Git General Training

Git Best Practices

Git –> manage AI tips

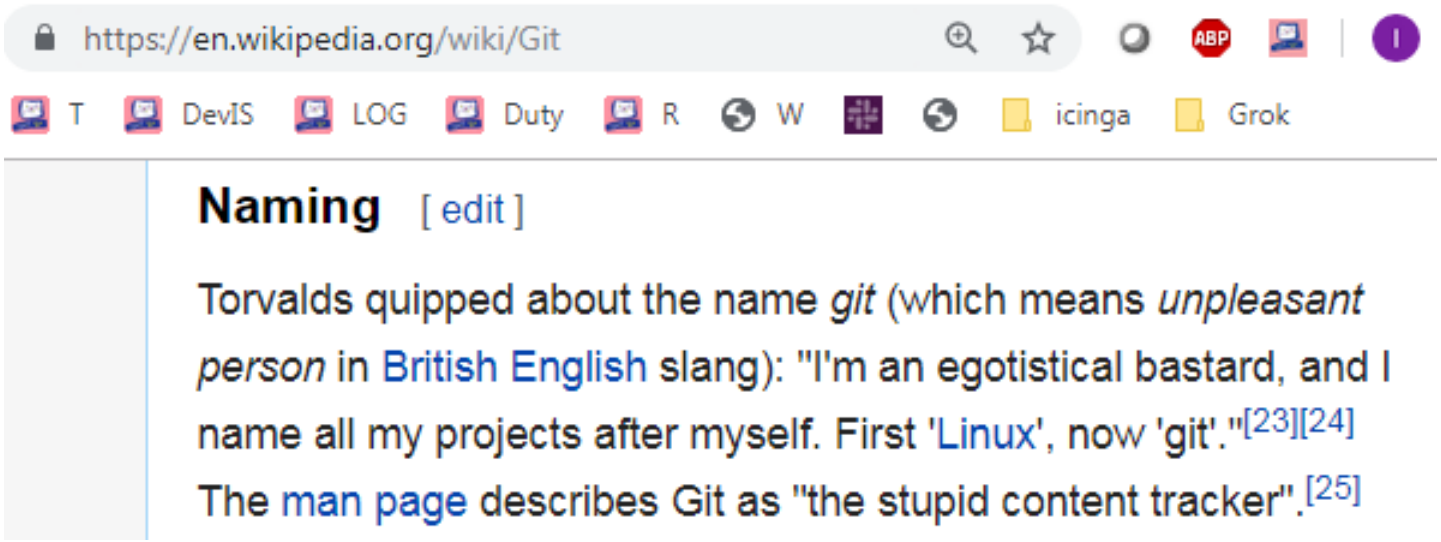
Deep inspection of basic commands

Ilya Rokhkin

Git – meaning?

What does the word Git mean?

GIT Overview



Agenda:

1. Basic concepts and commands in deep dive
   - Git Architecture, data model
   - DVCS, Repository, Commit, Parent Commit, Tree, Blob, Index, Refs
   - Basic commands to work in the Repository and outside.
   - Sharing work with peers.
   - Advanced commands explained (rebase -i, cherry-pick, squash, revert of all kinds, patch, reflog)
   - Best practices (rebase, commits per change, feature branch, etc.)
   - How to manage AI changes in Git

2. Practical part, lab work
   - 13 exercises from, basic to advanced
   - Best practices and manage AI changes exercise

GIT Overview

- Quick and efficient
- Expedite distributed development
- Atomic transactions, commit, cross repository
- Commits (Change) management
- A clear internal design
- Suited to handle everything from small to very large projects with speed and efficiency
- Support and encourage branched development

About myself:

Author: Ilya Rokhkin

+ Role: DevOps Engineer @ CloudOps DevOps Team

- Role: Harmony Connect DevOps

+ GitHub EMU Owner

Experience:

20+ years in VCS

Git trainer

+ (CHKP, Freelancer)

- (Intel, Marvell)

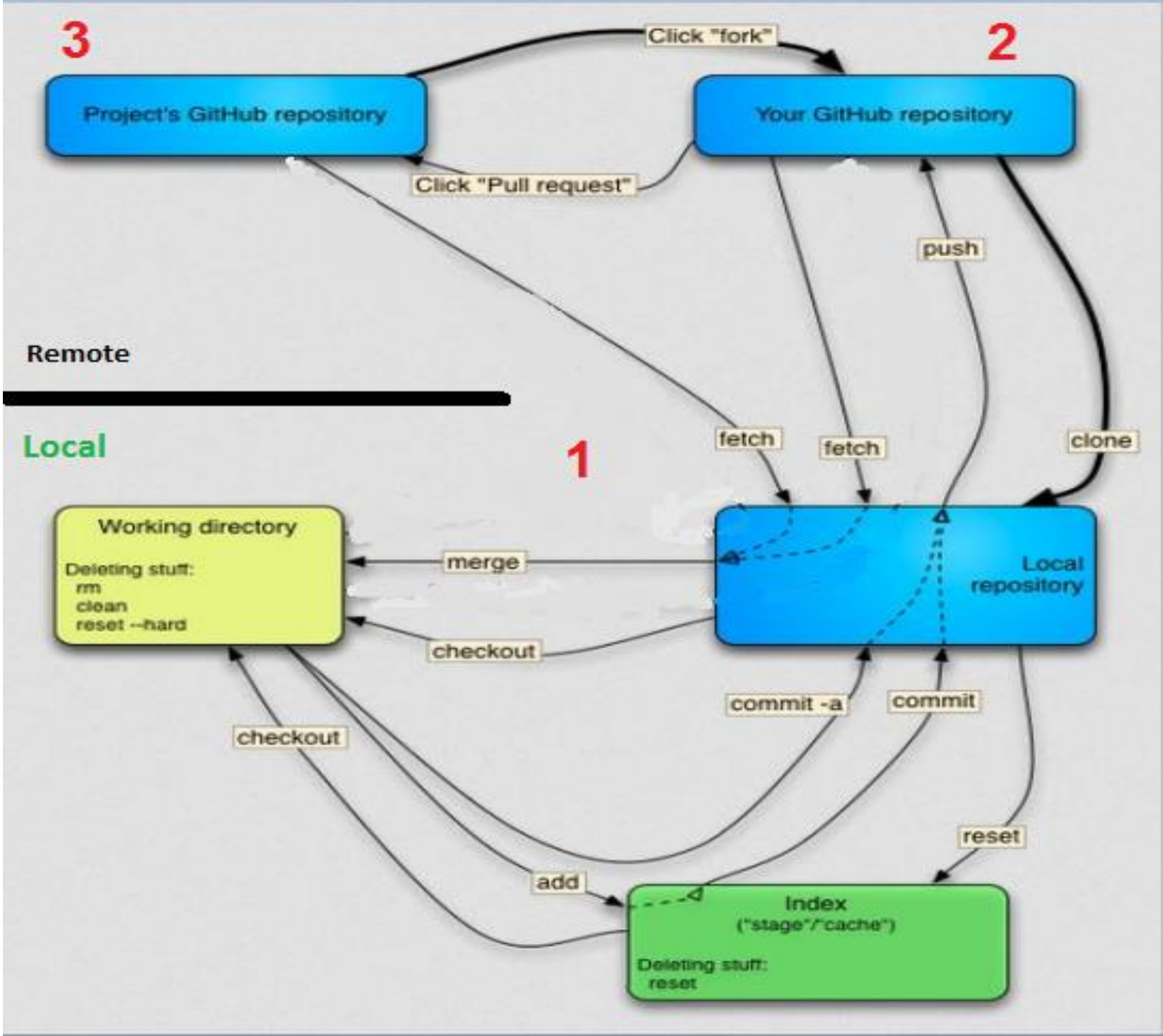
Hebrew teacher volunteer @ Ulpan

GIT Architecture

1. Your Local repo

2. Your Remote (Origin) repo

3. Common (Community) Remote Origin repo



Demo
Lab 1 - 2

- We will configure your git user and e-mail

  Will create a remote, bare repository in the home directory

  Clone it to work repository1 in the home directory also, add, commit, and push
- Restructure files, add, commit, and push

Git repository structure (.git)

```
$ ls -al
total 11
drwxr-xr-x   7 sheta   Administ   4096 Dec  3 15:17 .
drwxr-xr-x   3 sheta   Administ   4096 Nov 30 11:26 ..
-rw-r--r--   1 sheta   Administ     23 Nov 30 11:09 HEAD
-rw-r--r--   1 sheta   Administ    363 Nov 30 11:46 config
-rw-r--r--   1 sheta   Administ     73 Nov 29 17:03 description
drwxr-xr-x   2 sheta   Administ   4096 Nov 29 17:03 hooks
-rw-r--r--   1 sheta   Administ     32 Nov 30 11:26 index
drwxr-xr-x   2 sheta   Administ      0 Nov 29 17:03 info
drwxr-xr-x   3 sheta   Administ      0 Nov 29 17:03 logs
drwxr-xr-x  25 sheta   Administ   4096 Nov 30 11:08 objects
-rw-r--r--   1 sheta   Administ     94 Nov 29 17:03 packed-refs
drwxr-xr-x   5 sheta   Administ      0 Nov 30 11:07 refs
```

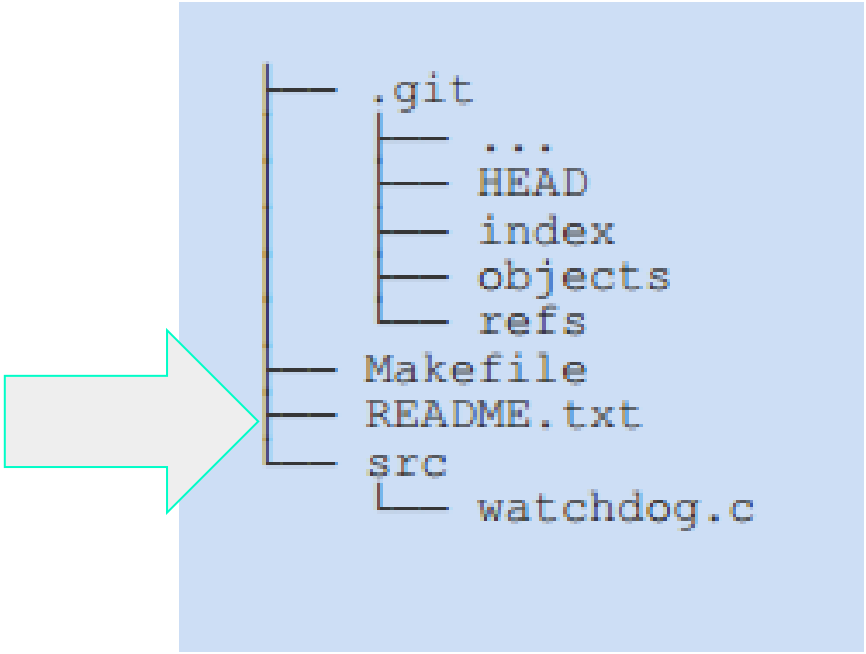
```
├── .git
│   ├── ...
│   ├── HEAD
│   ├── config
│   ├── description
│   ├── hooks
│   ├── index
│   ├── info
│   ├── logs
│   ├── objects
│   ├── packed-refs
│   └── refs
├── Makefile
├── README.txt
└── src
    └── watchdog.c
```

- HEAD — This file holds a reference to the branch (or commit) you currently have checked out
- config — This is the main Git configuration file. It keeps specific Git options for your project
- description — It will show when you have viewed your repository or the list of all versioned repositories.
- hooks — Directory contains shell scripts that are invoked after the corresponding Git commands
- index — The Git index is used as a staging area between your working directory and your repository
- info — Contains additional information about the repository.
- logs — Keeps records of changes made to refs.
- objects — In this directory the data of your Git objects are stored – all the contents of the files you have ever checked in
- packed-refs — The file consists of the tips of all branches and tags.
- refs — This directory normally contains the tips of branches and tags in three subfolders – heads, remotes and tags.

The Working Tree

- The working tree has all files and folders as found in your HEAD, plus the changes you made since your last commit
- There is only ONE main working tree per repository (and only 1 .git folder as well)

The Working Tree



States of files in Working Tree

- **Untracked** – in the repository folder, git does not keep a version of it.
- **Modified** – tracked, modified since last stage or commit.
- **Staged** – a snapshot of the file, ready to be committed. Even if modified, git will still keep the snapshot.
- **Committed** – version of file saved in repository DB

Objects

There are only **4** object types in GIT:

Type – “blob”, “tree”, “commit”, “refs” (branch/tag).

A "blob" is basically like a file – it is used to store the content of a source file.

A "tree" is basically like a directory - it references a group of other trees (subdirectories) and/or blobs (files).

A "commit" points to a single tree, marking it as what the project looked like at a certain point in time. Keeps changed files since the last commit, author of the changes, a reference to the parent commit(s), etc.

A “refs" is a way to mark a specific commit as special in some way. It is usually used to tag certain commits as specific releases or something along those lines.

Objects cont.

Almost all the GIT is built around manipulating this simple structure of four different object types. It is sort of it's own little file system that sits on top of your machine's file system.

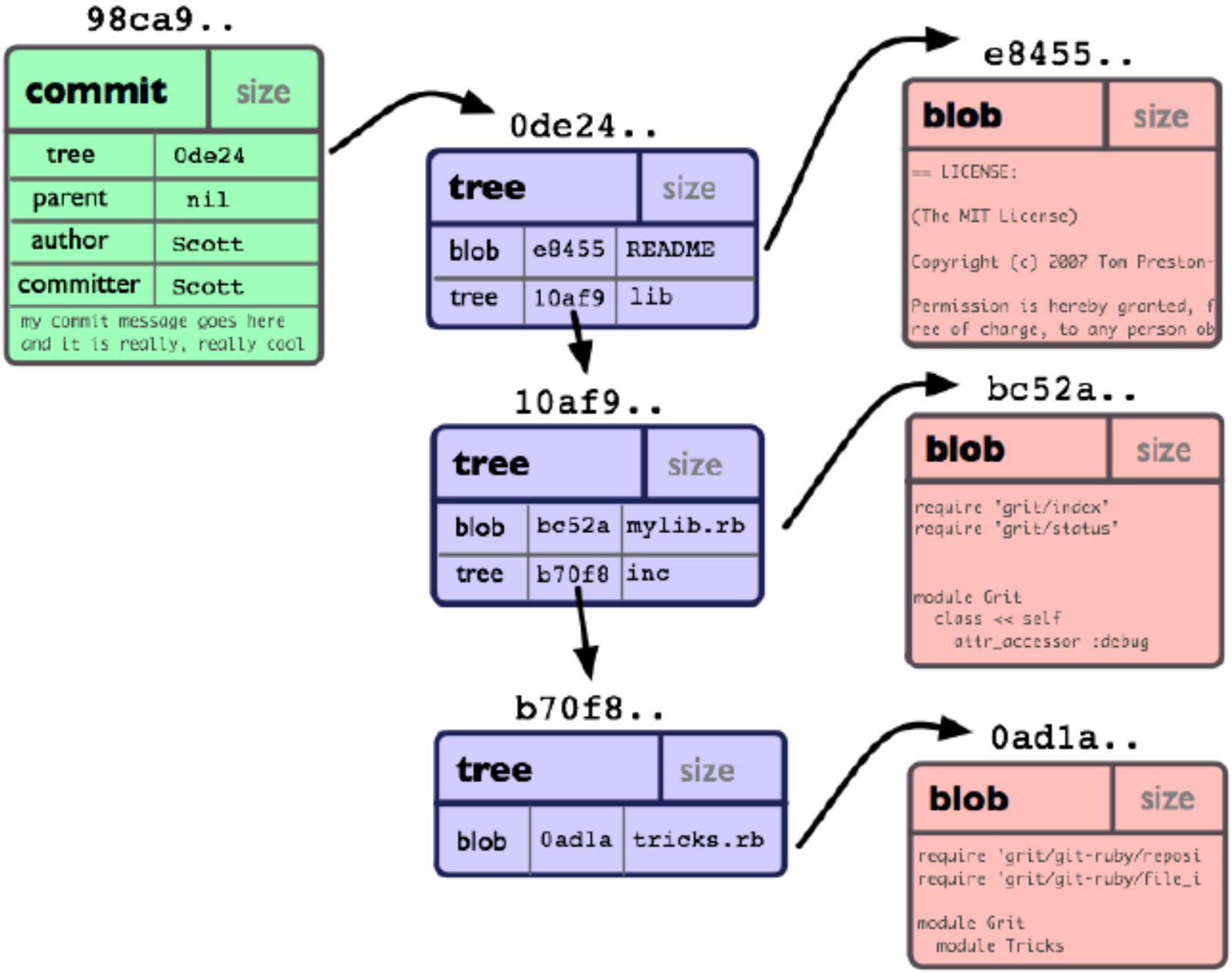
Let's say we have a small project that looks like this:

```
$>tree
.
|-- README
`-- lib
    |-- inc
    |   `-- tricks.rb
    `-- mylib.rb

2 directories, 3 files
```

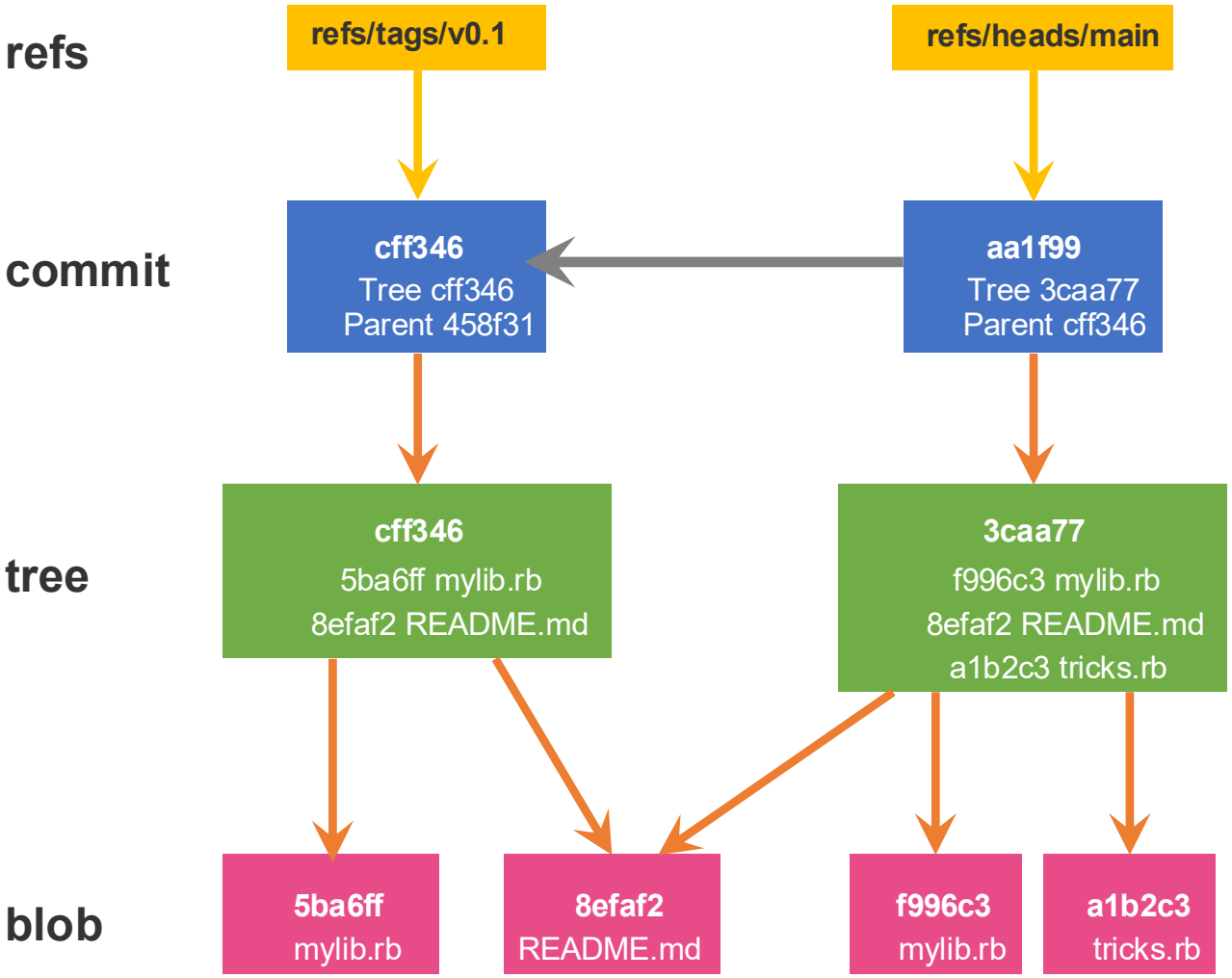
If we will commit this project to a GIT repository, it will be represented in GIT like this:

Commit Object



Commit the object with its parent

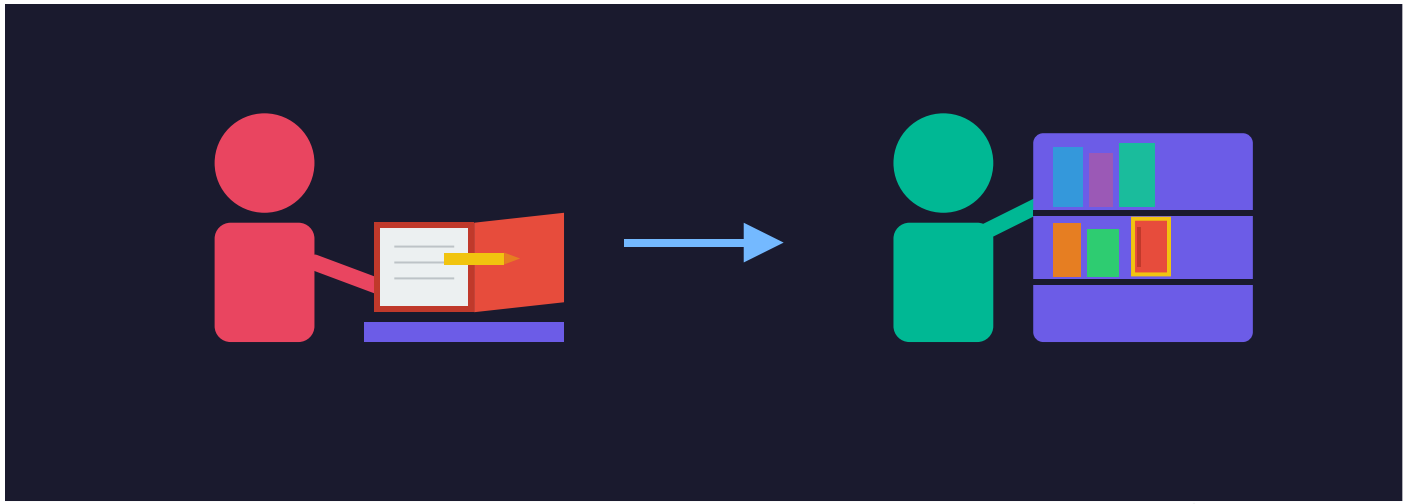
- Links from the tree object to the common blobs



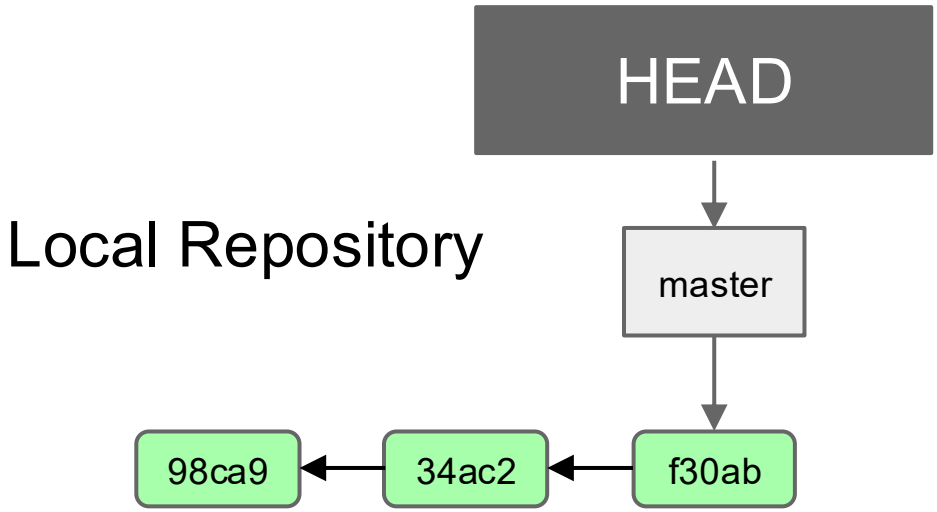
Secure Hash Algorithm – SHA1

- Each object in Git is represented by a 40-digit string that resembles this: **7bf68ebf3d8cff042bd3cb87e7592ddda9caa665**.
  This string is calculated by taking the SHA1 hash of the object's contents.

  $ git log --pretty-fuller –stat
  commit c3dd67d83ce90b75b9d94af5a357eb37a34d80db (HEAD -> main)
  Author: Or <or@yahoo.com>
  AuthorDate: Sun Mar 06 11:30:00 2016 +0300
  Commit: ilya <ilya@yahoo.com>
  Commitdate: Thu Jul 13 21:50:02 2017 +0300



The HEAD



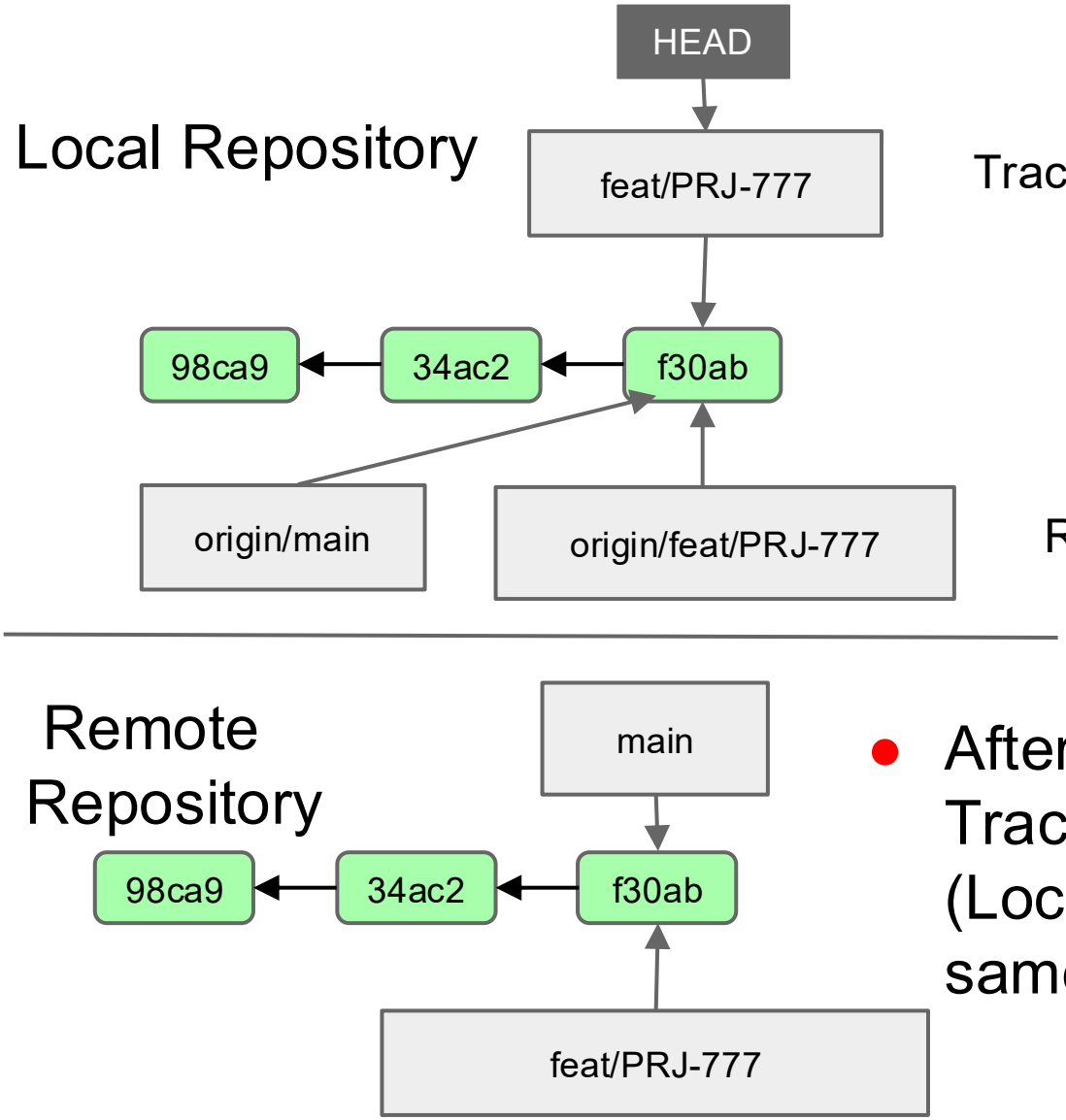
- `HEAD` is a 'pointer' to the tip of the currently checked out branch
  - In a *detached HEAD* state, `HEAD` points directly to a commit
- Only one `HEAD` per repository

Demo
Lab 3

Teamwork, parallel work:

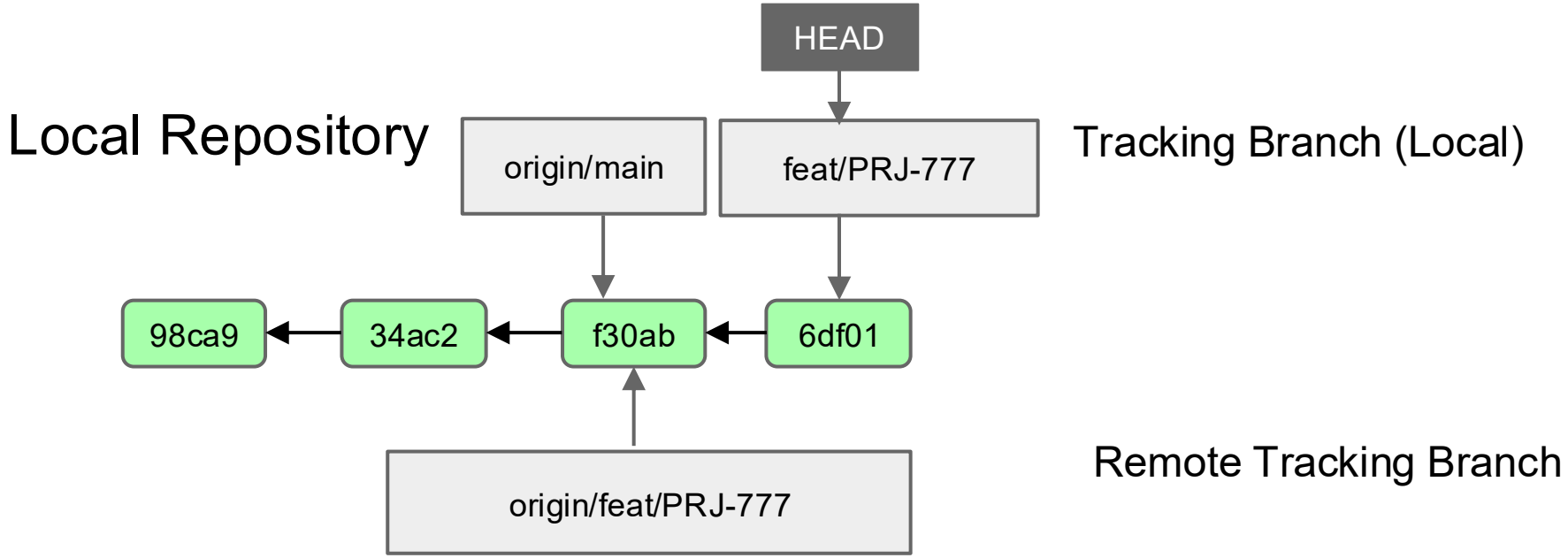
- We will clone the second work repository2
- We will commit changes in both repositories
- Push and pull with silent rebase to apply the commit of one repo into another
- Overview results

Rebasing Remote Tracking Branch



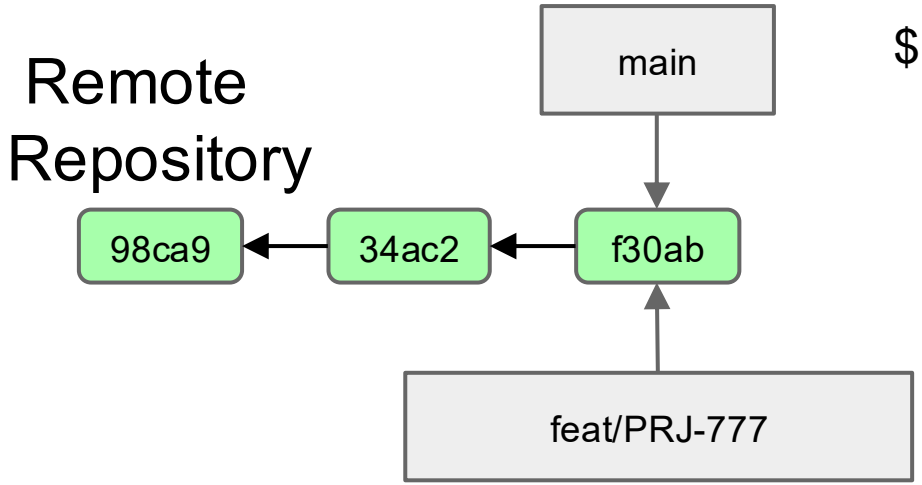
- After clone/pull Remote Tracking Branch and Tracking (Local) branch pointing to the same commit

Rebasing Remote Tracking Branch

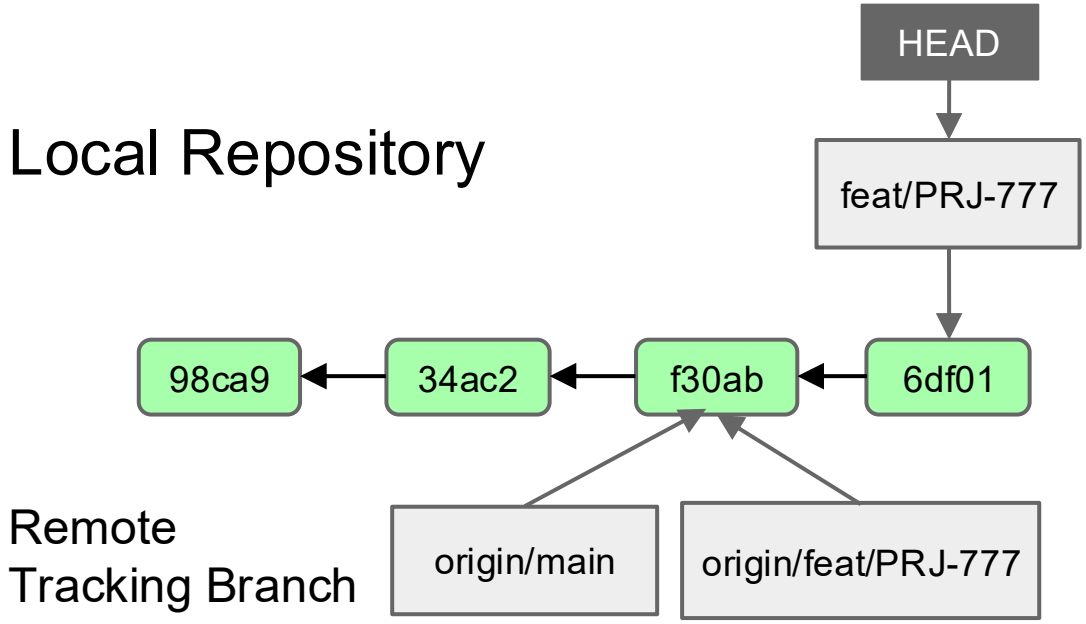


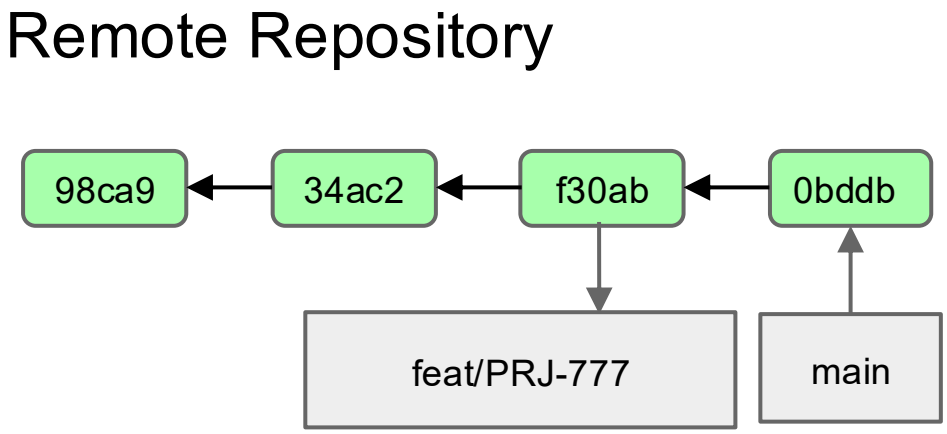
- **Commit on local repository**

$ git commit



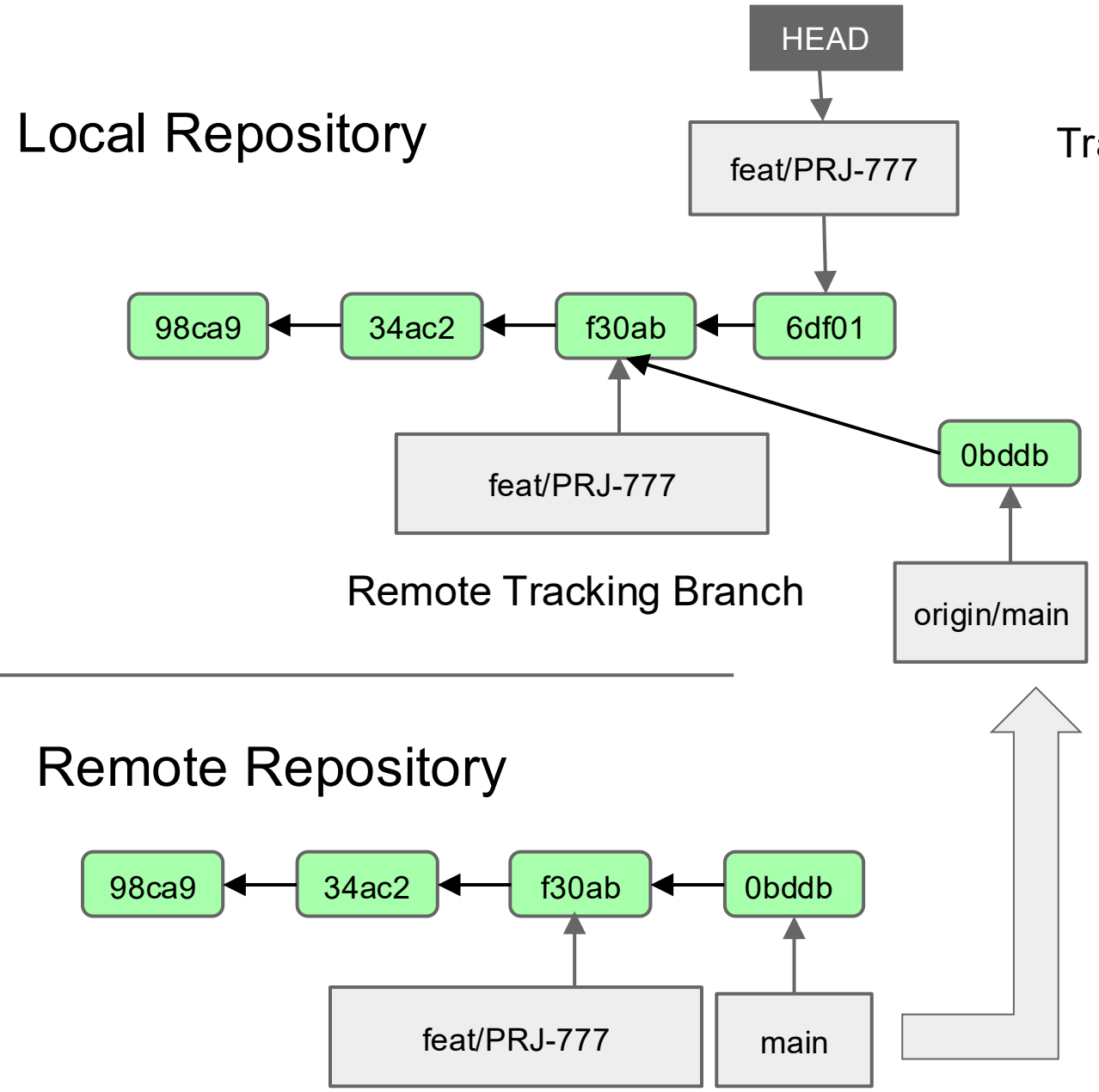
Rebasing Remote Tracking Branch





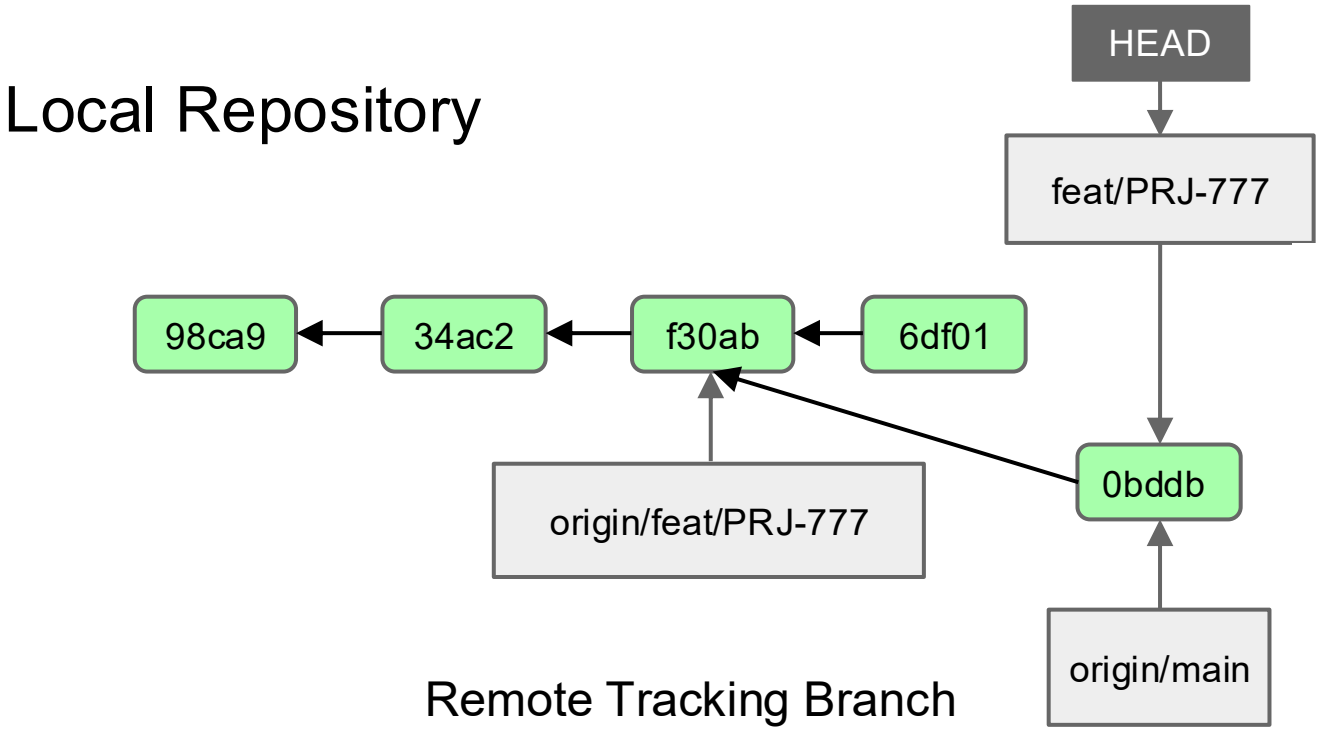
- **Merge another Commit to remote repository from another feature branch**

Rebasing Remote Tracking Branch



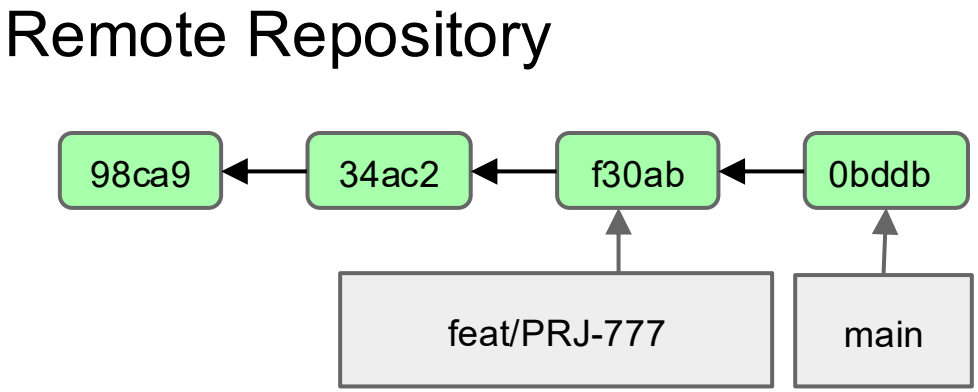
- **Fetch the Commit from remote repository to Remote Tracking Branch**

Rebasing origin/master

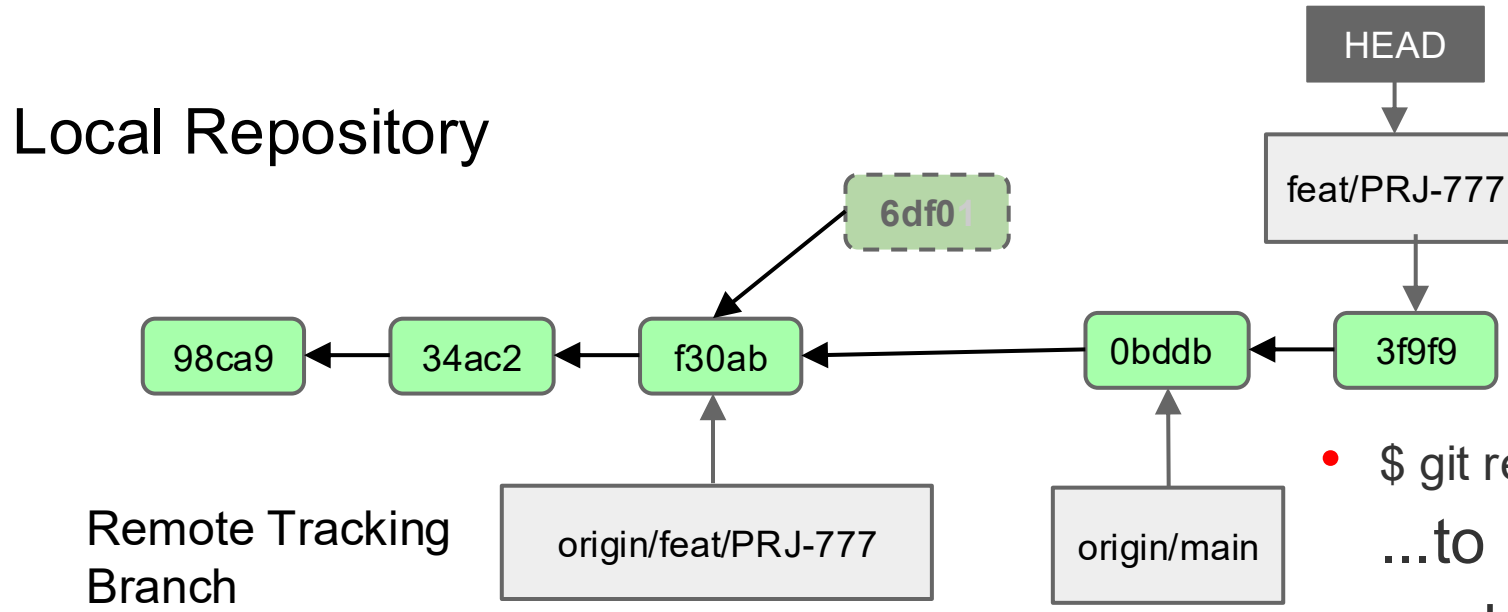


- $ git rebase origin/main

First, rewinding head to replay your work on top of it...

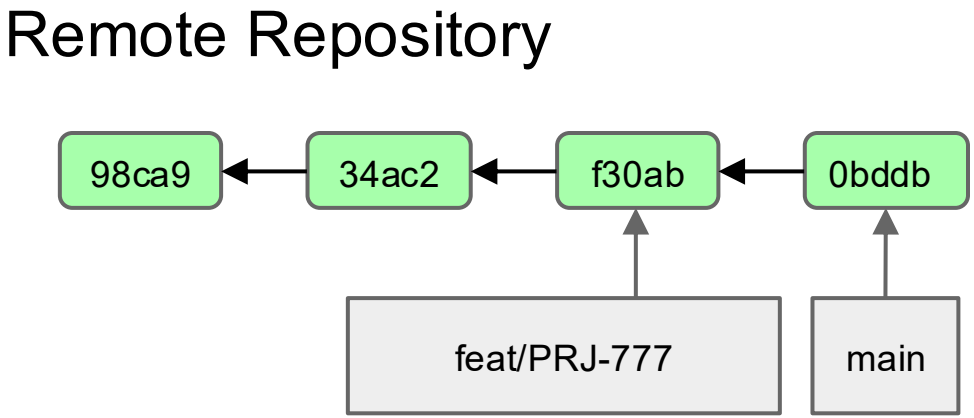


Rebasing Remote Tracking Branch



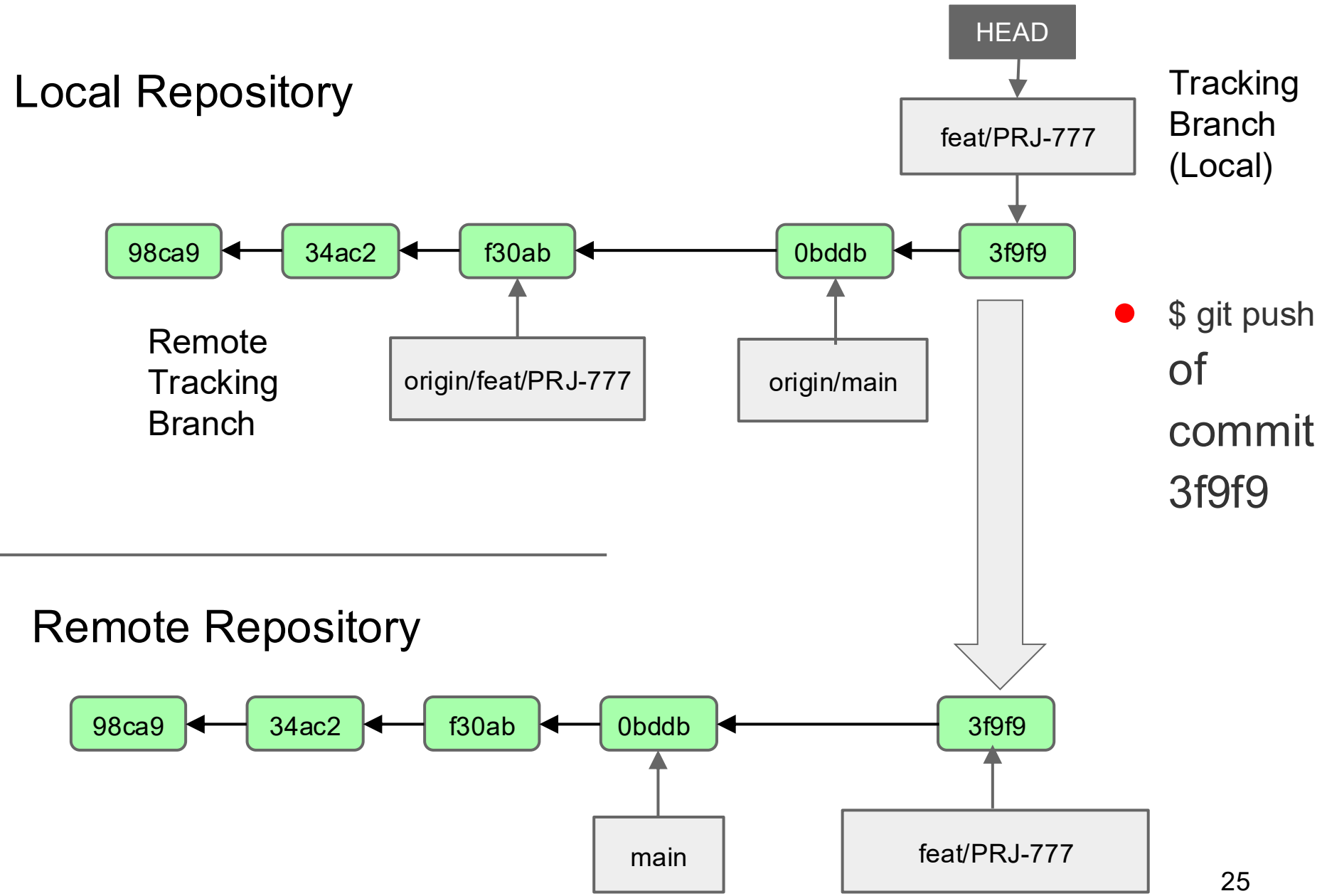
- $ git rebase origin/main

...to replay your work on top of it…

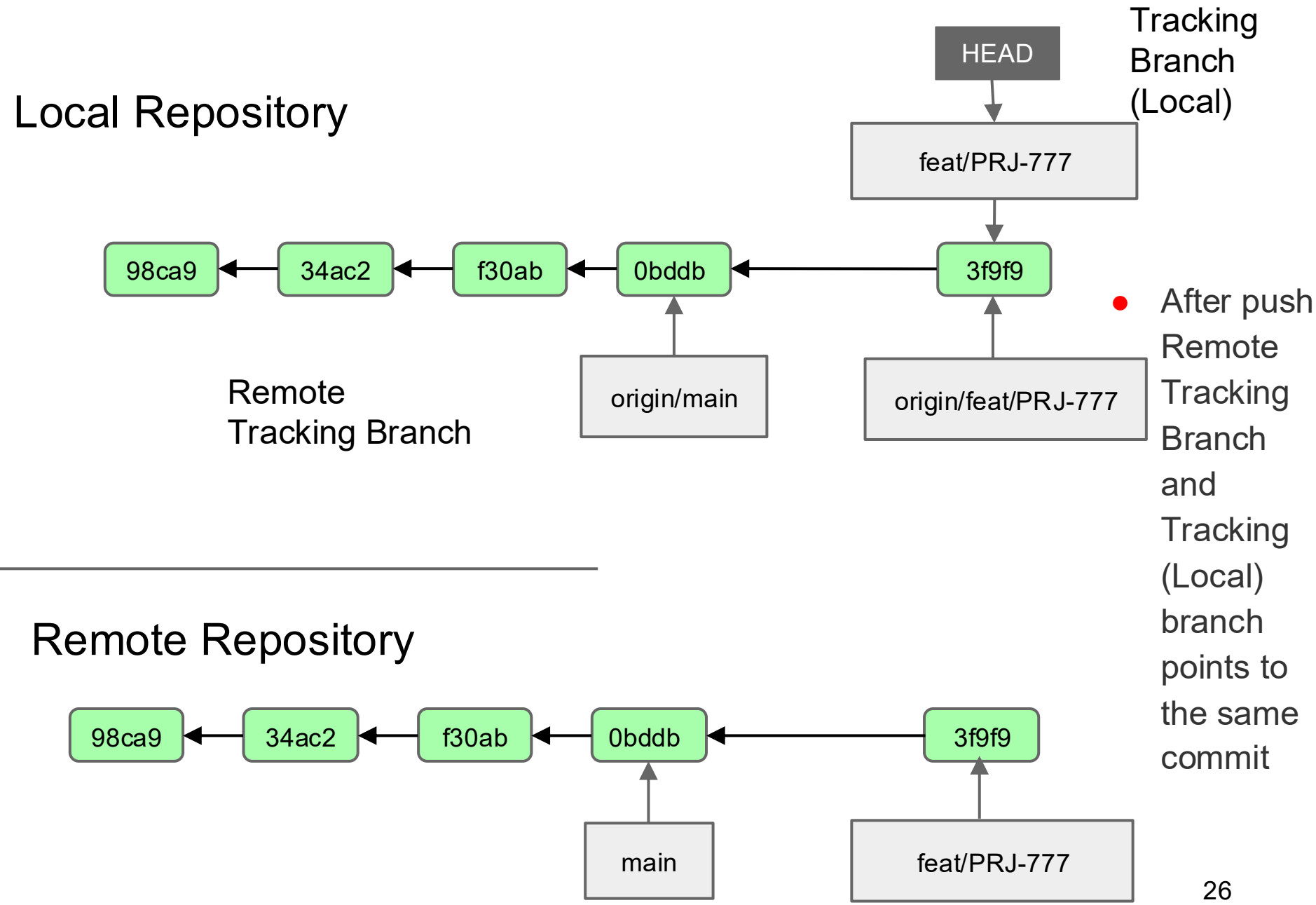


Applying: diff/patch of commit 6dfo on top of origin/main to create commit 3f9f9

Rebasing Remote Tracking Branch



Rebasing Remote Tracking Branch



- After push Remote Tracking Branch and Tracking (Local) branch points to the same commit

Perils of rebase

- **Do not rebase commits that you have pushed to a public repository.**

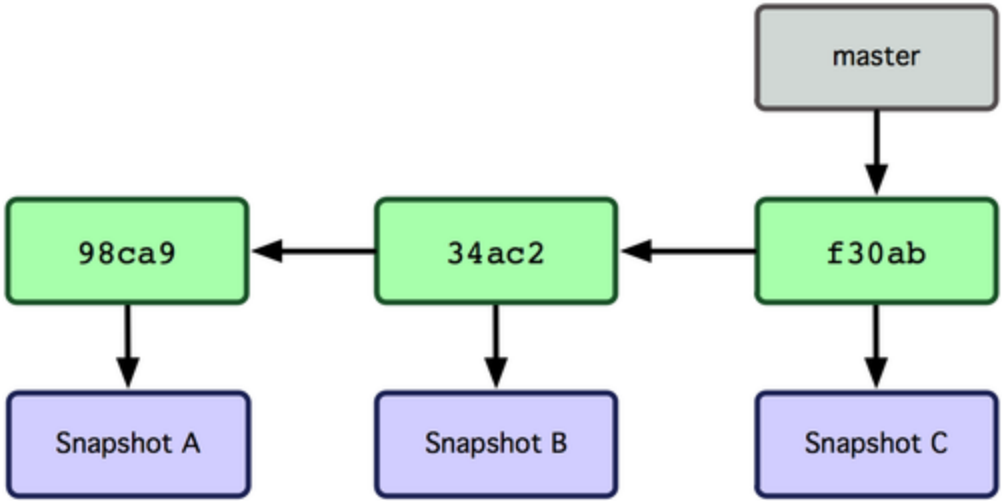
If you follow that guideline, you'll be fine. If you don't, people will hate you, and you'll be scorned by friends and family.

Demo
Lab 4

- Teamwork, parallel work with rebase, and conflict resolution:
- We will do commits in both repositories, with changes in the same line of the same file
- Pull with rebase, resolve conflicts, and save the resolution file
- Adding the file to the staging area means resolving the issue.
- Commit and push

Git branches

- Git branch is simply a movable pointer to a commit

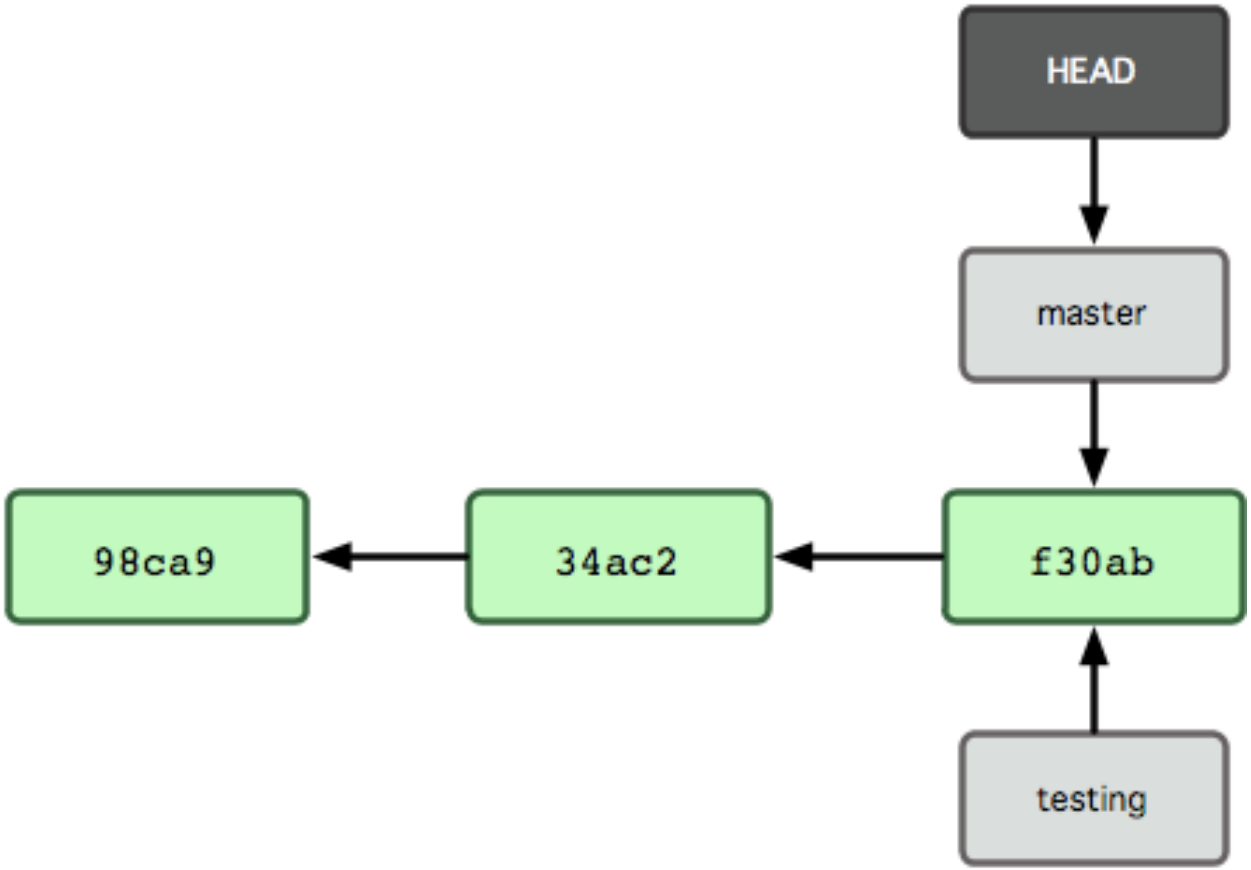


- Pointer moves forward automatically with each commit on a branch

Creating new branch

- New branch creates a new reference

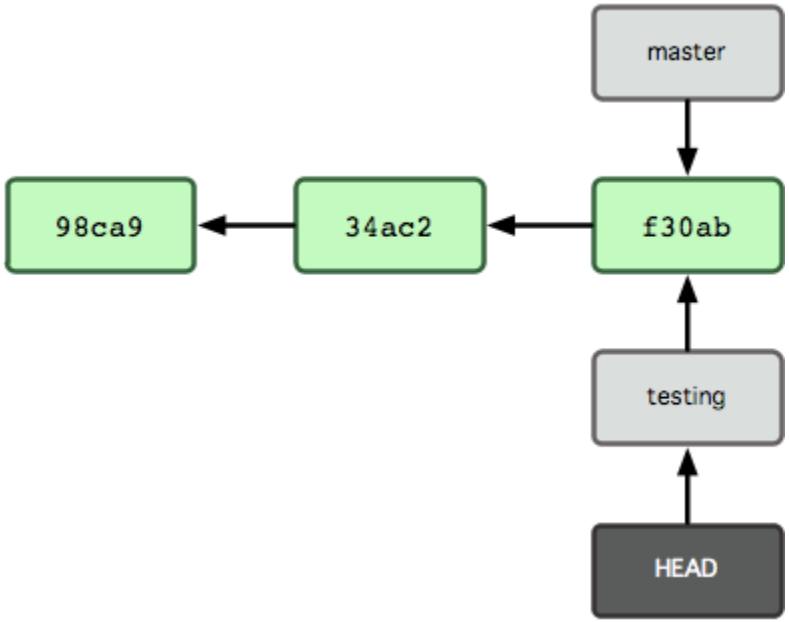
```
> git branch testing
```



Switching to a branch

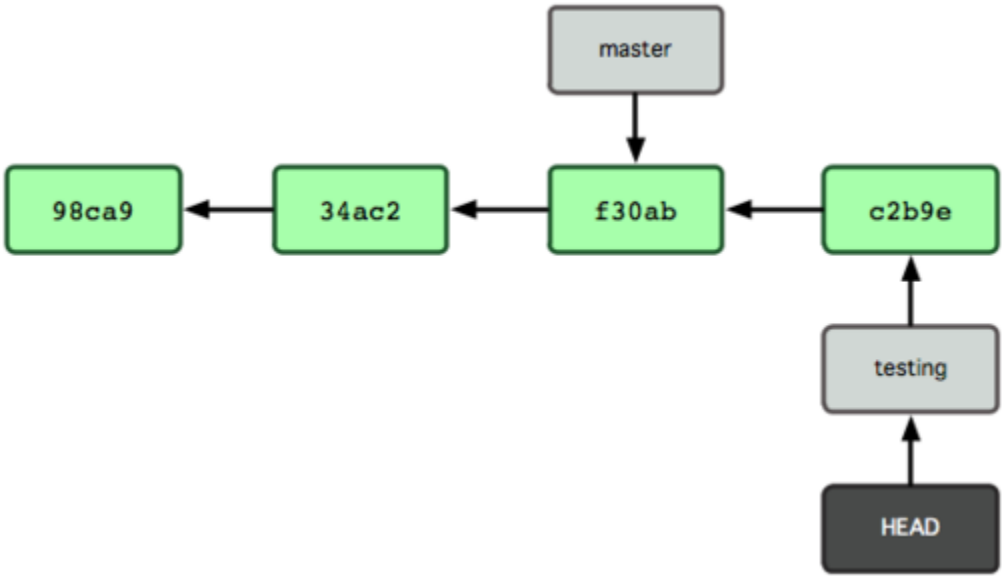
- Git checkout *branch-name* switches to an existing branch

```
> git checkout testing
```

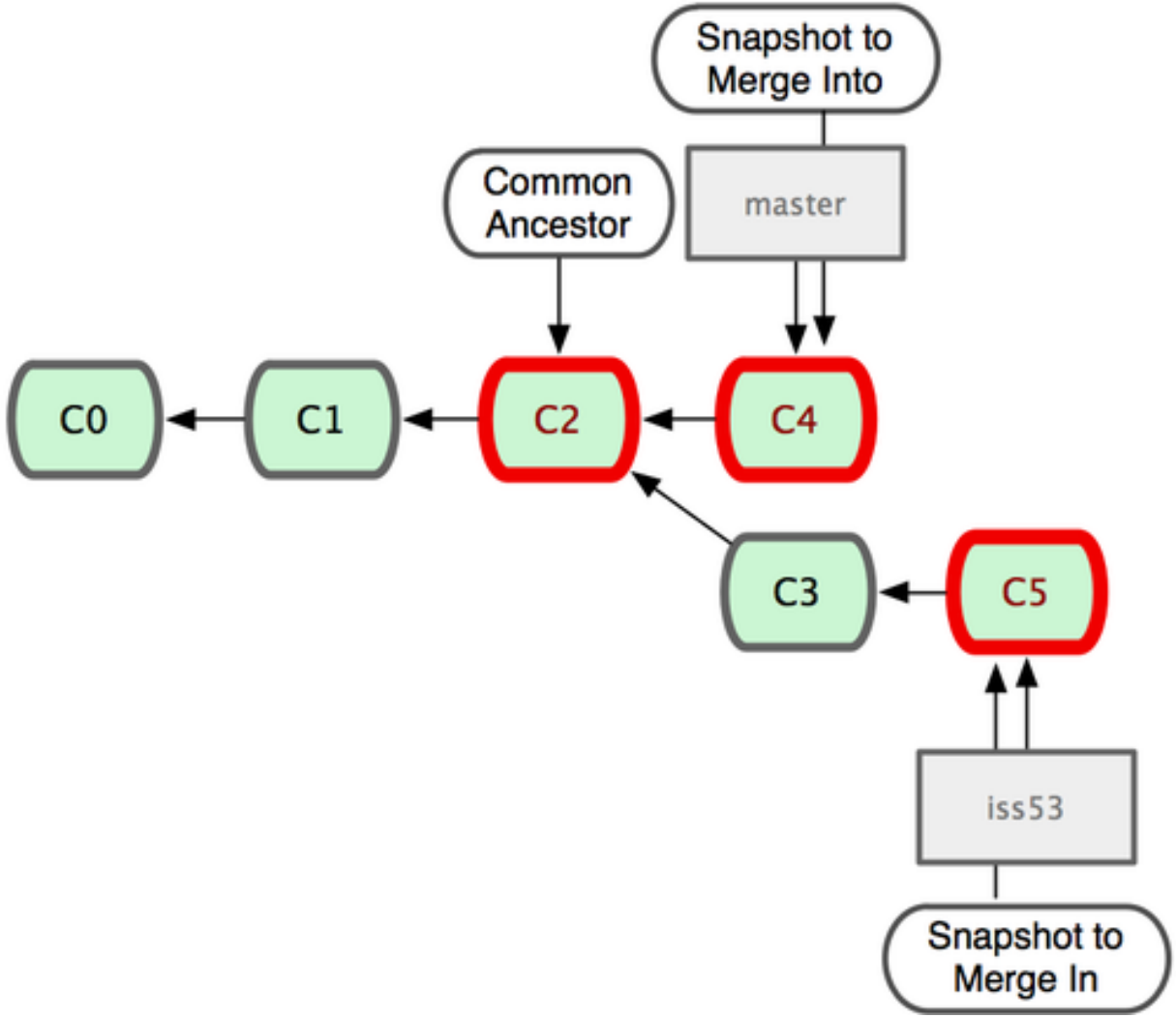


New commit moves current branch

```
> vi file04.txt
> git commit -a -m 'Commit message'
```

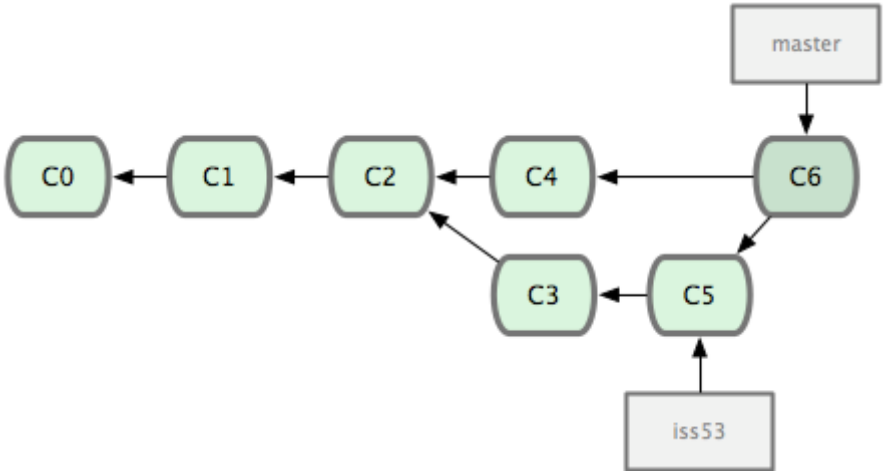


Merging branches



Merging branches (continued)

- As a result of merge Git creates a new commit, which has two parents:



Demo
Lab 5 - 7

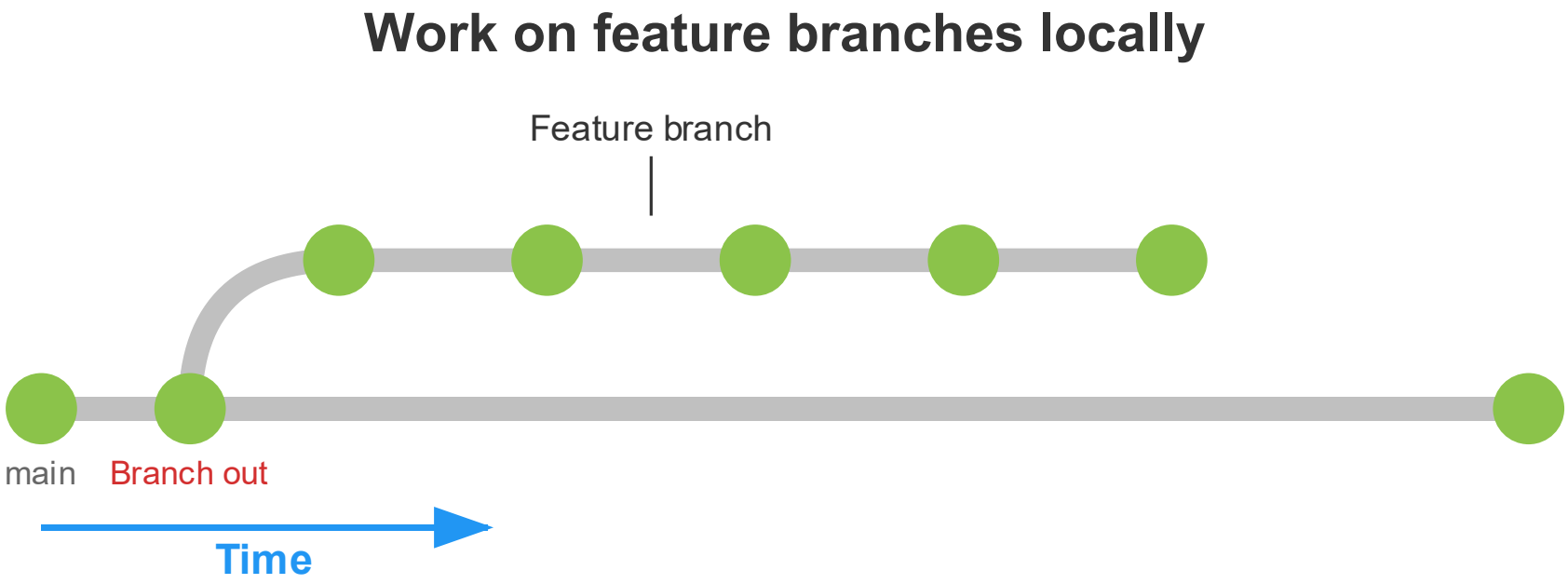
- We will create a tag
- Create branch bugfix from the tag

  I will do a bugfix commit in the bugfix branch

  Check out the master branch and commit a new change
- Merge bugfix branch

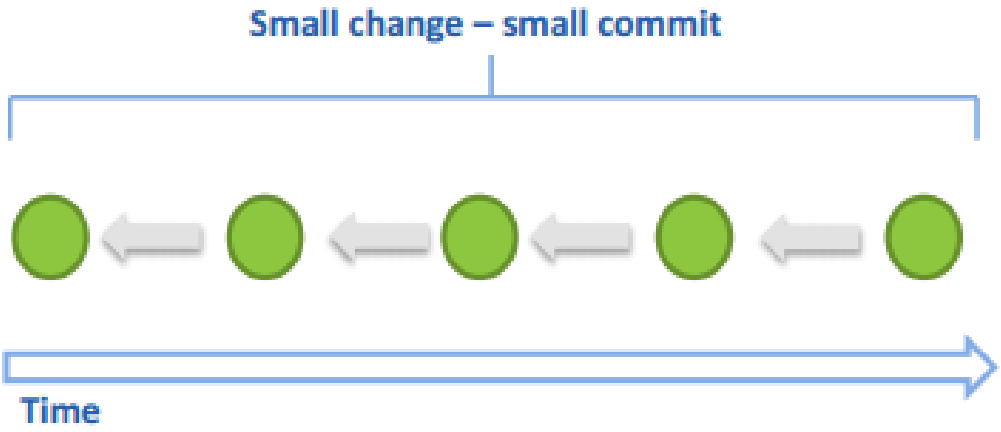
  Overview results

Best practices – local feature branches



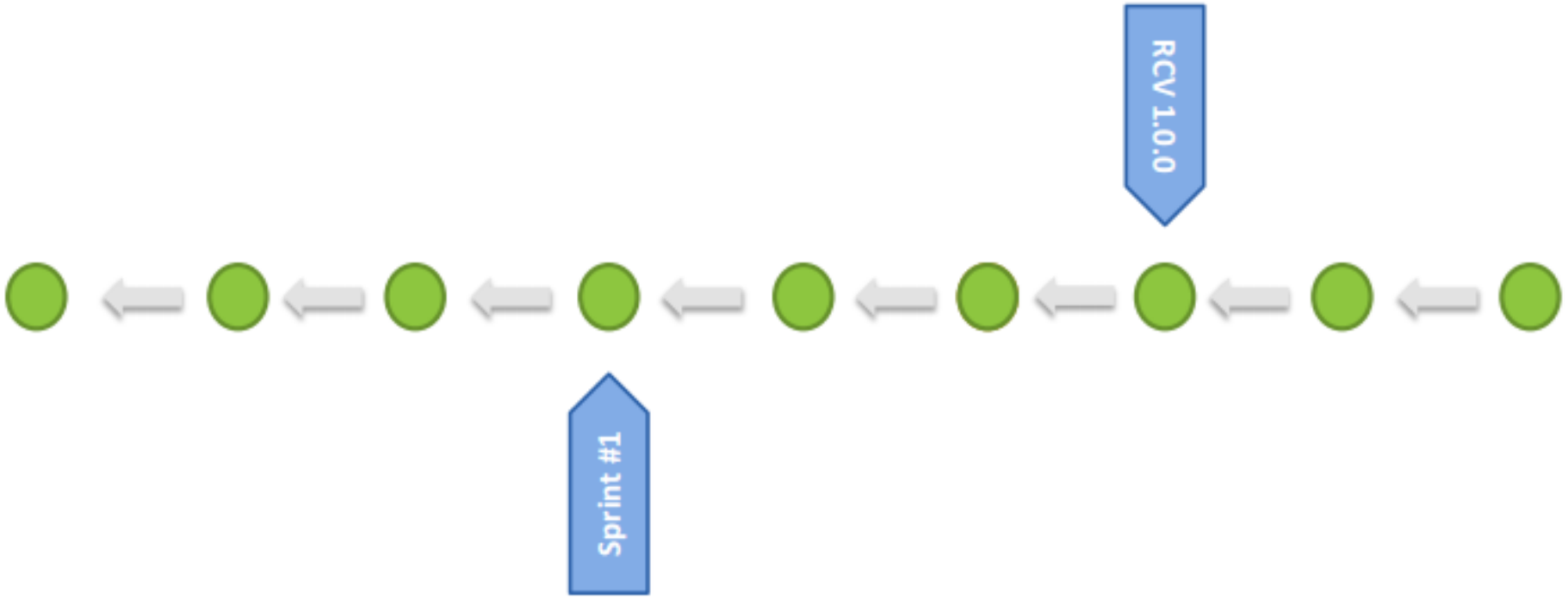
Best practices – commit

Keep changes small wherever possible and commit frequently



Best practices – tag milestones and important working changes!

Tag important milestones (for history and for accessibility)

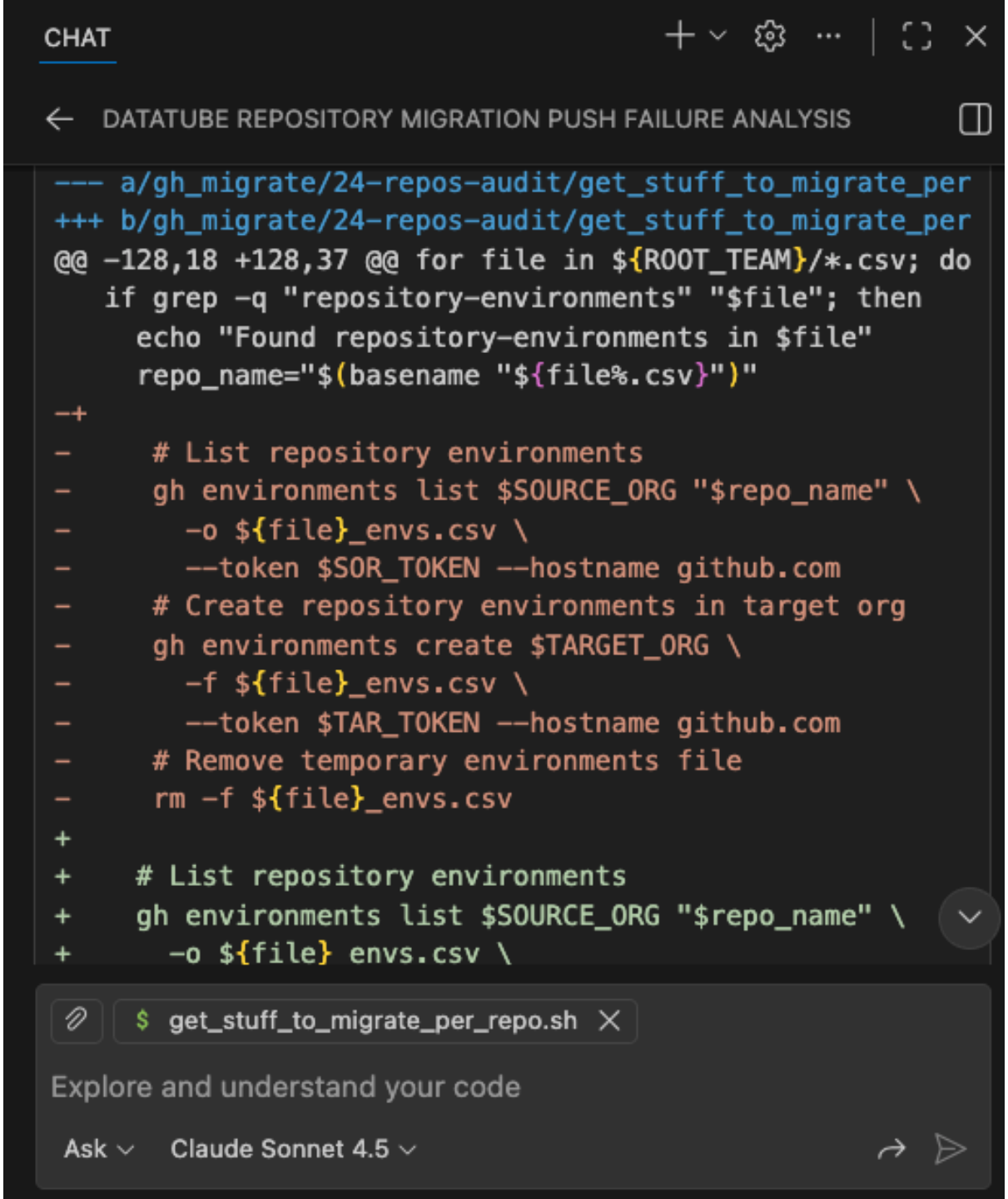


How Git can manage AI changes

- VS Code Preferences: Open User Settings (JSON)

```
"github.copilot.chat.codeGeneration.instructions": [
{
  "text": "When suggesting code changes, must show only diff"
}]
```

- **Ask AI give you small changes**



Control AI Changes with **Auto-Commit**-Push

VS Code Preferences: Open User Settings (JSON)

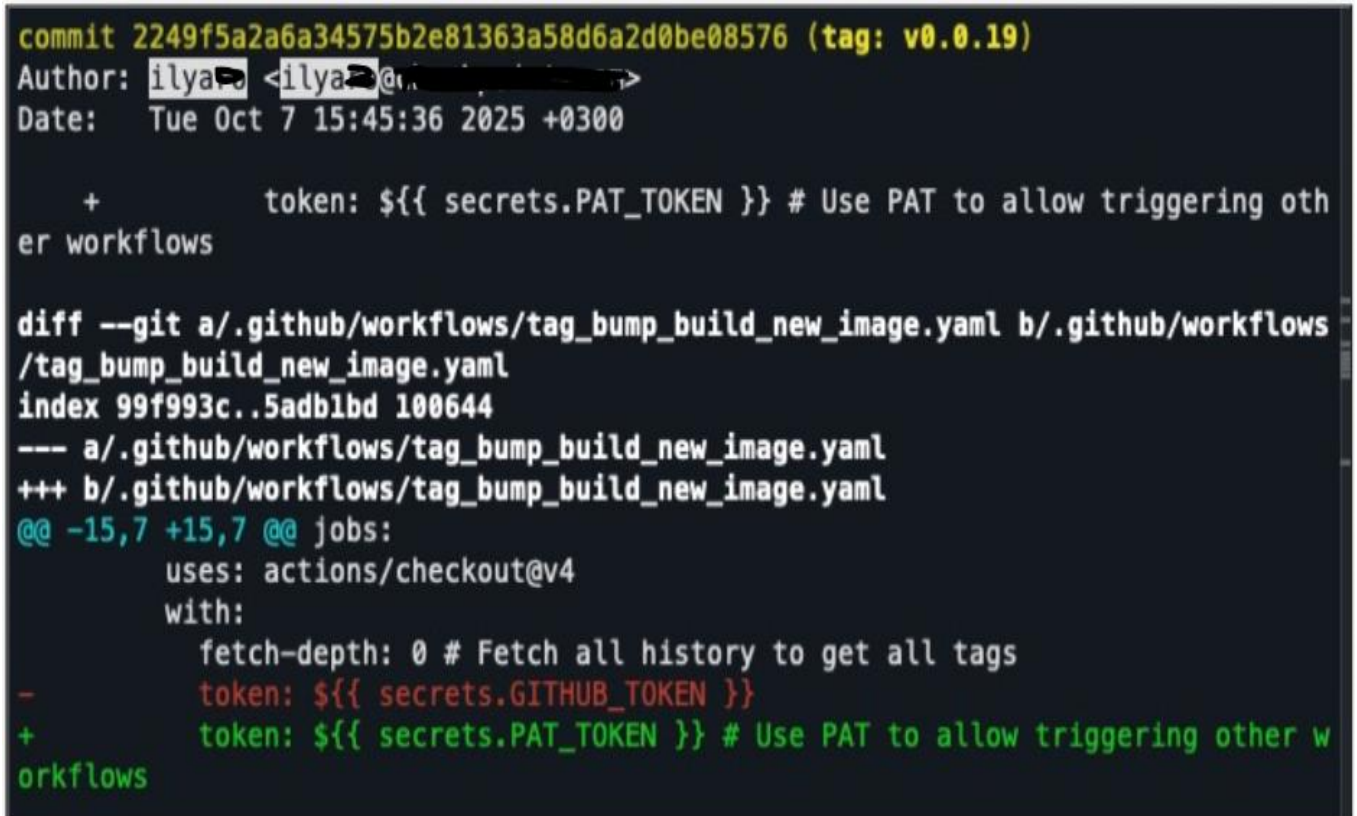
```
"emeraldwalk.runonsave": {
    "commands": [
    {
        "match": ".*",
        "cmd": "git rev-parse --is-inside-work-tree >/dev/null 2>&1 && git ls-files --error-unmatch ${file} && git commit -m \"`git diff -U0 ${file} | tail -1`\" ${file} && git symbolic-ref --short refs/remotes/origin/HEAD | grep -v $(git branch --show-current) > /dev/null 2>&1 && git push"
    }]},
```

Tag important working changes and push:

$ git tag v0.0.19 && git push –tags

- Save time for commits, keep them on the git server in a feature branch
- Reset to working tag when needed

$ git reset –hard v0.0.19



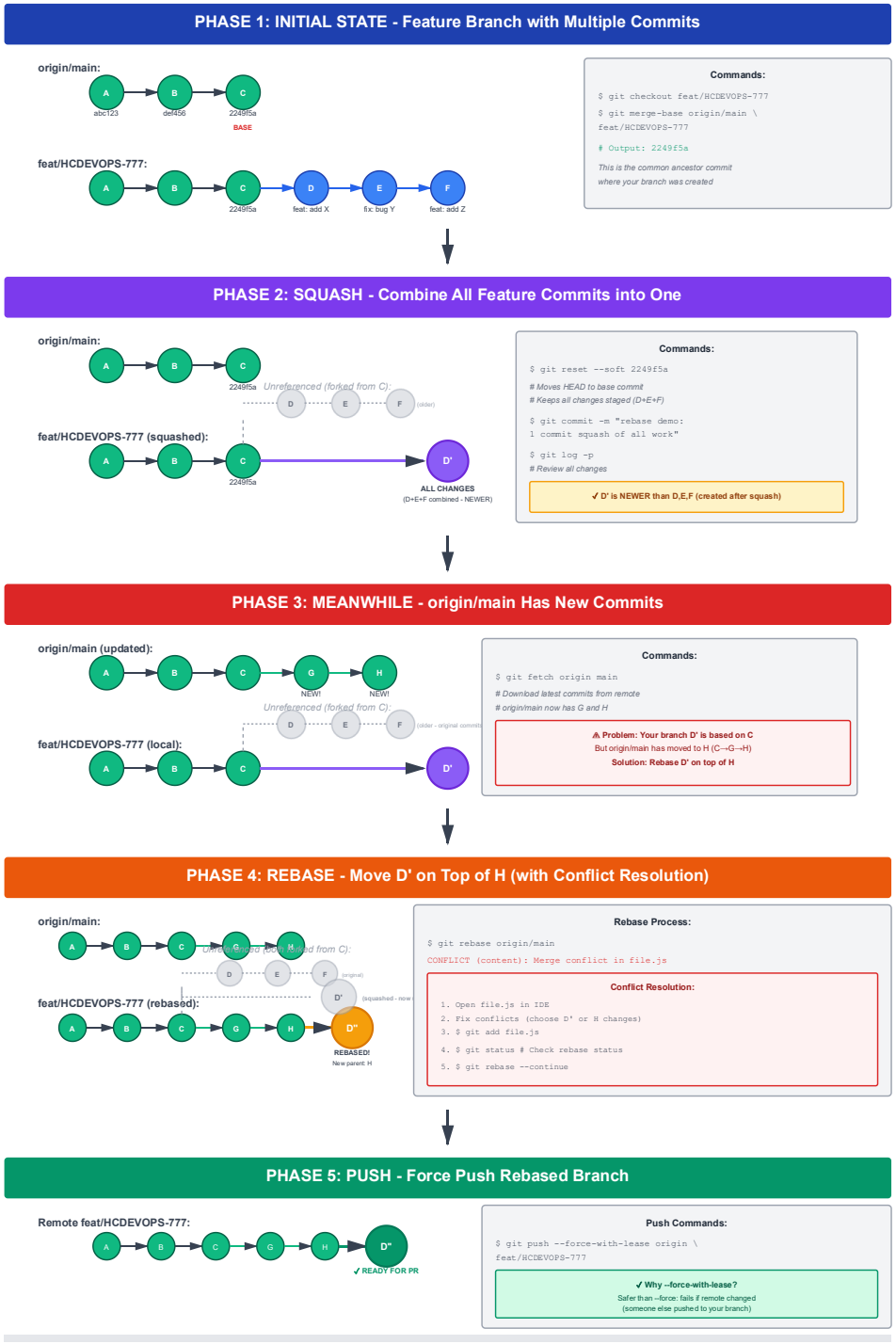
Best practices – squashing

Before pushing, squash related changes together to make for better understanding by others

Rebase on top of updated origin/main branch before pushing for code review.

Do it now on your current work branch.

https://github.com/ilyaro/git_best_practices_ppt/blob/master/Git_Rebase_Best_Practices.md



Best practices - concise commit messages
For the **main** branch, 1 commit

```
commit 212a24598895e038abc2e58611758cdac04012de
Author: Ilya Rokhkin <ilya@domain.com>
Date:   Tue Dec 30 13:59:04 2025 +0200

    CLOUDEVOPS-4466: Removing cron: '0 8-22 * * *' (#15)

    Removal cron from workflow
    Will be run on demand and not on GitHub

    ---
    Jira: CLOUDEVOPS-4466
    Branch: feat/CLOUDEVOPS-4466-gh-migration-remove-workflo
w-cron
```

- **Concise commit message header from 50 to 70 chars for best readability, with Jira task ahead**
- **Meaningful message body**

Best practices – Branch Layout

Git branches with type/JIRA-123-task-short-description

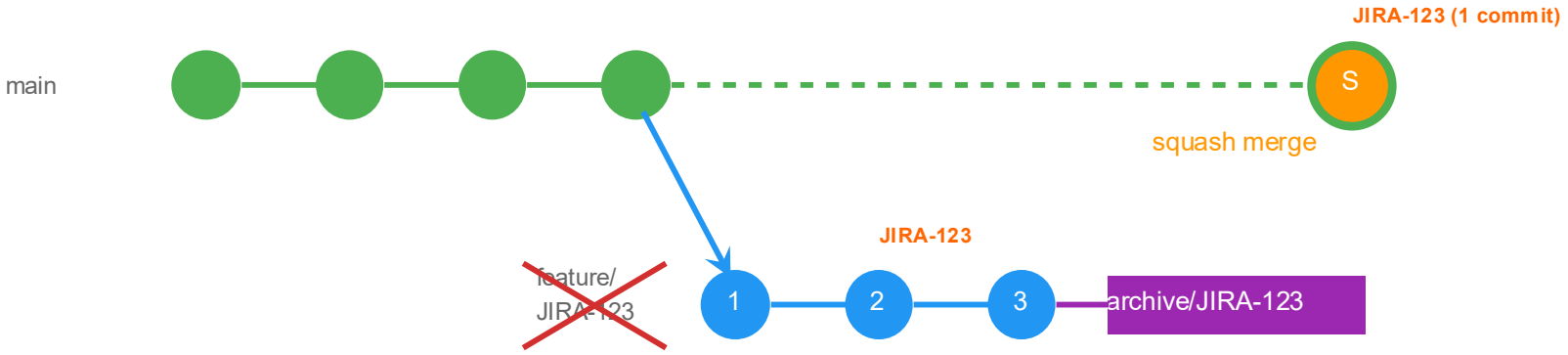
Type: fix, feat, release, doc, etc. – type of task you do
Slash: / - it is folder/file separation, you can select all fixes
Jira: JIRA-123 - CHKP Jira task
Short Description of the task: -task-short-description

Examples:

```
$ git branch
  feat/CLOUDEVOPS-4335-check-gitlab-blocked-urls
  fix/CLOUDEVOPS-4466-gh-migration-fix-mirroring
  feat/CLOUDEVOPS-4466-gh-migration-remove-workflow-cron
* feat/CLOUDEVOPS-4674-add-environments-and-their-vars
  main
```

Best practices – clean up local branches

Delete local and remote branches, once the code is merged to the main branch



Cleanup Commands:

```
1. git tag archive/JIRA-123 feature/JIRA-123
2. git fetch --prune
3. git branch -D feature/JIRA-123
```

Save branch history (push –tags if needed)

Fetch and delete local and remote-tracking refs

Force delete local branch

Lab 8 - 10

- We will create branch bugfix2, from Release_01
  Cherry-pick 1 commit from the master branch
- Undo modified file, undo staged file
  Undo the latest local commit, revert the pushed commit
- Stash meanwhile work aside, make a commit, return work from stash

Lab 11 - 13

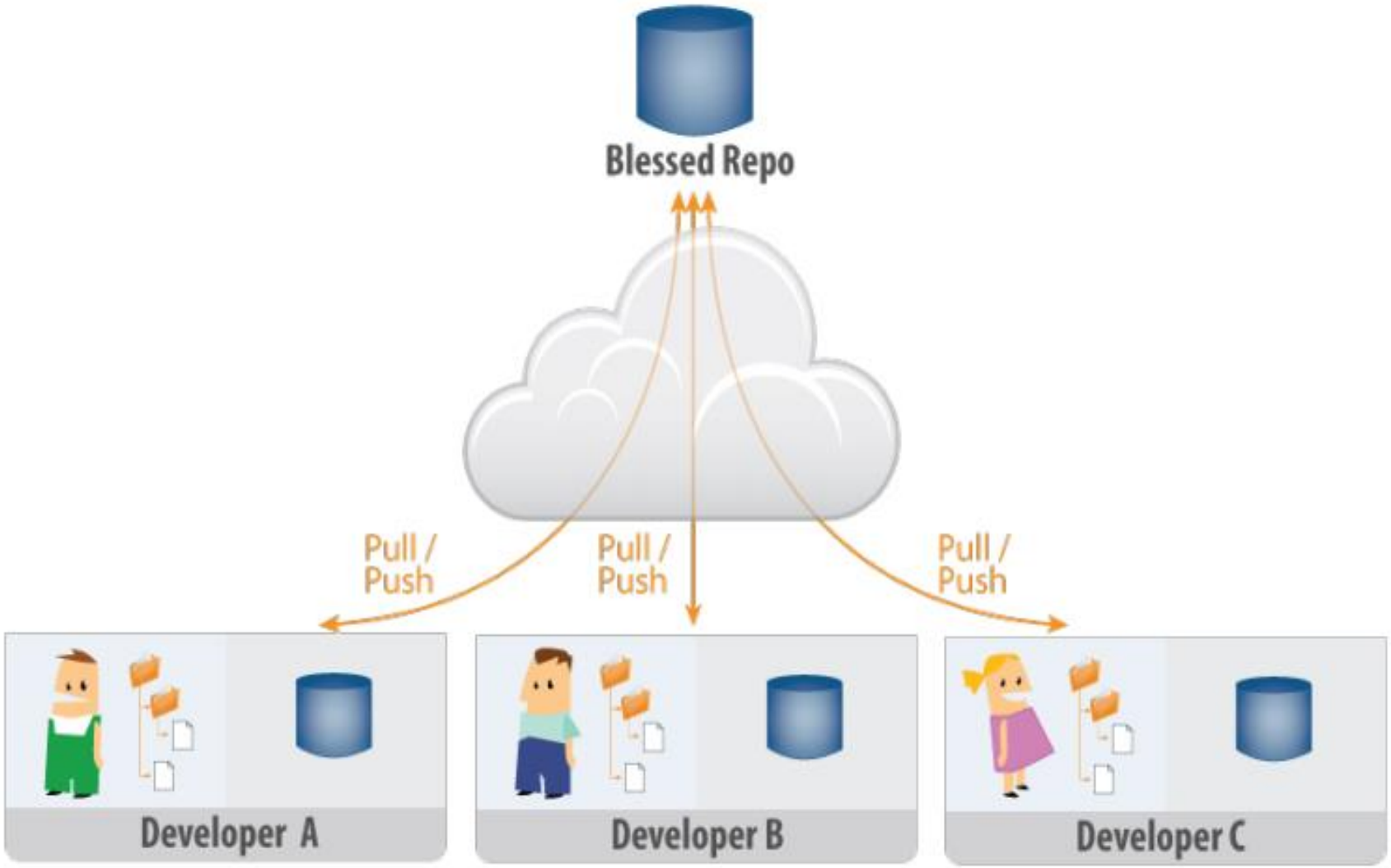
- Format patch in 1 repository, and apply it in another repository
- We will create 2 local commits squash them to 1 commit by interactive rebase, and push only 1 commit to the remote repository
- Create a commit in 1 repository and pull it from another repository, without pushing it to the origin repository

Question?

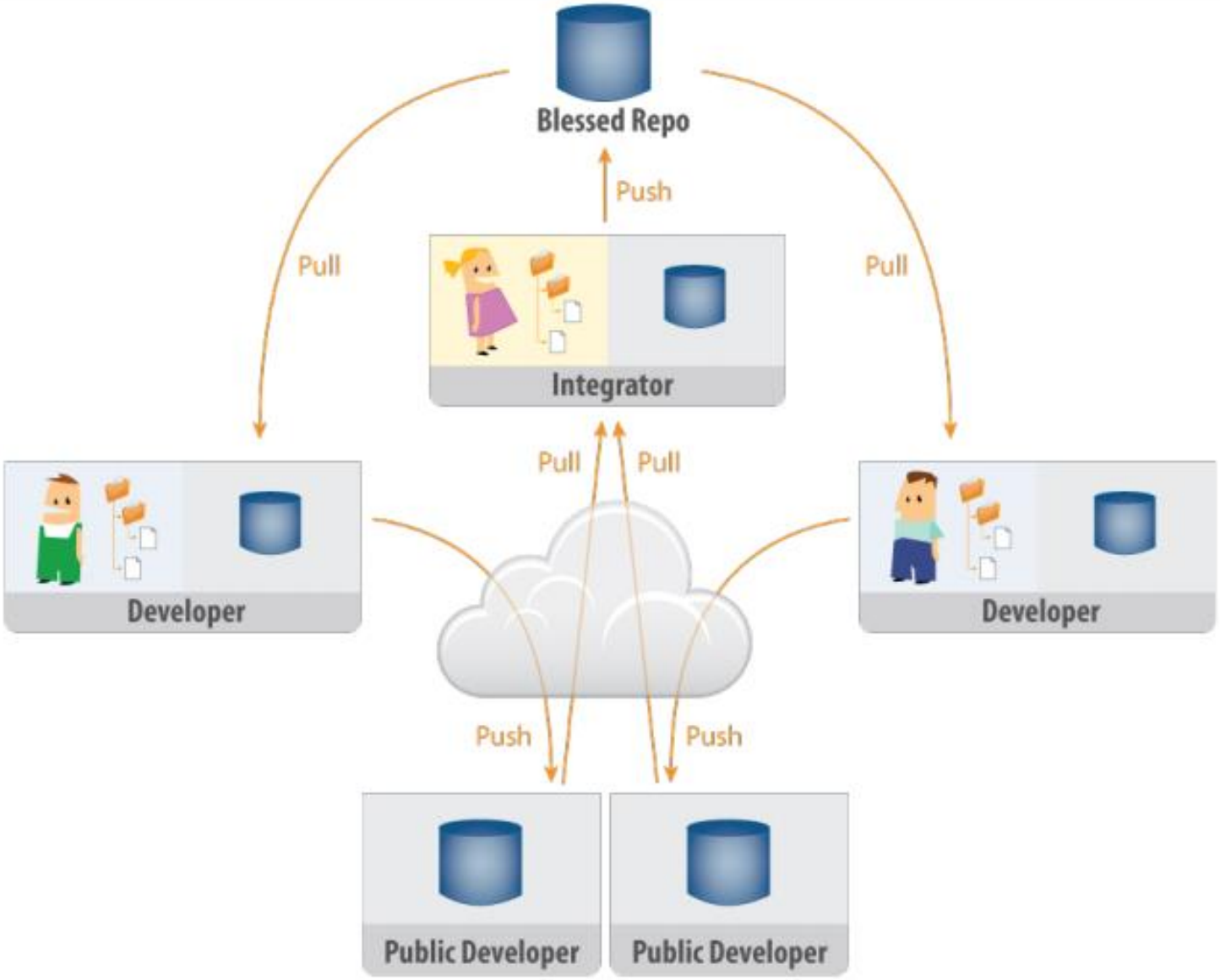
Thanks!

Backup slides

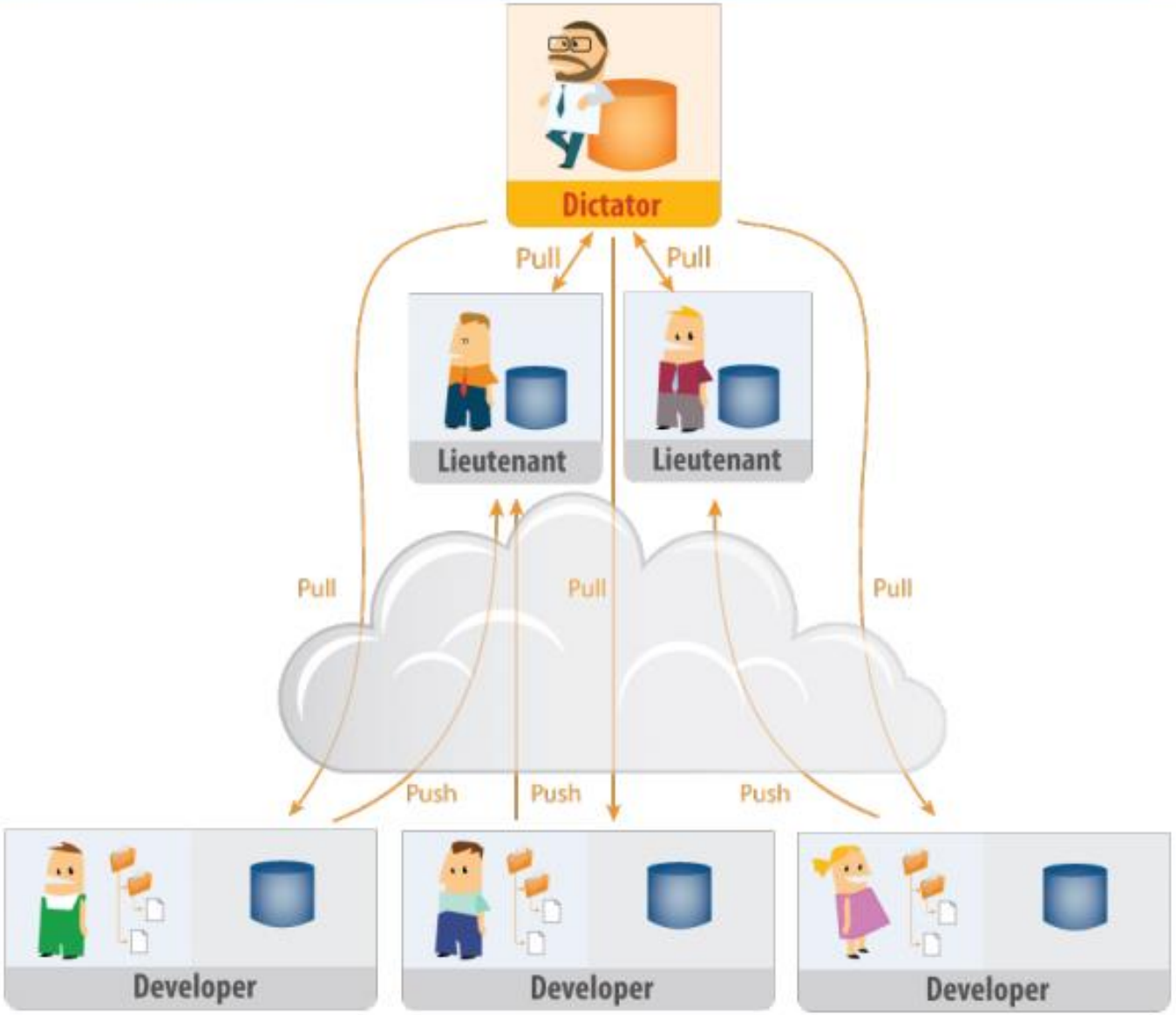
Centralized Workflow



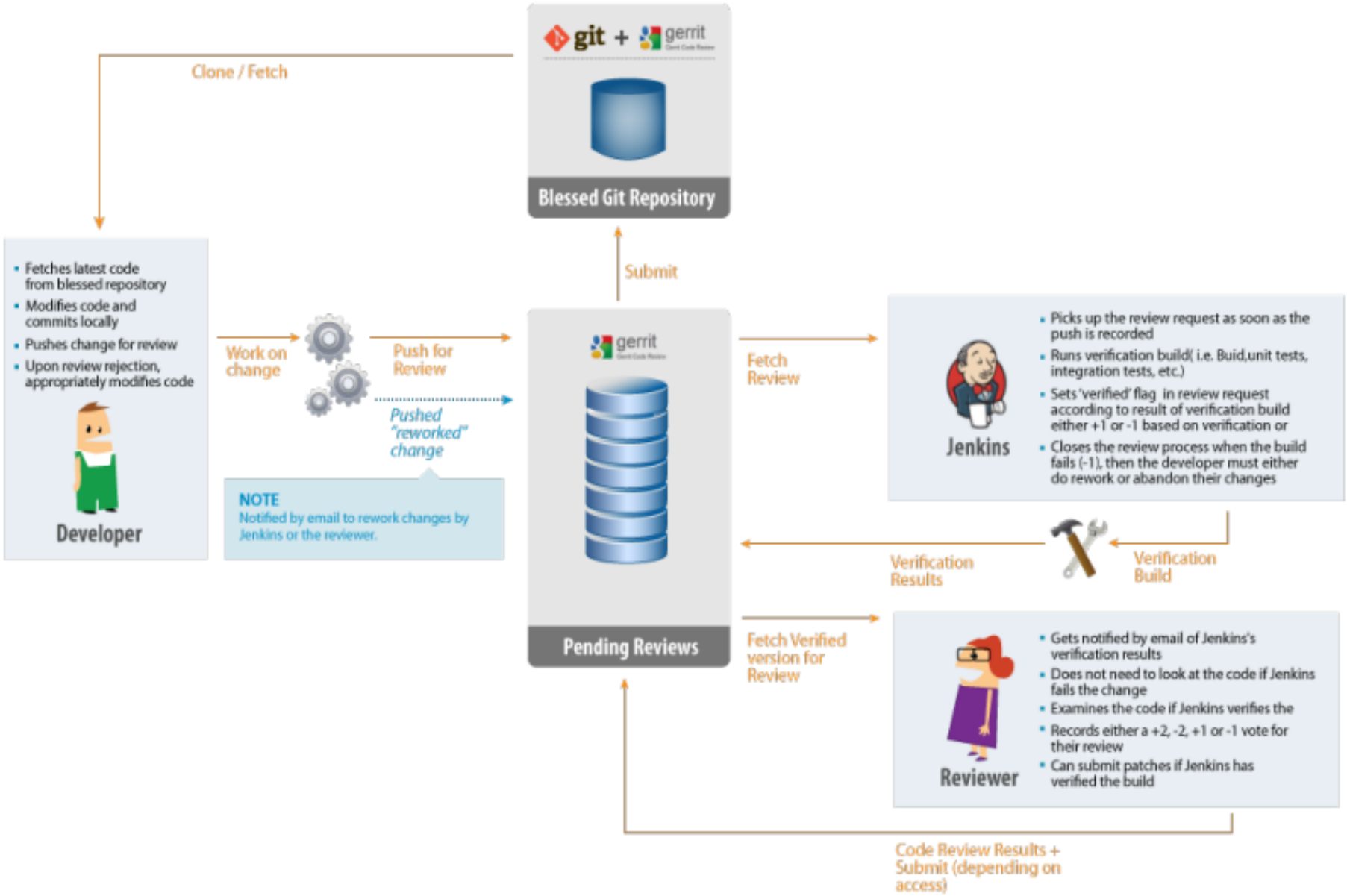
Integrators Workflow



Dictator / Lieutenants Workflow



Gerrit Code Review Workflow



Detached HEAD

If you checkout any commit SHA1, tag, or remote-tracking branch then you will end up having a “detached HEAD“:

```
$ git checkout 494e2cb73ed6424b27f9766bf8a2cb29770a1e7e
Note: checking out '494e2cb73ed6424b27f9766bf8a2cb29770a1e7e'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 494e2cb... Added README file
```

Git stash

You may be in a state where you have some changes that are not ready for committing, but you need to change branches in order to work on something else.

```
sheta@SHETA-THINK ~/my-project (fix-off-by-one)
$ git status
# On branch fix-off-by-one
# Changes to be commited:
#     (use "git reset HEAD <file>…" to unstage)
#
#       modified:    README.txt


$ git checkout master
error: Your local changes to the following files would be overwritten by checkout

        README.txt
Please, commit your changes or stash them before you can switch branches.
Aborting
```

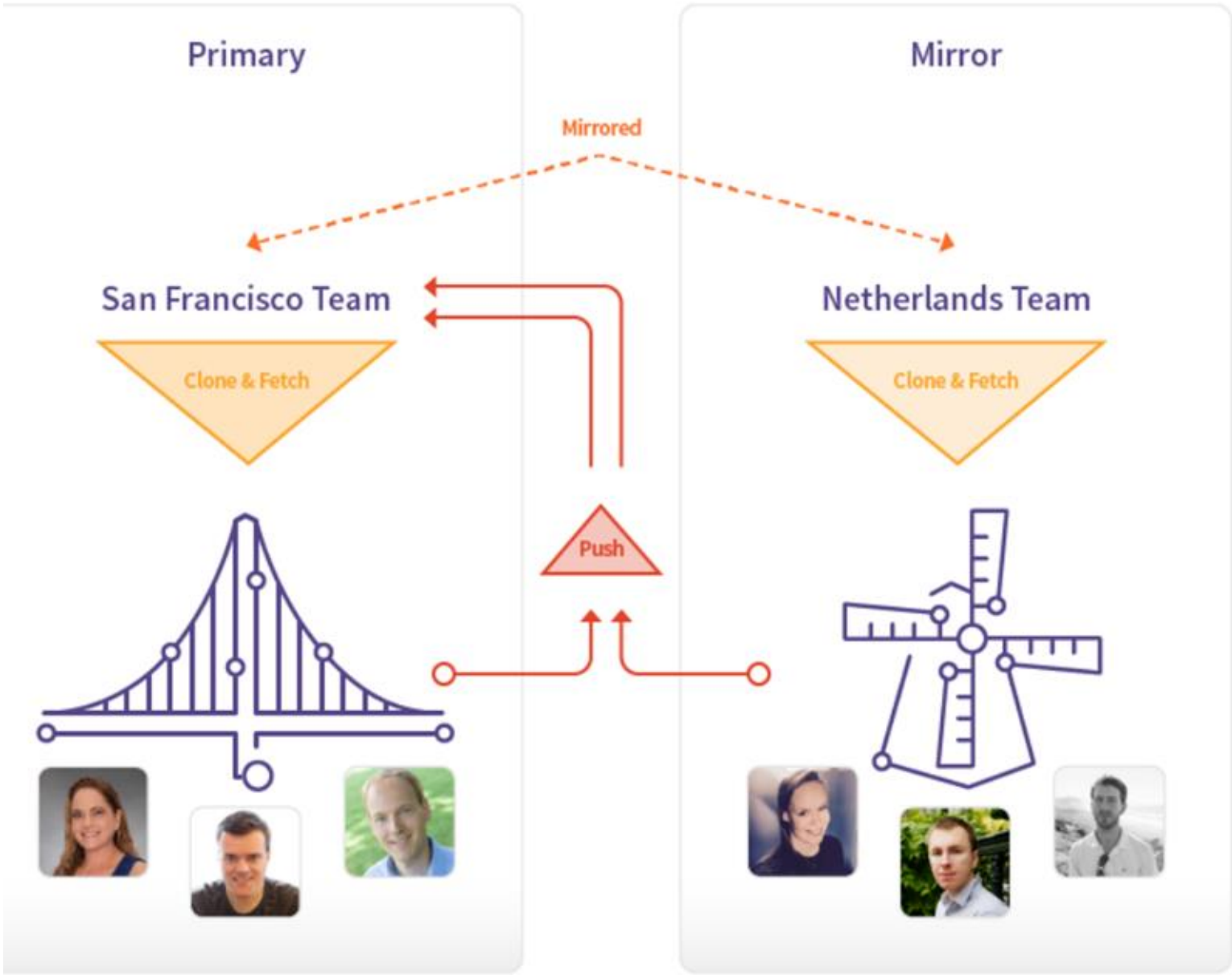
`git stash` takes current state of your working directory (what is staged, modified, etc.) and saves it as a stack of unfinished changes in **`refs/stash`**.

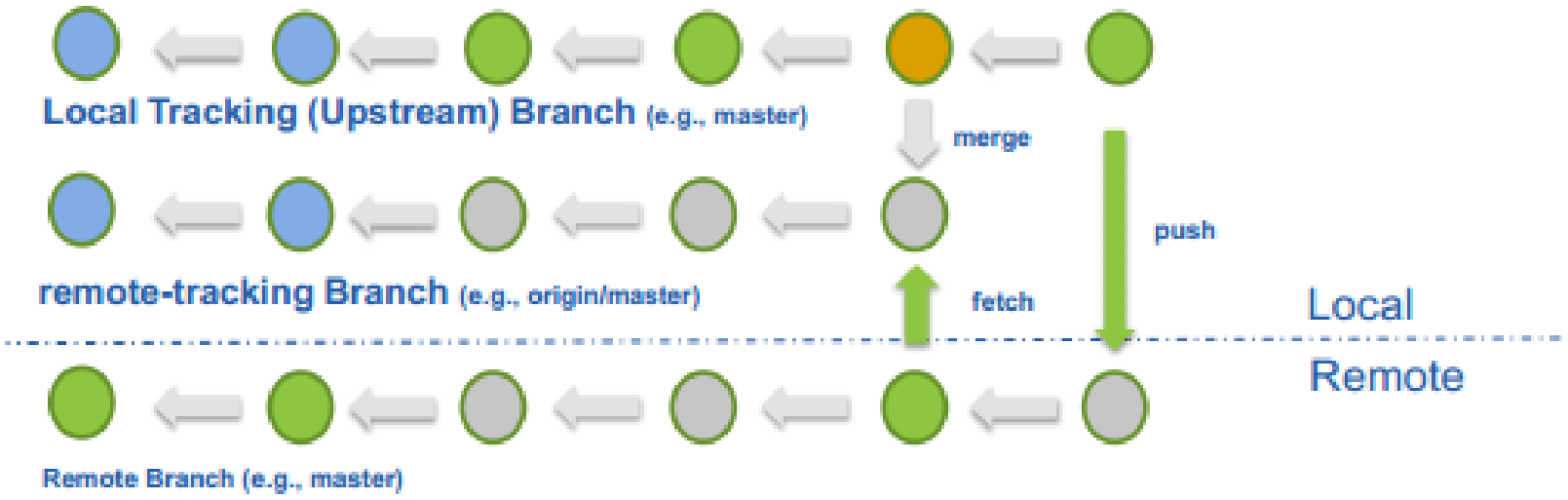
```
$ git stash save --all
Saved working directory and index state WIP on fix-off-by-one: ef2f6c3 Release r
e added
HEAD is now at ef2f6c3 Release note added
```

Later you can switch back to the previous branch and apply your saved changes to your working tree to have it exactly the way you had it prior to stashing your changes. You should

Git Master->Slave

Mirroring



What are tracking and remote-tracking branches?

- The combination of these branches defines a relationship between a local branch and one in the remote repository.
- When a repository is cloned, Git automatically creates **remote-tracking branches** (e.g., origin/master) for the remote branches and a **tracking branch** (e.g., master) to allow for local changes in relationship to the remote branch