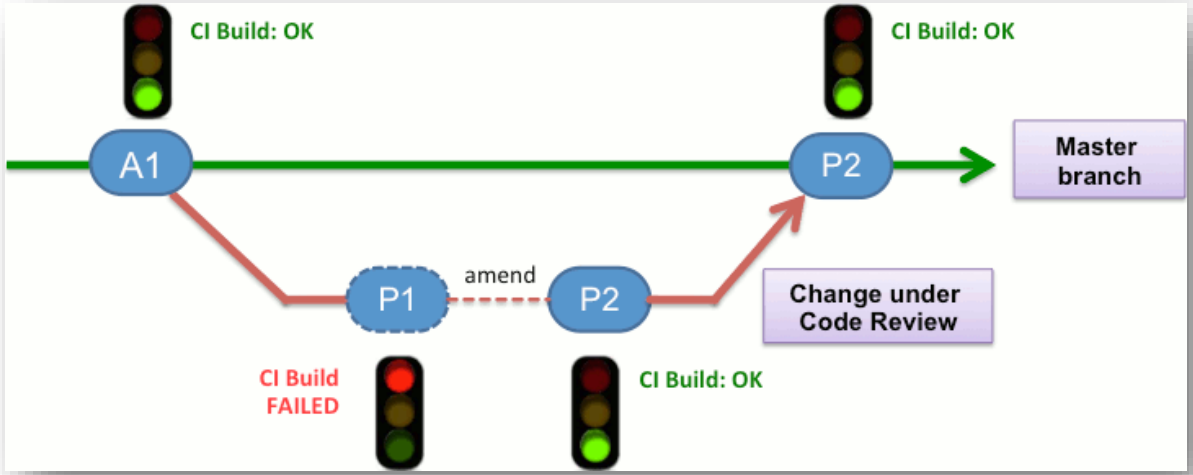
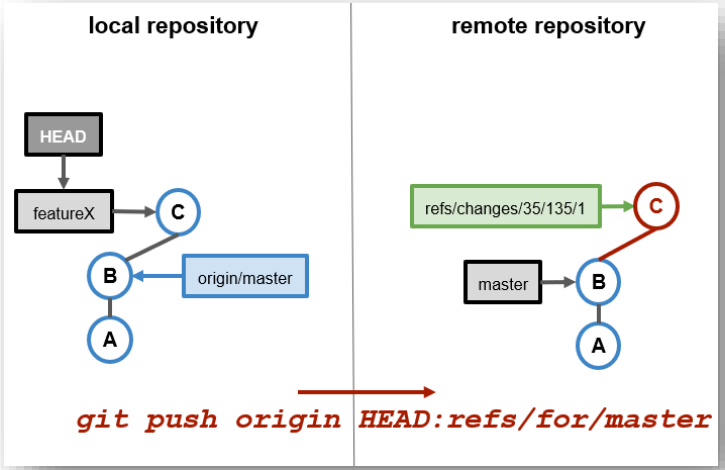
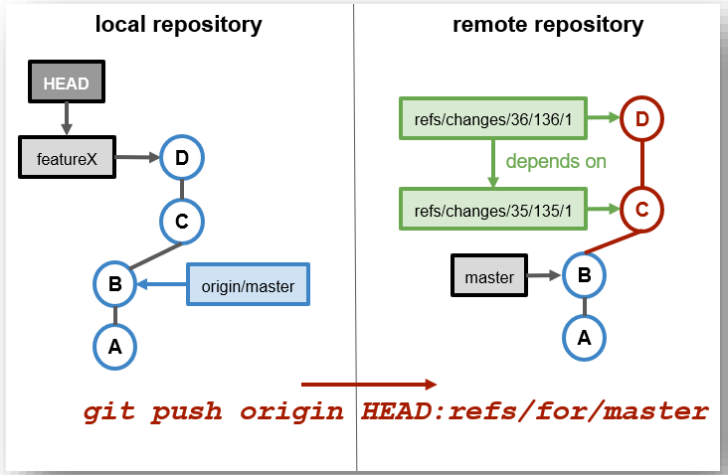
Code Review #3

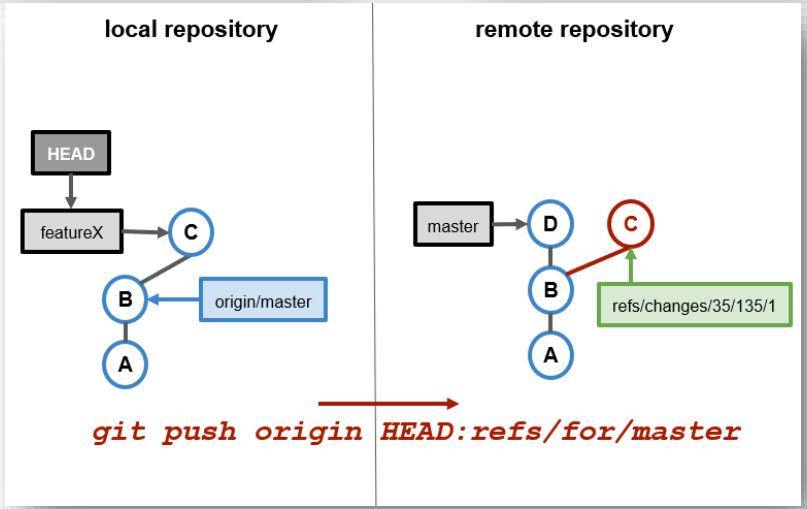
Majid Babaei


CI Build: OK
CI Build
FAILED
CI Build: OK
A1
P1
P2
Master
branch


CI Build: OK
CI Build: OK
A1
P2
Master
branch
amend
P1
P2
Change under
Code Review
CI Build
FAILED
CI Build: OK

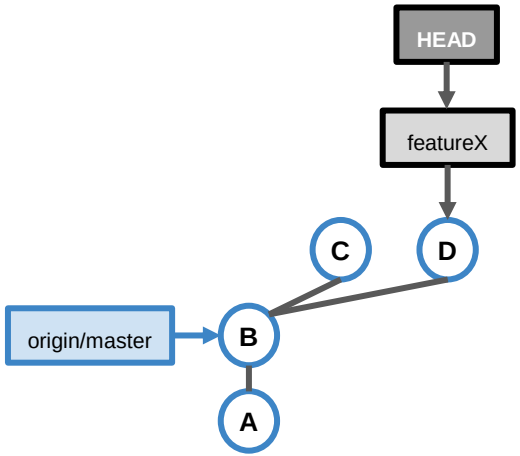

local repository
remote repository
HEAD
featureX
C
B
origin/master
A
refs/changes/35/135/1
C
master
B
A
git push origin HEAD:refs/for/master


local repository
remote repository
HEAD
featureX
D
C
B
origin/master
A
refs/changes/36/136/1
D
depends on
refs/changes/35/135/1
C
master
B
A
git push origin HEAD:refs/for/master

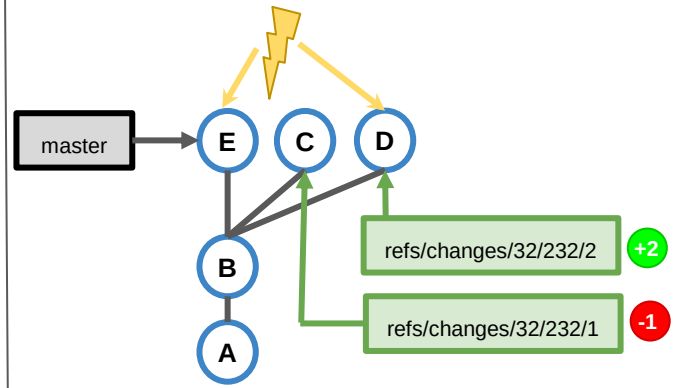

local repository
remote repository
HEAD
featureX
C
B
origin/master
A
master
D
C
B
refs/changes/35/135/1
A
git push origin HEAD:refs/for/master

Conflict Resolution

local repository



remote repository

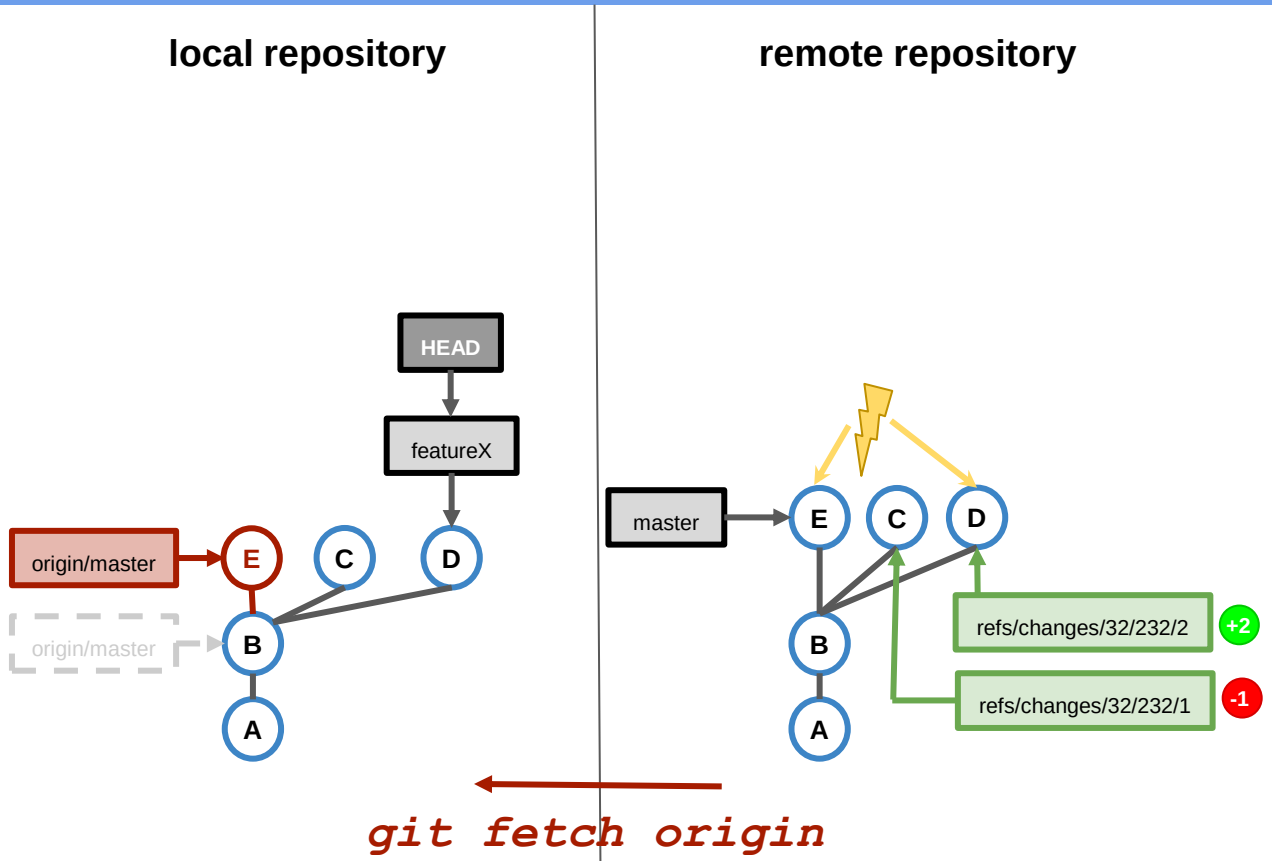


Situation:

- 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?

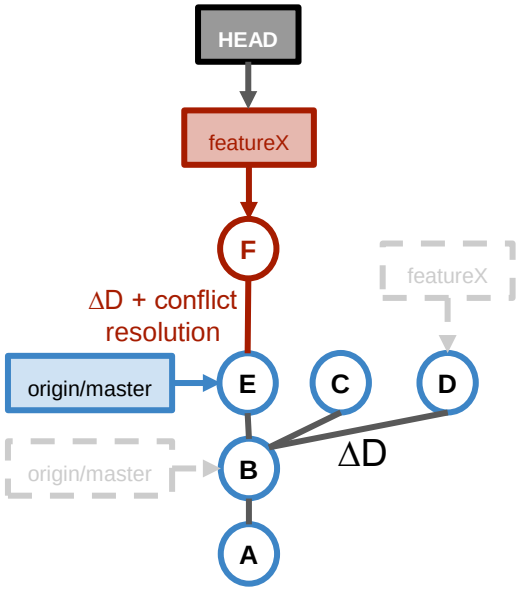
Conflict Resolution - Rebase Change



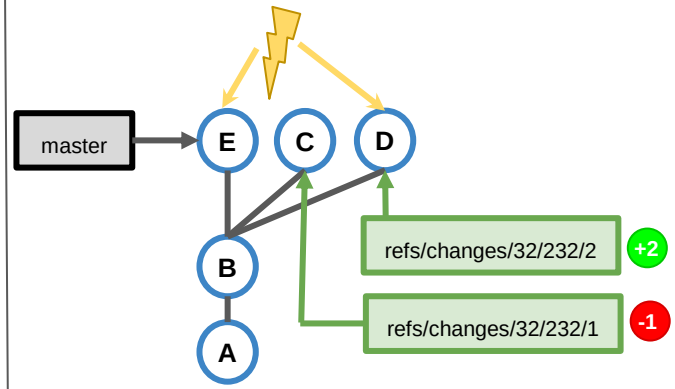
1. *git fetch origin*:
   - Brings commit ***E*** into the local repository and updates the remote tracking branch.

Conflict Resolution - Rebase Change

local repository

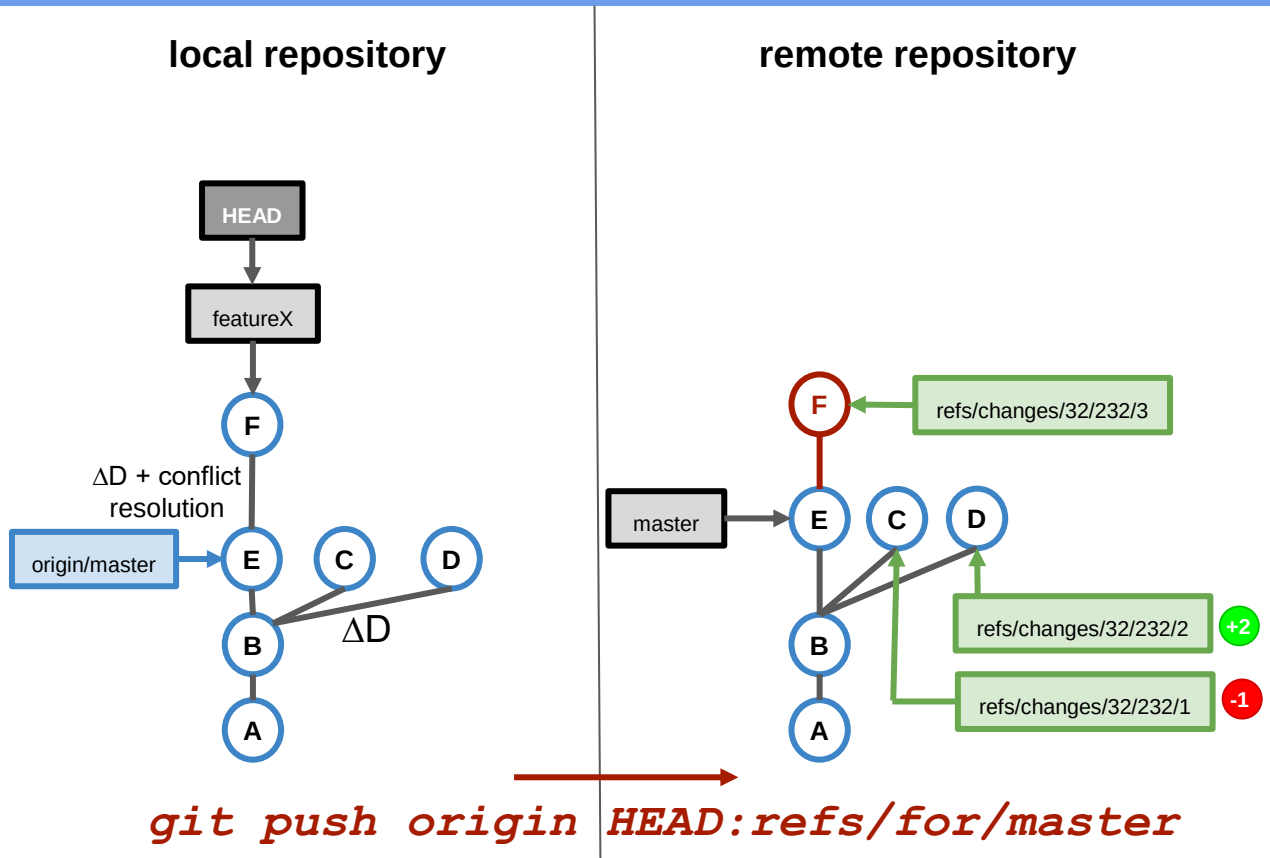


remote repository



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.

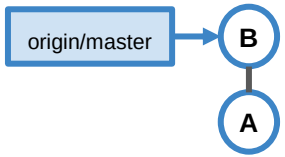
Conflict Resolution - Rebase Change



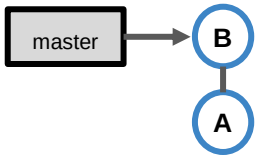
1. *git fetch origin*:
   - Brings commit E into the local repository and updates the remote tracking branch.
2. *git rebase origin/master*:
   - Rebases the *featureX* branch (commit ***D***) onto the remote tracking branch *origin/master*. During the rebase the conflicts need to be resolved.
   - On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

Standard Workflow (Summary)

local repository



remote repository

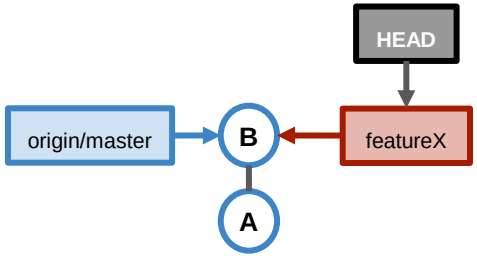


Situation:

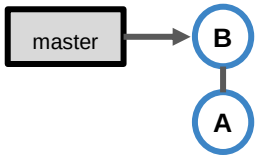
- The `master` branch in the remote repository contains two commits, commit ***A*** and commit ***B***. Both commits have been fetched into the local repository.

Standard Workflow (Summary)

local repository



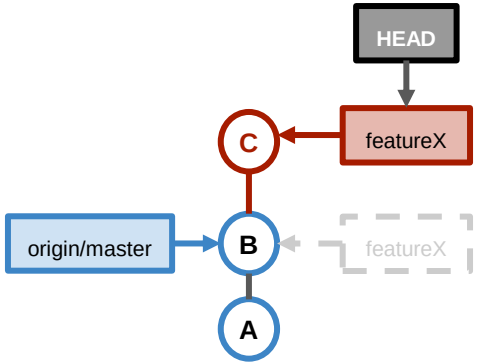
remote repository



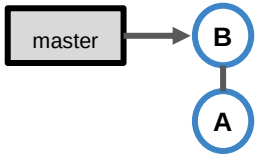
1. Create and checkout a local feature branch which is based on *origin/master*:
   git checkout -b featureX origin/master

Standard Workflow (Summary)

local repository



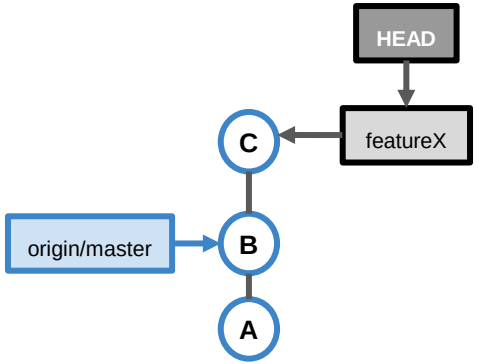
remote repository



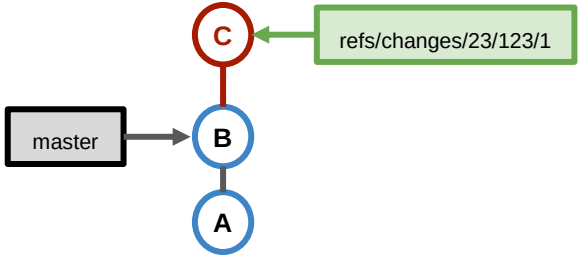
1. Create and checkout a local feature branch which is based on *origin/master*:
   `git checkout -b featureX origin/master`
2. Make a new commit that implements the feature.

Standard Workflow (Summary)

local repository



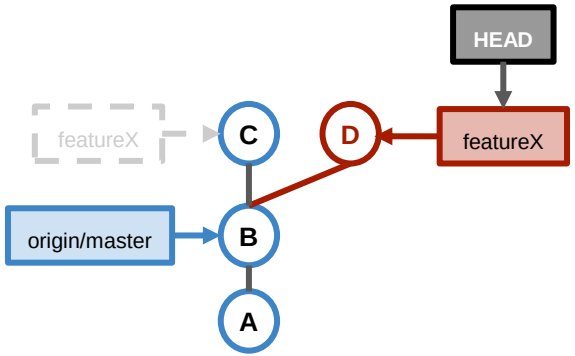
remote repository



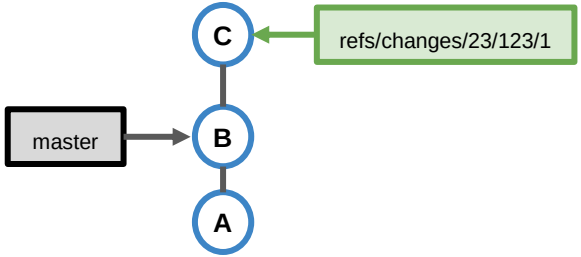
1. Create and checkout a local feature branch which is based on *origin/master*:
   ```
   git checkout -b featureX origin/master
   ```
2. Make a new commit that implements the feature.
3. Push the commit for code review:
   ```
   git push origin HEAD:refs/for/master
   ```

Standard Workflow (Summary)

local repository



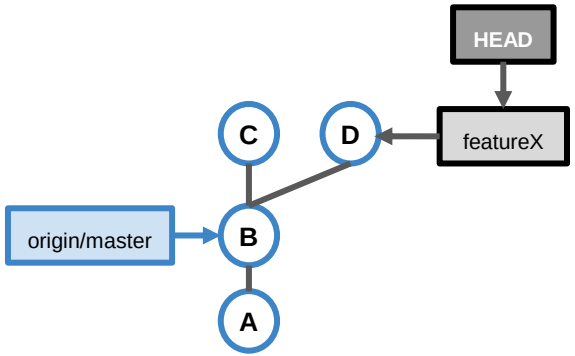
remote repository



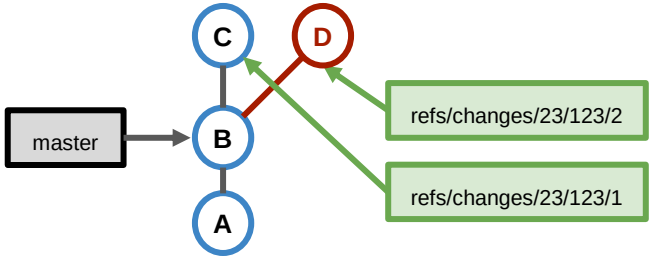
1. Create and checkout a local feature branch which is based on *origin/master*:
   git checkout -b featureX origin/master
2. Make a new commit that implements the feature.
3. Push the commit for code review:
   git push origin HEAD:refs/for/master
4. If rework is needed:
   a. Checkout the *featureX*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

Standard Workflow (Summary)

local repository



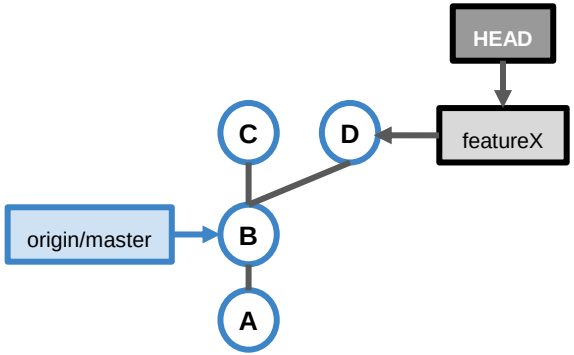
remote repository



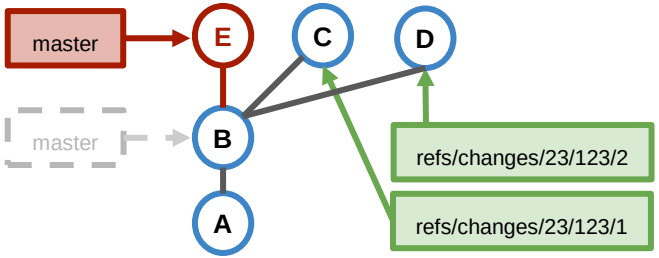
1. Create and checkout a local feature branch which is based on *origin/master*:
   git checkout -b featureX origin/master
2. Make a new commit that implements the feature.
3. Push the commit for code review:
   git push origin HEAD:refs/for/master
4. If rework is needed:
   a. Checkout the *featureX*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

Standard Workflow (Summary)

local repository



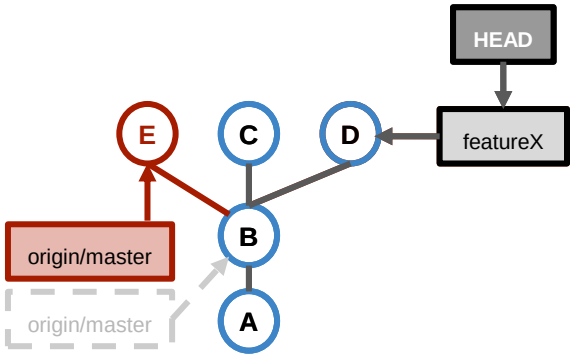
remote repository



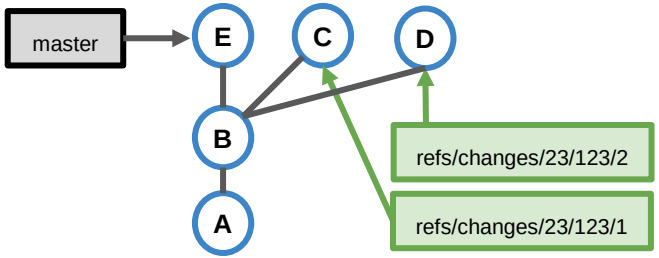
1. ...
2. Make a new commit that implements the feature.
3. Push the commit for code review: `git push origin HEAD:refs/for/master`
4. If rework is needed:
   a. Checkout the `featureX` branch and amend the commit.
   b. Push the new commit for review to create a second patch set
5. If rebase is needed

Standard Workflow (Summary)

local repository



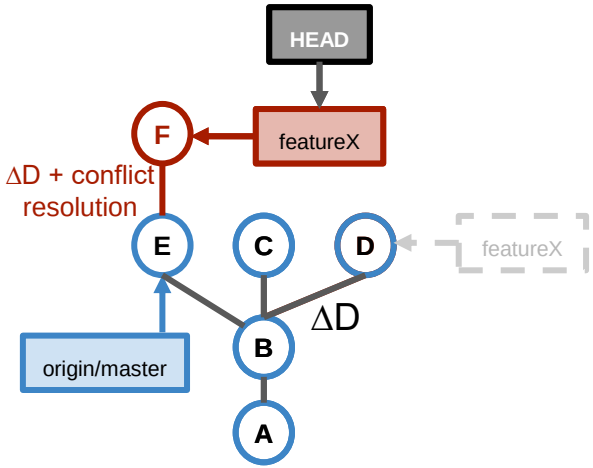
remote repository



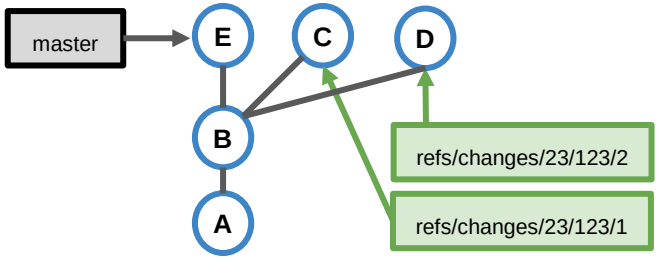
1. ...
2. Make a new commit that implements the feature.
3. Push the commit for code review: `git push origin HEAD:refs/for/master`
4. If rework is needed:
   a. Checkout the *featureX* branch and amend the commit.
   b. Push the new commit for review to create a second patch set
5. If rebase is needed: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

Standard Workflow (Summary)

local repository



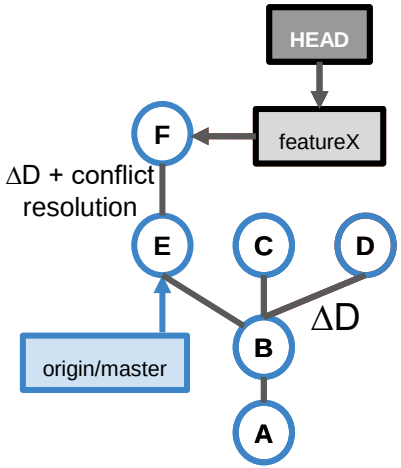
remote repository



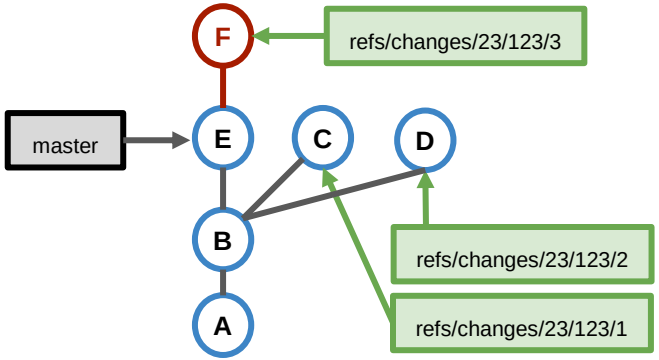
1. ...
2. Make a new commit that implements the feature.
3. Push the commit for code review: `git push origin HEAD:refs/for/master`
4. If rework is needed:
   a. Checkout the `featureX` branch and amend the commit.
   b. Push the new commit for review to create a second patch set
5. If rebase is needed: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

Standard Workflow (Summary)

local repository



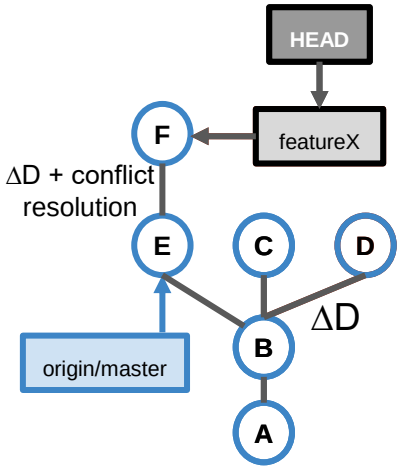
remote repository



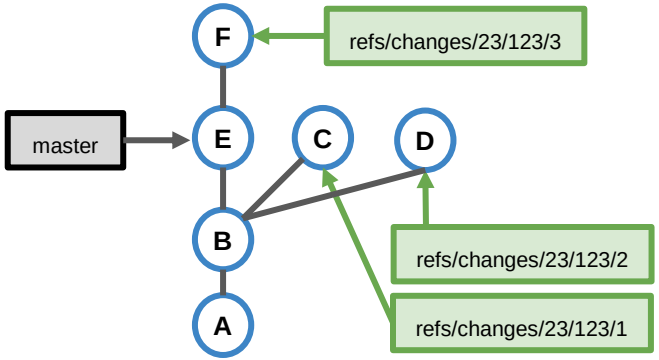
1. ...
2. …
3. ...
4. If rework is needed:
   a. Checkout the *featureX* branch and amend the commit.
   b. Push the new commit for review to create a second patch set
5. If rebase is needed:
   a. Fetch the updates from the remote repository
   b. Checkout the *featureX* branch and rebase it onto *origin/master*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

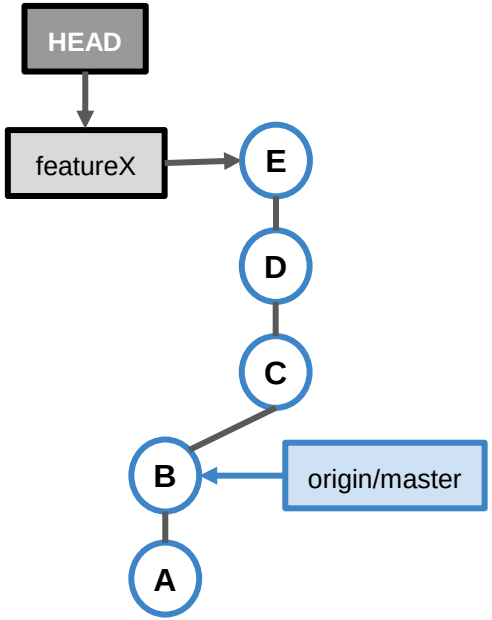
Alternative Workflow

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

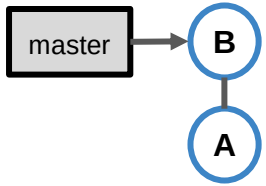
- After pushing a commit for code review it is available on Gerrit and it can be fetched from there whenever the change needs to be reworked or rebased (copy the fetch command from the change screen).
- Working with detached HEAD has the risk of losing reference to commits when you checkout another state 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

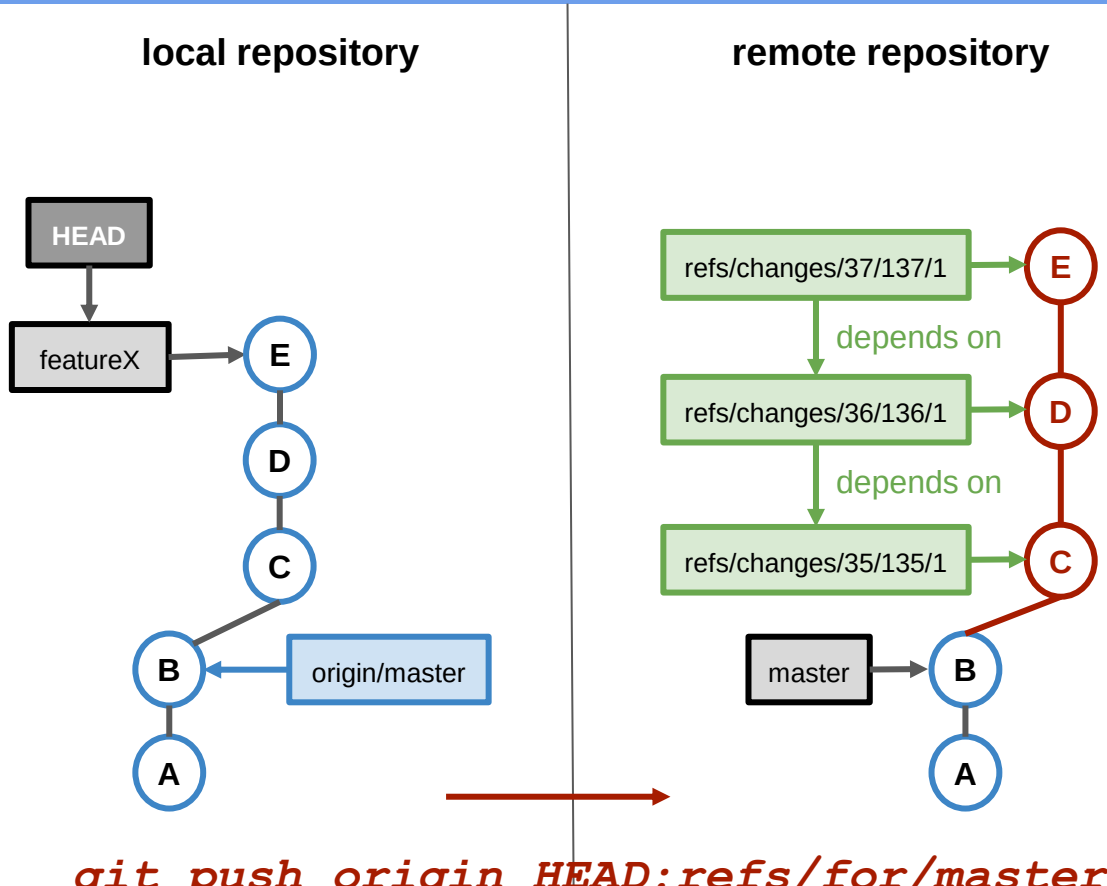


It is common that features are implemented by **multiple self-contained commits** that are based on each other.

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



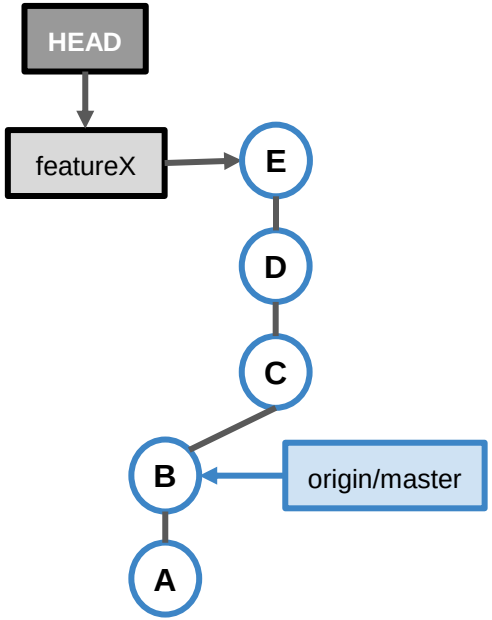
All commits in the *featureX* branch can be pushed for code review by a single *git push* command.

For each of the pushed commits a change is created.

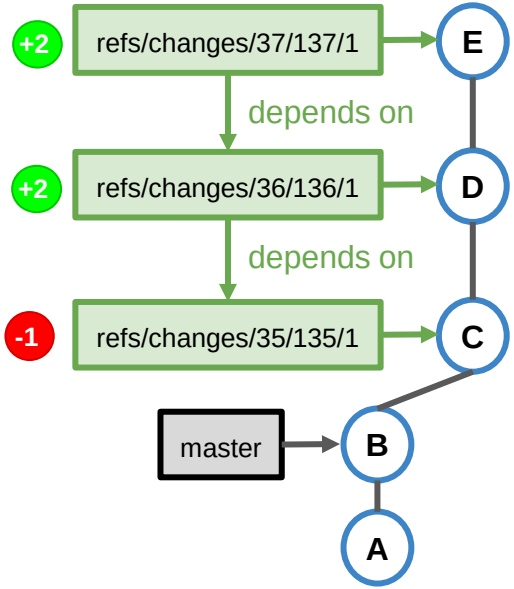
The changes depend on each other the same way as the commits depend on each other.

Working with Change Series

local repository



remote repository



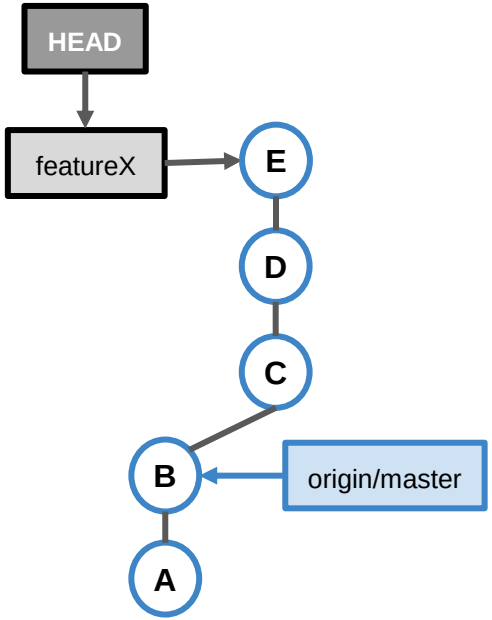
If a change in the series needs to be reworked:

- checkout the *featureX* branch
- 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
```

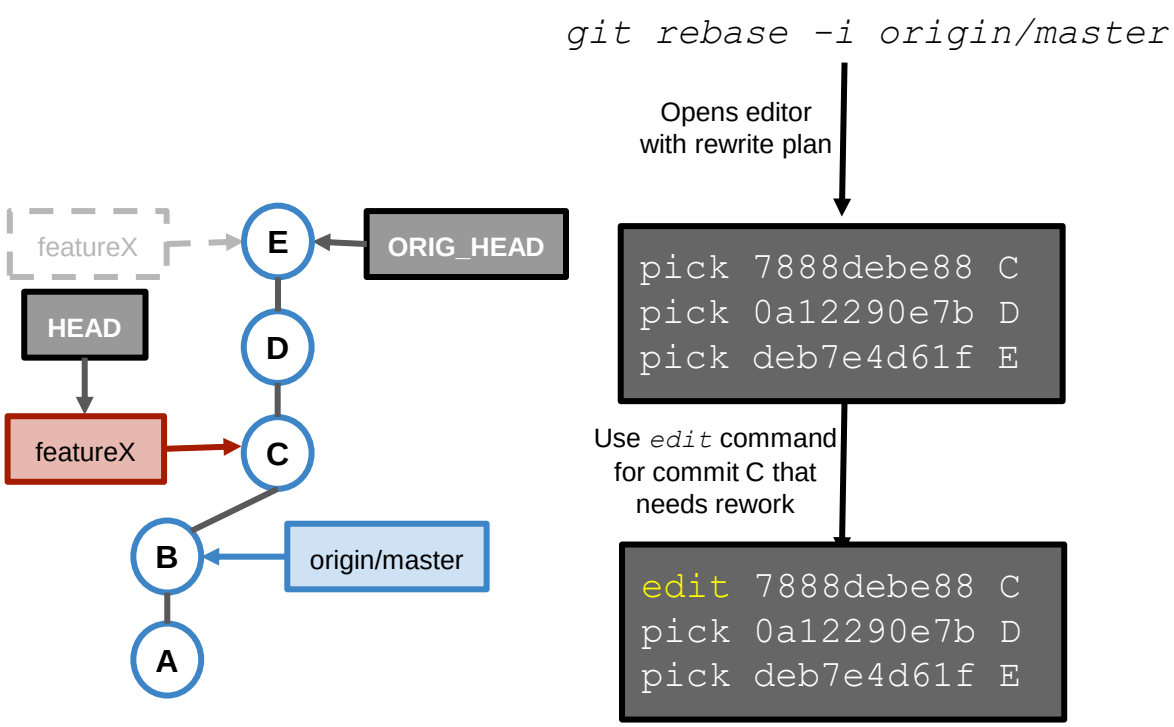
If a change in the series needs to be reworked:

- checkout the *featureX* branch
- use interactive rebase to rewrite the commits in the *featureX* branch:

git rebase -i origin/master

Working with Change Series

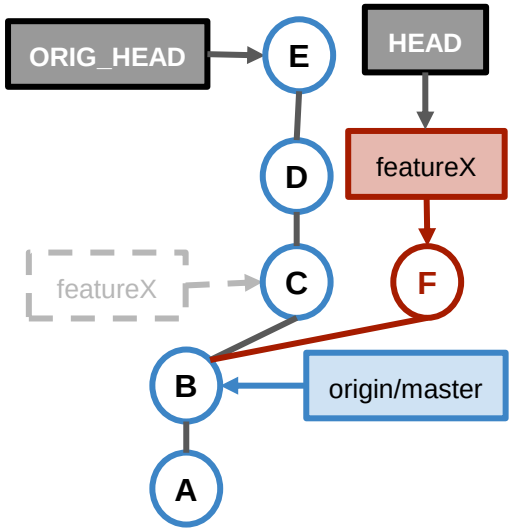
local repository



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

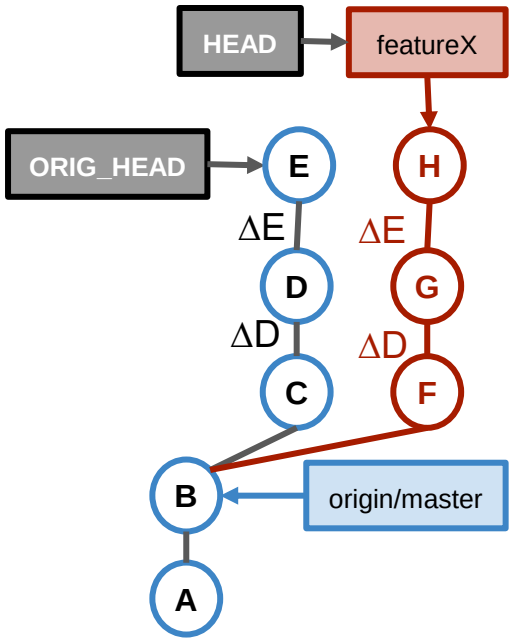
```
edit 7888debe88 C
pick 0a12290e7b D
pick deb7e4d61f E
```

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
```

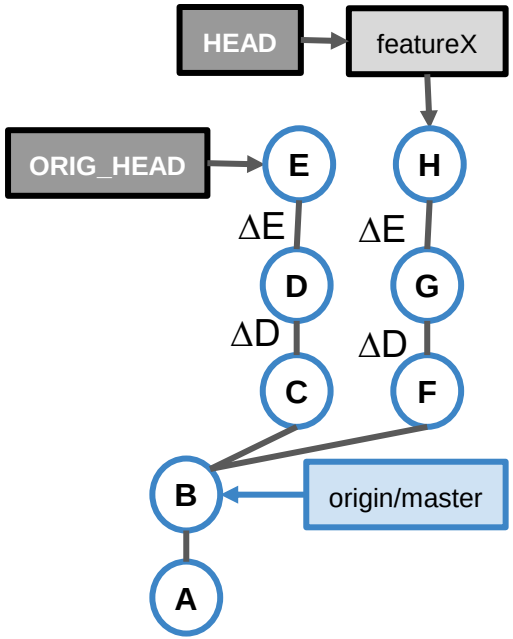
This rewinds the *featureX* branch to commit ***C*** where the interactive rebase stops so that it can be edited:

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

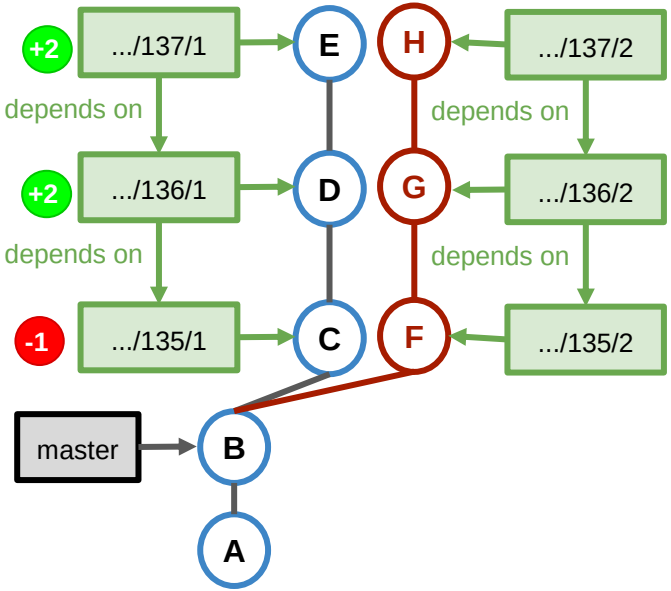
2. Continue the rebase by *git rebase --continue*, this will ***cherry-pick*** commit ***D*** and commit ***E*** on top of the amended commit ***F***.

Working with Change Series

local repository



remote repository



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 (Note: the `Change-Ids` in the commits 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. This is the case if the change was rebased onto a different parent and that rebase did not require git to perform any conflict resolution, or if the parent did not change at all.

- ***review labels*** can be defined and configured per repository
- If a vote or approval is copied forward to new patch sets it is called ***sticky***
- `copyMinScore` must be set to enable ***veto votes***

Sticky Votes, Continue…

- `copyAllScoresIfNoCodeChange`:
  The vote is copied if the new patch set only modified the commit message.

- `copyAllScoresIfNoChange`:
  when a new patch set is uploaded that has the same parent tree, code delta, and commit message as the previous patch set. This means that only the patch set SHA-1 is different. This can be used to enable sticky approvals, reducing turn-around for this special case.

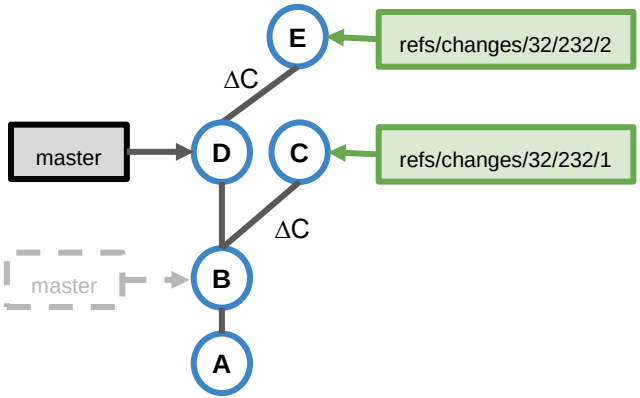
- `copyAllScoresOnMergeFirstParentUpdate`:
  This policy is useful if you don't want to trigger human verification again if your target branch moved on but the feature branch being merged into the target branch. It only applies if the patch set is a merge commit.

- 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.

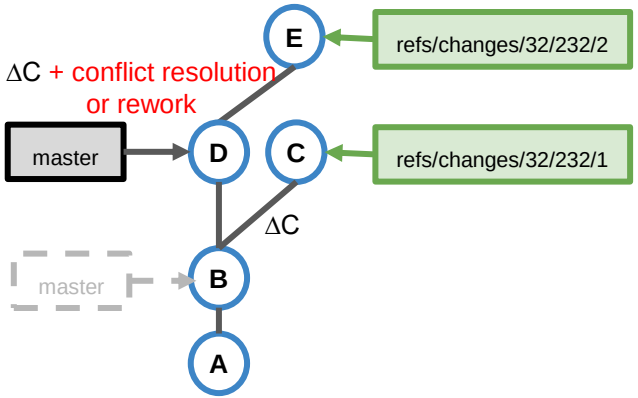
- In the API the kind of changes that are done by a patch set in comparison to the predecessor patch set is exposed as ***change kind***.

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***.

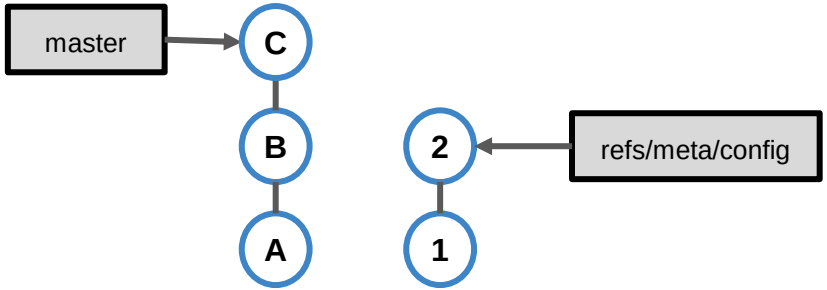
A new patch set is considered as ***trivial rebase*** if the commit message is the same as in the previous patch set and if it has the same code delta as the previous patch set.

Repository Configuration

Gerrit stores a configuration per repository.

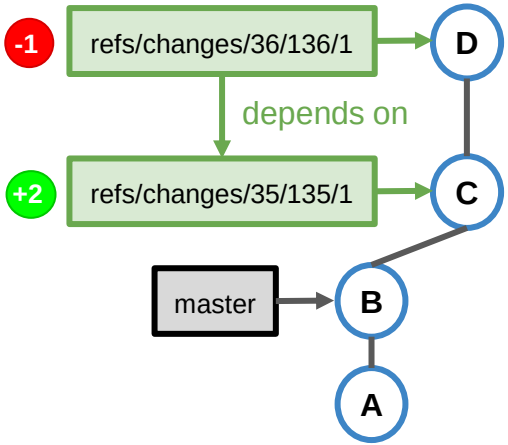
This configuration contains:

- 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

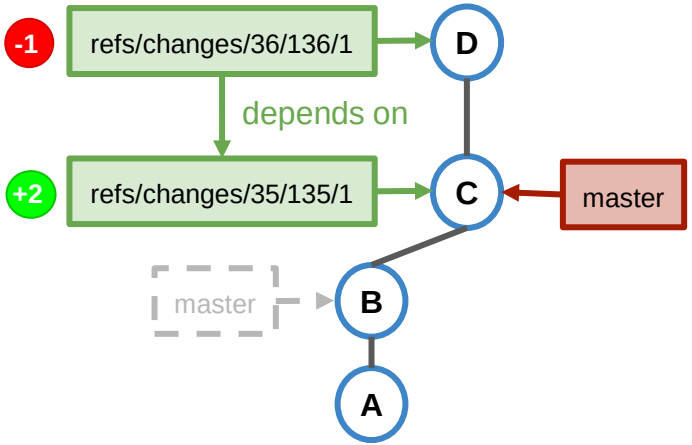


Situation:

- 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)?

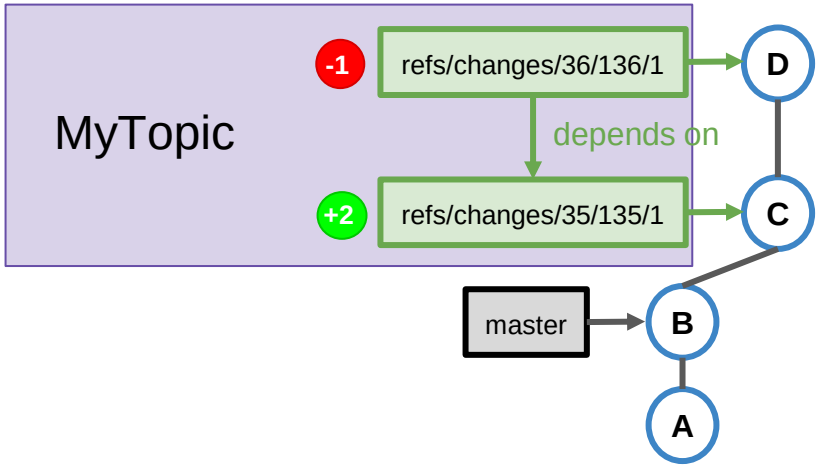
Working with Topics



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



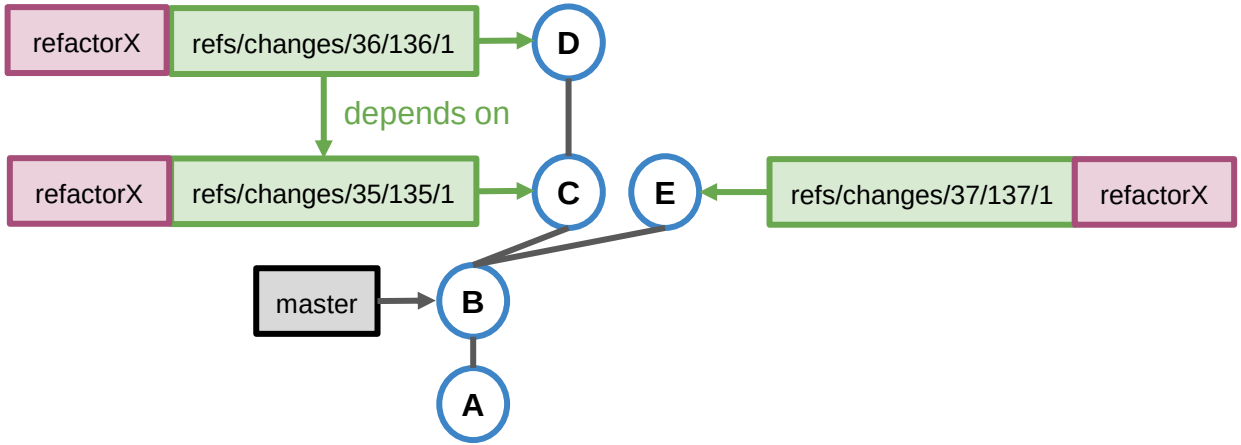
Changes can be grouped by ***topics*** to enforce that they can only be **submitted together**. If both changes share the same topic, the topic can only be submitted when both changes are approved and submittable.

- 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 can be set on push:

```
git push origin HEAD:refs/for/master -o
topic=MyTopic
```

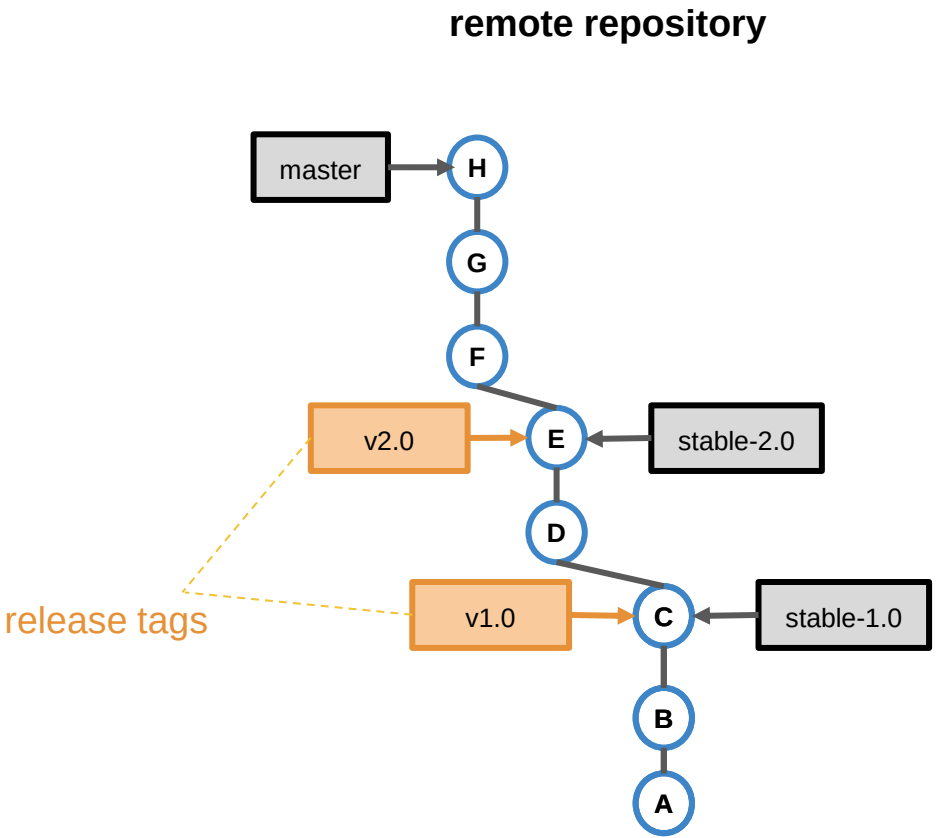
Hashtags



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



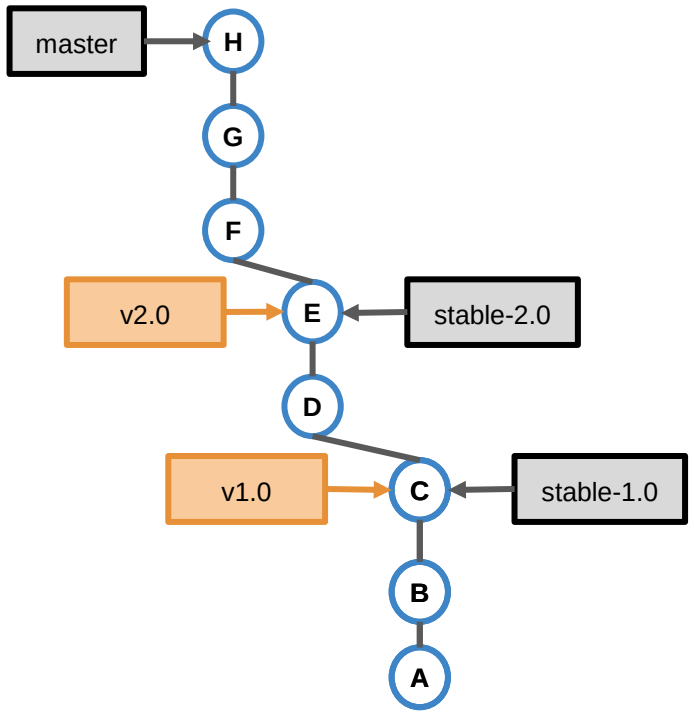
If **several versions of a project** need to be maintained it's a common practise to have one (central) branch for each major project version:

- 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.

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

Working with Stable Branches

remote repository

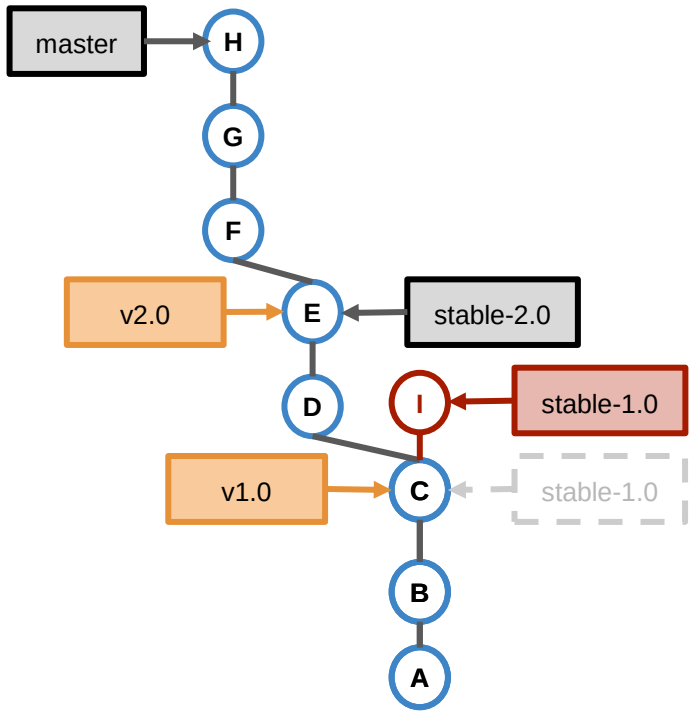


There are 2 possibilities to do bug-fixes:

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

Working with Stable Branches - Option 1

remote repository

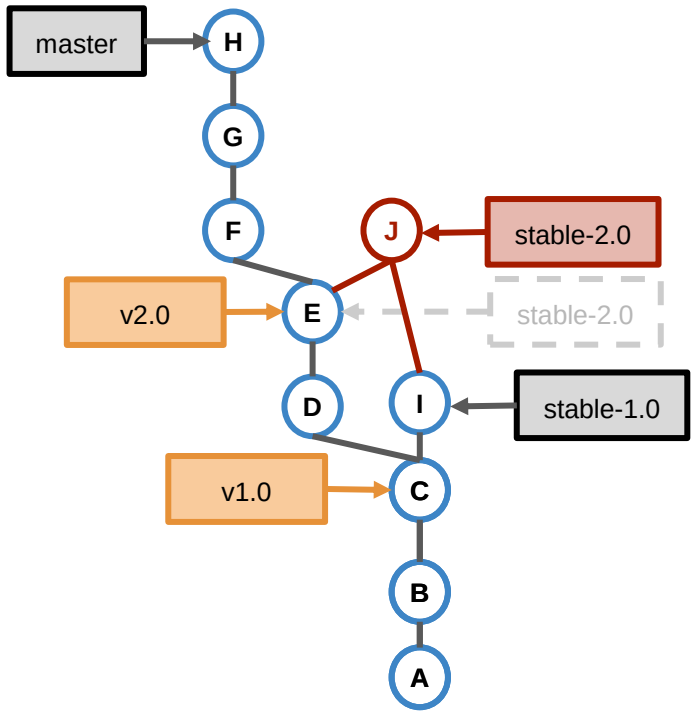


There are 2 possibilities to do bug-fixes:

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

Working with Stable Branches - Option 1

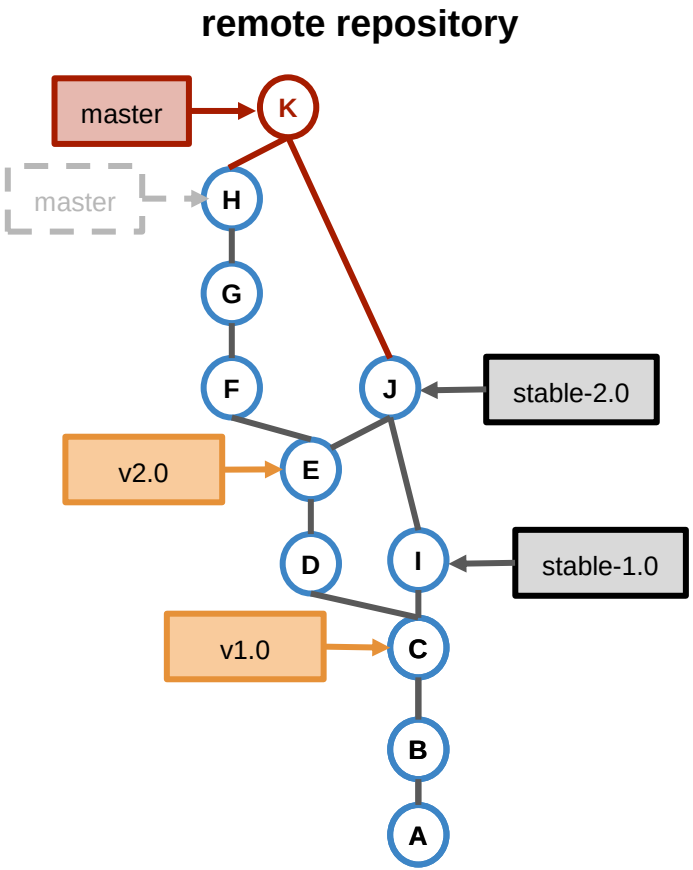
remote repository



There are 2 possibilities to do bug-fixes:

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

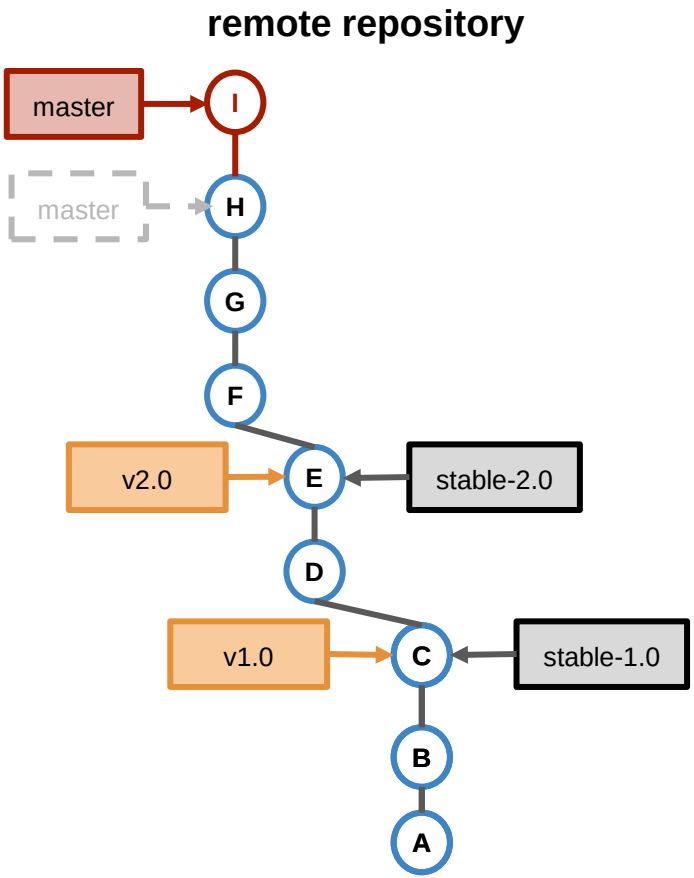
Working with Stable Branches - Option 1



There are 2 possibilities to do bug-fixes:

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

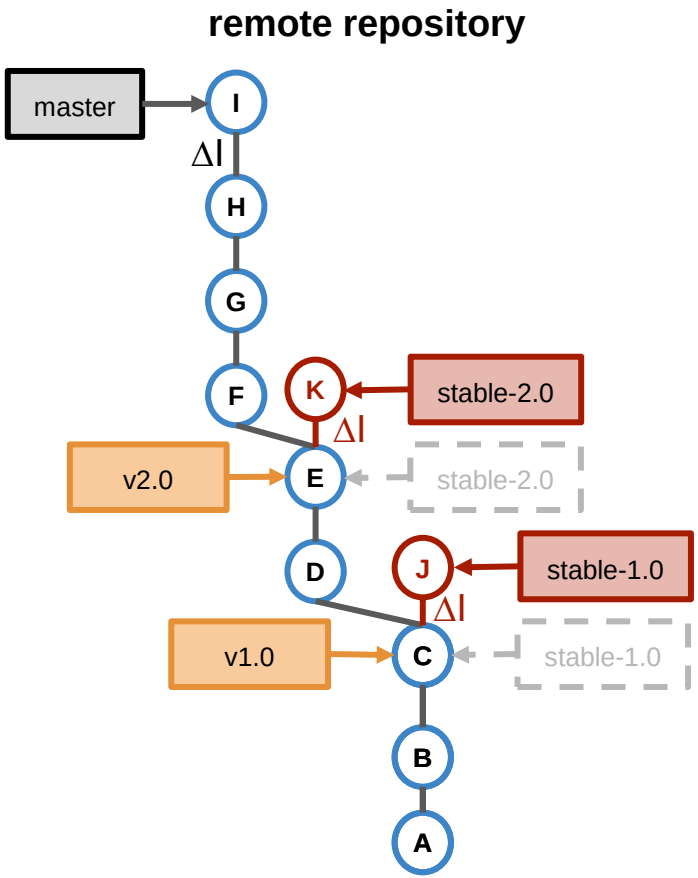
Working with Stable Branches - Option 2



There are 2 possibilities to do bug-fixes:

1. Do the bug-fix in the oldest stable branch that is affected 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.

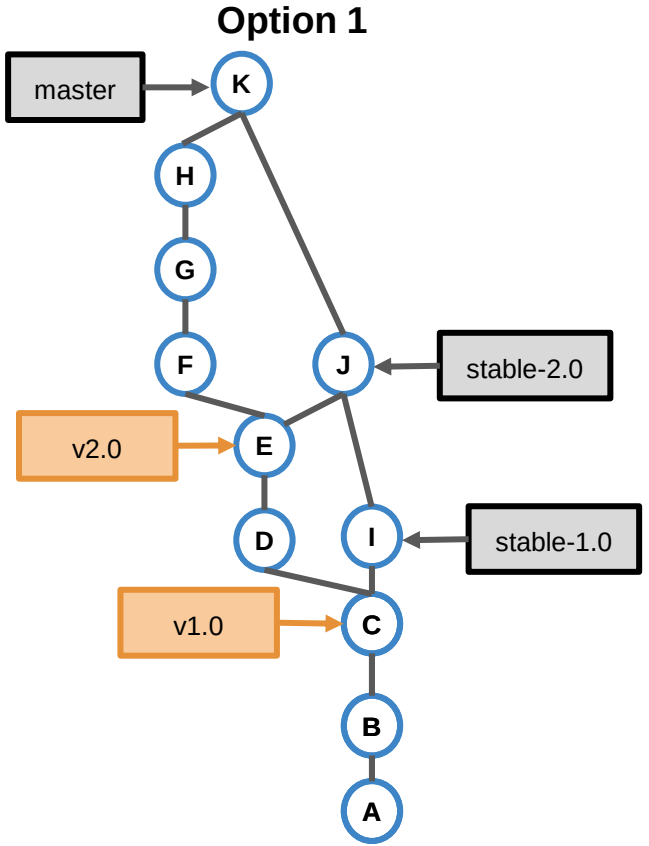
Working with Stable Branches - Option 2

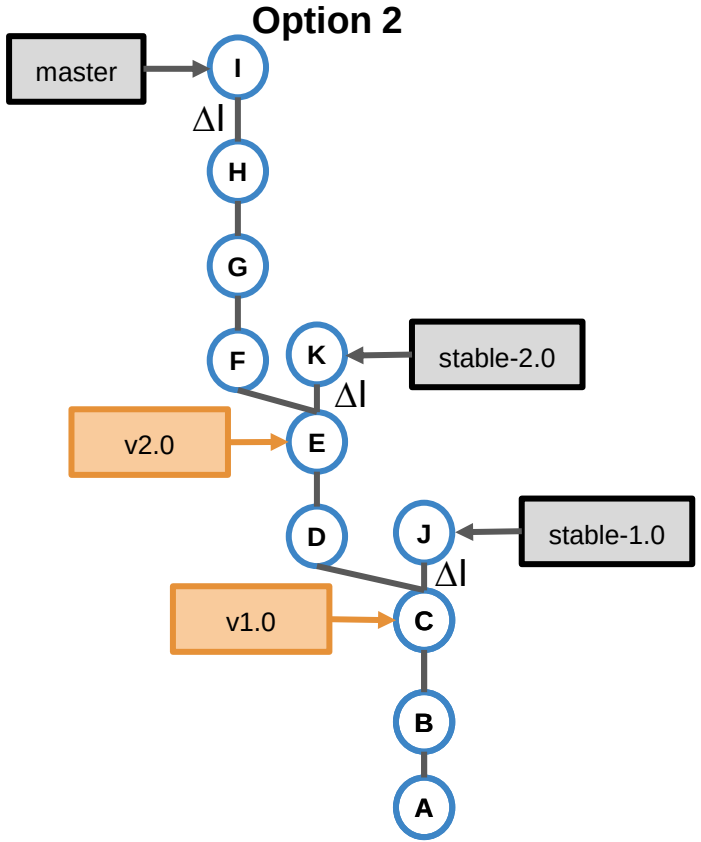


There are 2 possibilities to do bug-fixes:

1. Do the bug-fix in the oldest stable branch that is affected 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



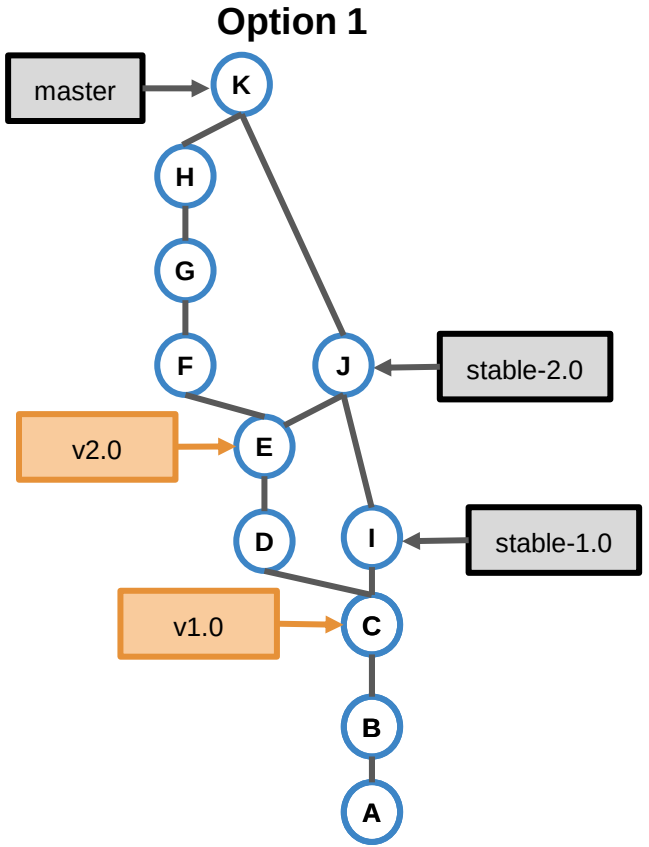


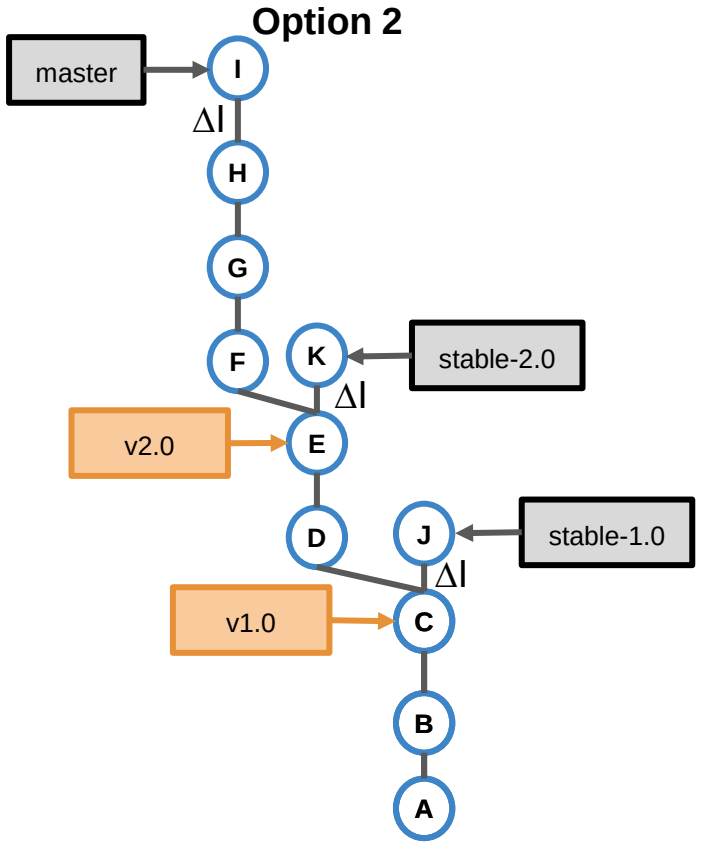
There are 2 possibilities to do bug-fixes:

1. Do the bug-fix in the oldest stable branch that is affected 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



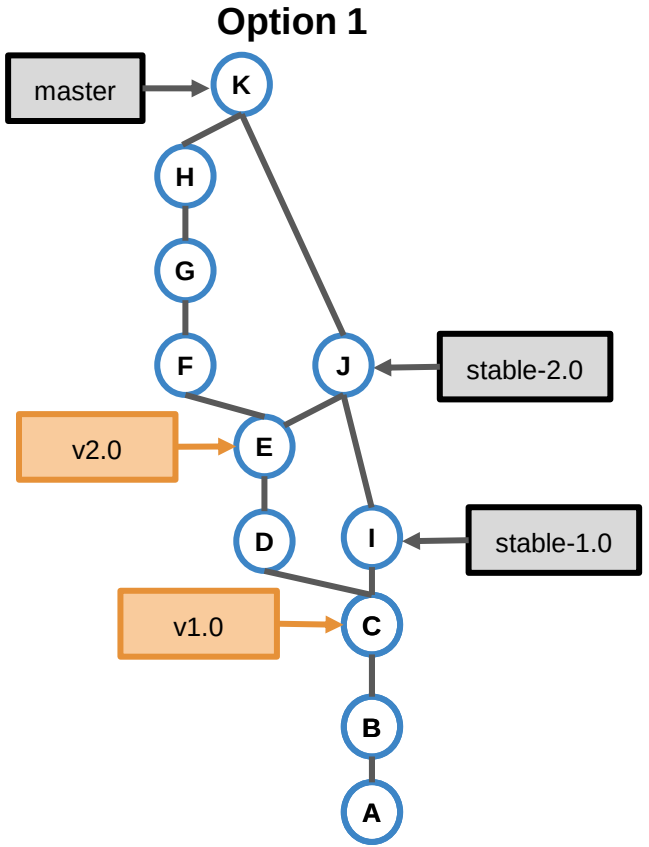


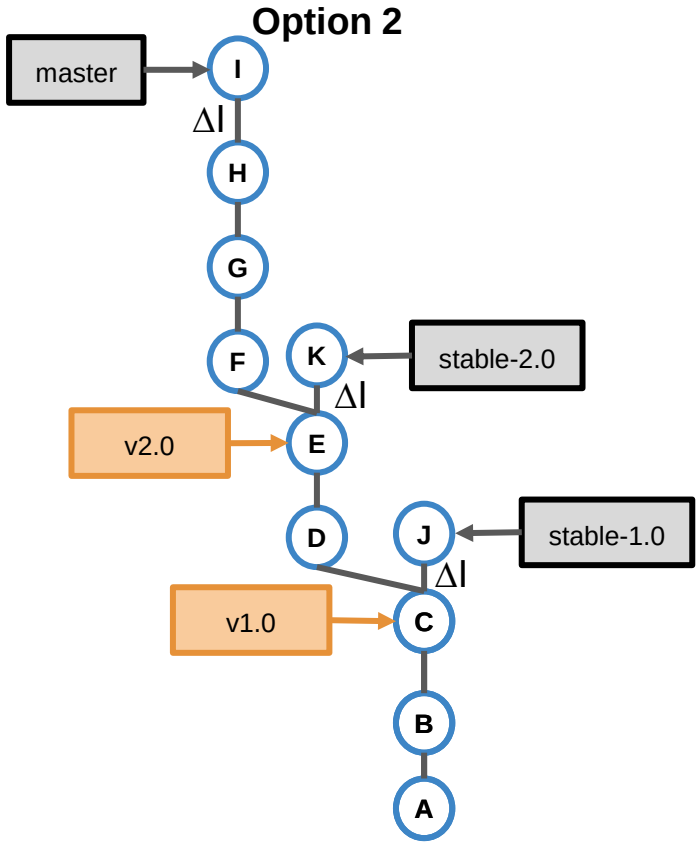
Option 1 is generally preferred because there is exactly one commit that implements the bug-fix (commit *I*).

The rest of the merges can be done automatically by Git

Option 1 assumes that everything that is done in a stable branches should also be applied to `master` and newer stable branches.

Working with Stable Branches





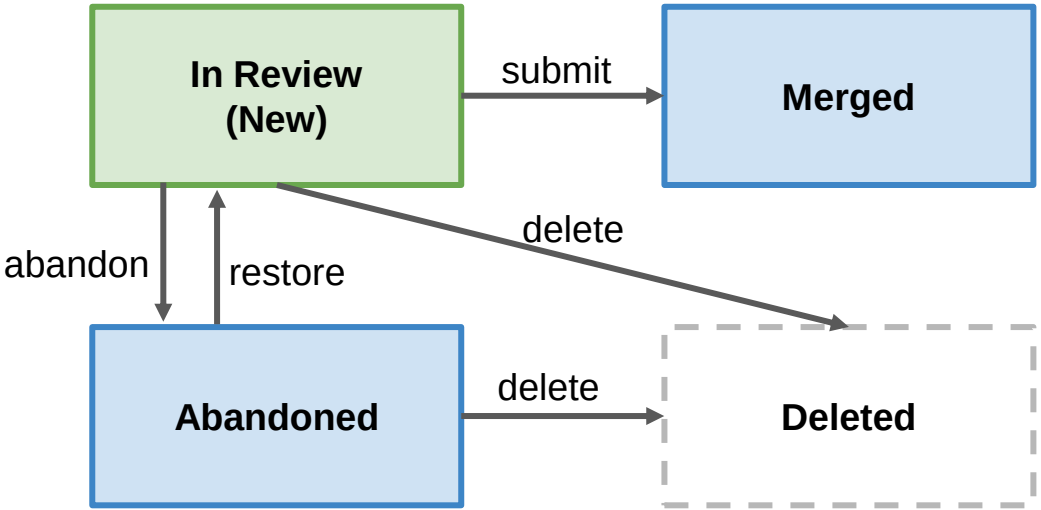
Option 2 there are multiple commits that implement the bug-fix (commits ***I***, ***J***, ***K***) but from the version graph you can't see that they do the same modifications.

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`.

Resolving conflicts in a single cherry-pick is easier than resolving conflicts during merge.

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.

Change States



open closed

- 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 and the private flag from the change is automatically removed.
  - 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.

Alternatively you can do the security fix in your normal repository in a branch with restricted read permissions.

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.