Advanced Git and GitHub best practices for software developers

Micael Oliveira and Harshula Jayasuriya

ACCESS Training Day 2024

Advanced Git

Disclaimers

- This tutorial focuses on concepts and ideas, not commands
- I don't know all the Git commands and options!

ACCESS
National Research Infrastructure

The typical introduction to Git

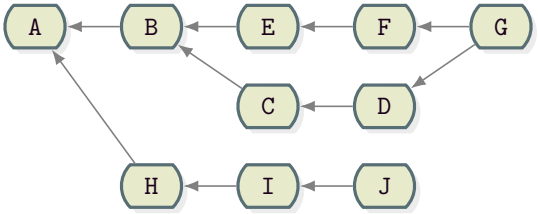
- Git is a version control system
- `git clone` to get a local copy of a remote repository
- `git add` and `git commit` to add changes
- `git pull` to get new changes from the remote repository
- `git push` to add your changes to the remote repository
- `git branch` and `git checkout` to create and change branches
- `git merge` to include changes from one branch into another branch

When do things usually start to go wrong?

- Need to fix mistakes
- Trying to use `git rebase`
- Using complex workflows

An alternative introduction to Git

- Git is a tool to create and manage graphs
- Commits are nodes and connections between commits are edges
- Git graphs are directed acyclic graphs (DAG)
- The graph tells us a story



Commits

What is included in a commit?

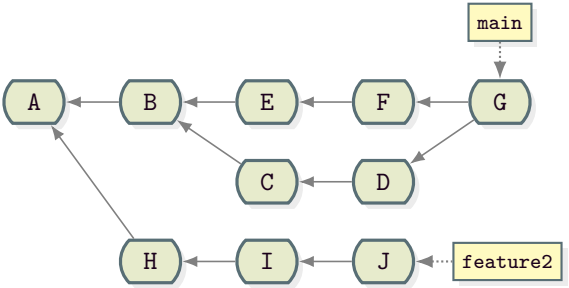
- A snapshot of the sources
- A timestamp
- A log message
- Zero or more parent commits

Two commits whose content differ in any way are **different** commits!

Note 1: A merge commit has two or more parent commits.
Note 2: The initial commit has no parent commit.
Note 3: Commits and all other Git objects are uniquely identified by a hash.

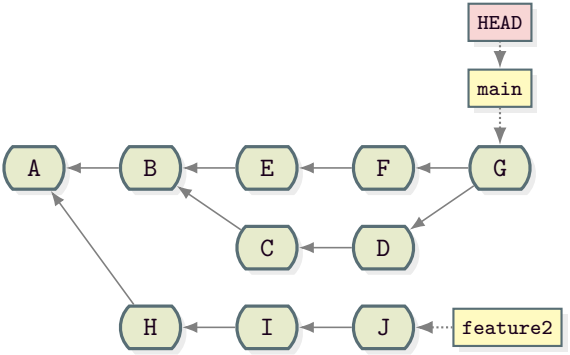
Branches and tags

- A branch is a reference to a commit



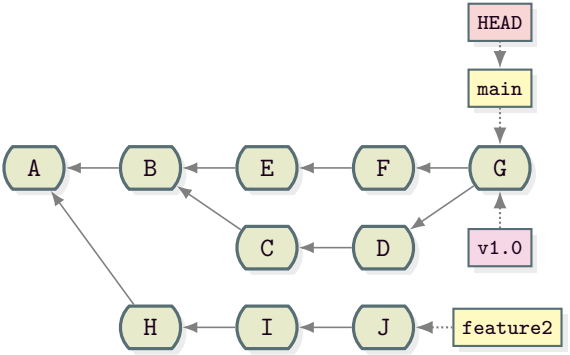
Branches and tags

- A branch is a reference to a commit
- The `HEAD` is a special reference to the commit we are currently on



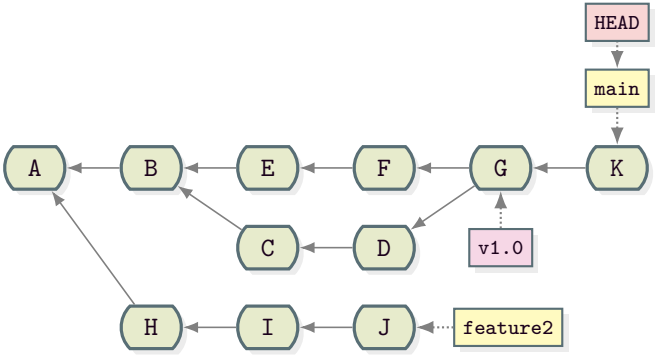
Branches and tags

- A branch is a reference to a commit
- The `HEAD` is a special reference to the commit we are currently on
- A tag is an immutable reference to a commit



Branches and tags

- A branch is a reference to a commit
- The `HEAD` is a special reference to the commit we are currently on
- A tag is an immutable reference to a commit
- Adding a commit to a branch updates the reference



Branches and tags

- A branch is a reference to a commit
- The `HEAD` is a special reference to the commit we are currently on
- A tag is an immutable reference to a commit
- Adding a commit to a branch updates the reference
- Branches and tags are often interchangeable!

Example:

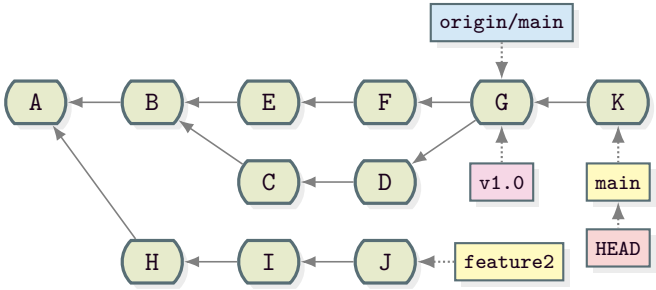
```
$ git branch foo bar
```

- Creates a new branch named `foo` using `bar` as a starting point.
- `bar` can be any commit-ish object: branch, tag, commit, etc.

ACCESS
National Research Infrastructure

Remotes and remote references

- Remote repositories are versions of our repositories that are stored somewhere else, typically on a server that can be reached over the Internet
- Remote references are references to a commit on a remote repository
- When cloning a repository, a remote named “origin” is created automatically pointing to the original repository



Visualising the DAG

1. View the log using a dog:

```
$ git log --all --decorate --oneline --graph
* bcd9864 (HEAD -> main) Commit K
*   402a92c (tag: v.1.0, origin/main, origin/HEAD) Merge commit G
|\
| * 6b926ae (feature1) Commit D
| * 4bb4450 Commit C
* | 458be76 Commit F
* | acf0e4a Commit E
|/
* 45f537f Commit B
| * 4b579a1 (feature2) Commit J
| * 8894a8b Commit I
| * aa49b65 Commit H
|/
* 33a4139 Commit A
```

2. External tools: Gitk, GitKraken, etc

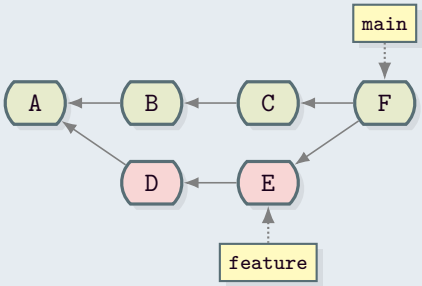
Exercises

- Clone the following repository from GitHub:
  `git@github.com:ACCESS-NRI/training-day-2024-advanced_git.git`
- Inspect the DAG. Can you identify the existing branches and tags? How many merge commits are there?
- Compare the last commit in branch `undo` with the parent of the last commit in branch `update_maintainers`. How do they differ? (Hint: use `git show` to display the commit metadata)

Merges

Merges are used to bring changes from one branch into another

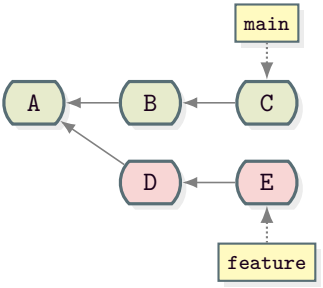
Example:



The merge commit adds the changes from the `feature` branch to the `main` branch.

How does this work in practice?

Basic merging: three-way merge



Comparing the contents of `C` and `E` is not enough:

Commit C

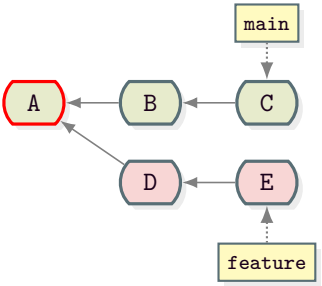
```
1 ...
2 ...
3 Print("Hello")
4 ...
```

Commit E

```
1 ...
2 ...
3 Print("Bye")
4 ...
```

Which one should we keep?

Basic merging: three-way merge



We need to compare `C` and `E` with their common ancestor `A`:

Commit A

```
1 ...
2 ...
3 Print("Hello")
4 ...
```

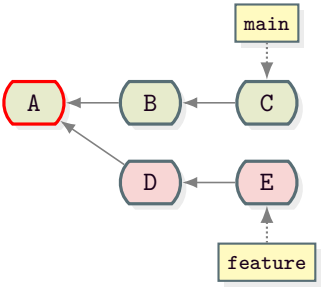
Commit C

```
1 ...
2 ...
3 Print("Hello")
4 ...
```

Commit E

```
1 ...
2 ...
3 Print("Bye")
4 ...
```

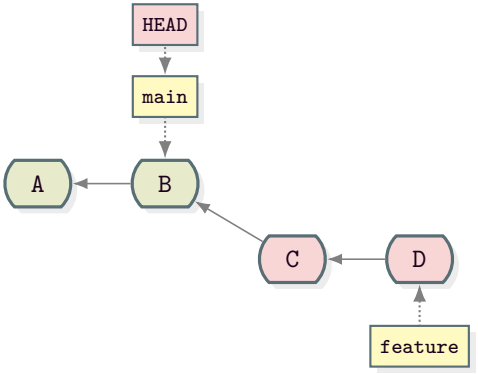
Basic merging: three-way merge



For a given line:

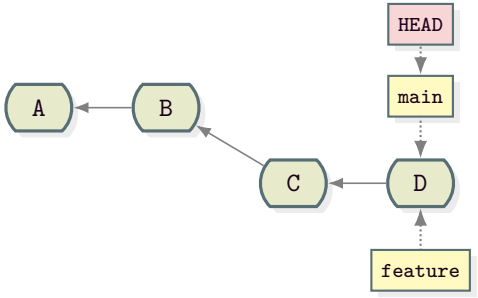
- If it is the same in `C` and `E`, keep either `C` or `E`
- If it is the same in `A` and `C`, but different in `E`, keep `E`
- If it is the same in `A` and `E`, but different in `C`, keep `C`
- If it is different in `A` and `C` and `D`, there is a conflict

Special case: fast-forward merge




```
$ git merge feature
```

Special case: fast-forward merge

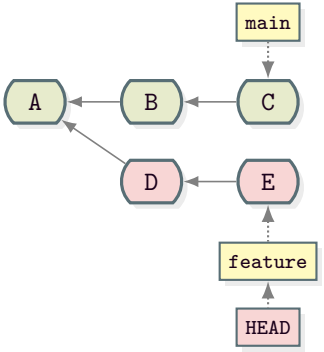


- No merge commit is created
- Use `git merge --no-ff` to force creation of merge commit

Exercises

- Do a `git diff` of branches `main` and `update_maintainers` and verify that the `package.py` file differs by two lines.
- Now merge `update_maintainers` into `main`. Can you explain why there is a conflict in one of the lines, but not the other?
- Resolve the conflict keeping the version from `main` and finish the merge.
- Look at the DAG again. Can you tell if merging branch `undo` into `main` will be a fast-forward merge?

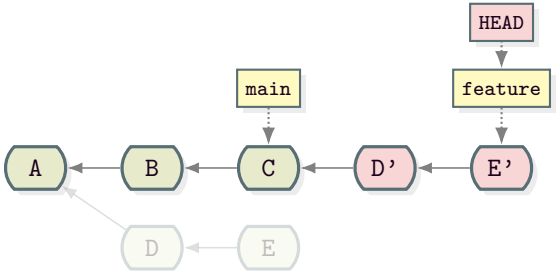
Rebases



How to change the parent of commit `D` from `A` to `C`?

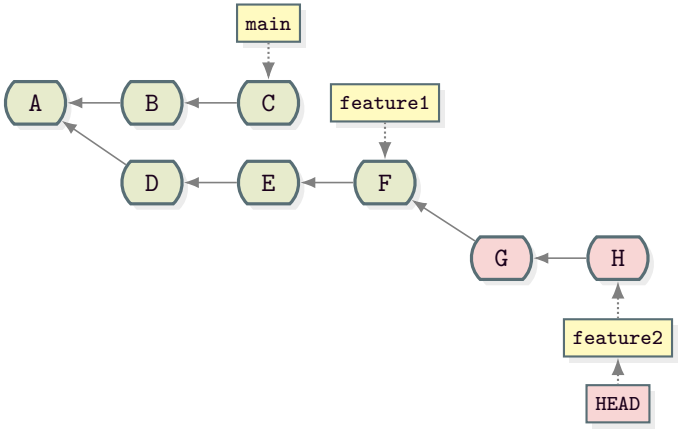
Rebases

```
$ git rebase main
```



- Commits `D` and `E` are reapplied on top of commit `C`
- Reapplying the commits works like a merge
- Conflicts might occur during the rebase

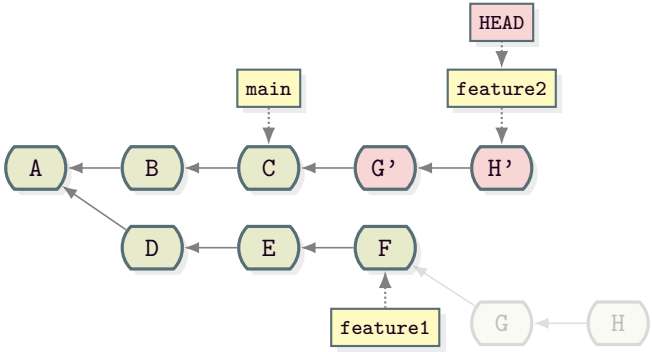
Rebases



How to change the parent of commit `G` from `F` to `C`?

Rebases

```
$ git rebase --onto main feature1 feature2
```

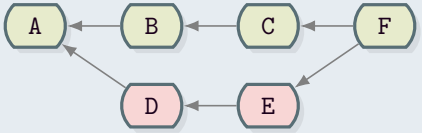


Exercises

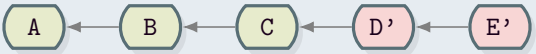
- Rebase branch `undo` onto `main`. Verify that after the rebase it is possible to fast-forward merge `undo` into `main`.
- Rebase branch `change_version` from `update_maintainers` onto `main` (hint: you need to use the `--onto` option). What happens if you don't specify `update_maintainers` when doing the rebase?

Merge vs Rebase

Merge



Rebase



Merge:

- Preserves history
- Easy to revert/reset

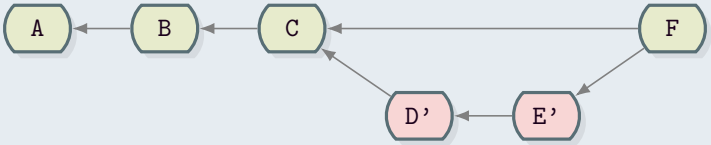
Rebase:

- Linear history
- Opportunity to clean up the branch history

Merge vs Rebase

- It's a matter of taste!
- Should the git history reflect the "real" development history?

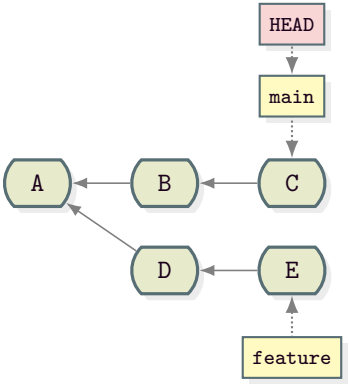
Compromise: merge with `--no-ff`, but only if fast-forward was possible



- A rebase is usually needed before merging

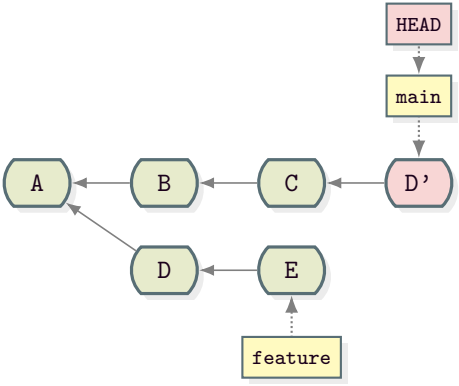
Cherry-pick

How to apply any commit on top of the `HEAD`?



Cherry-pick

```
$ git cherry-pick D
```



- Equivalent to rebasing a single commit onto `HEAD`
- Conflicts might occur during cherry-pick

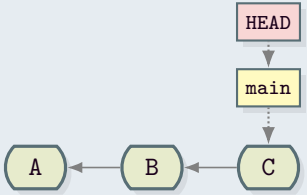
Undoing things and fixing mistakes

- There are several ways to undo/fix things in Git
- Some methods rewrite the history, i.e., they change how the history of a branch looks like
- Rewriting the history should be done with care
- Rule of thumb: do not change parts of the history that are public or that have been merged into other branches

Note

The following examples deal with fixing and undoing things that have been committed.
Use `git reset` or `git restore` to undo changes to the staging area.

Reverting changes

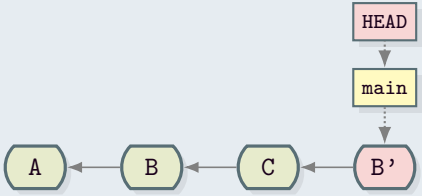


How to undo the changes from `C` without changing the history?

Undoing things and fixing mistakes

Reverting changes

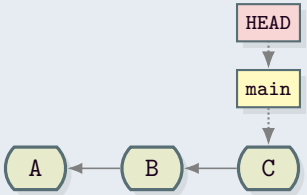
```
$ git revert HEAD
```



- Creates a commit that undoes the changes from another commit
- Does not change the history
- It is possible to revert any commit
- Can create conflicts that need to be resolved manually

Undoing things and fixing mistakes

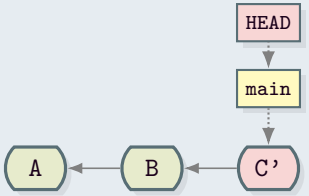
Amending the last commit



How to change commit `C`?

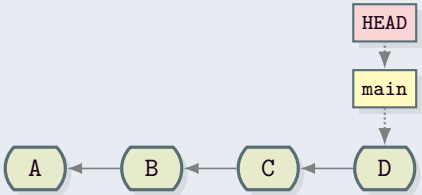
Amending the last commit

```
$ git add ...
$ git commit --amend
```



- Rewrites the history
- Can only be used to modify the last commit

Removing last commits from branch

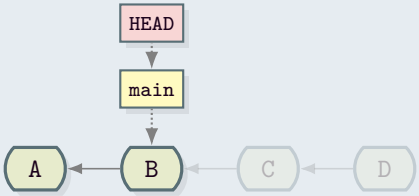


How to remove commits `C` and `D`?

Undoing things and fixing mistakes

Removing last commits from branch

```
$ git reset HEAD~2
```



- Rewrites the history
- By default, `git reset` leaves the working files unchanged
- Use `--hard` to also change the working files
- Commits are still in the repository (`git merge D` will bring the branch back to previous state)

Undoing things and fixing mistakes

Changing many commits at once: interactive rebase

```
$ git rebase -i foo
```

- Rebase all commits that came after `foo` onto `foo`
- Allows to modify, delete, reorder and squash commits
- Before running, command provides an editable list of commits being rebased

Undoing things and fixing mistakes

Changing many commits at once: interactive rebase

```
pick f7f3f6d First commit
pick 310154e Second commit
pick a5f4a0d Another commit

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
...
```

Exercises

- The changes from the last commit in branch `mistake` are wrong (version should be `1.0.0`, not `1.1.0`). Try to fix the error using:
  - `git commit --amend`
  - `git reset`
- Which of the above methods rewrite the history?
- Is `git revert` appropriate for fixing the above mistake?

ACCESS
National Research Infrastructure

Git Workflows

Workflows

Why?

- Git is very flexible and powerfull
- Workflows are recipes and recommendations to use Git in a consistent way
- They are necessary to develop code in a collaborative way
- They are necessary to prepare releases and hotfixes

Workflows

Good workflows:

- Scale with the number of developers
- Do not impose any large overhead
- Prevent mistakes or allow to easily fix them

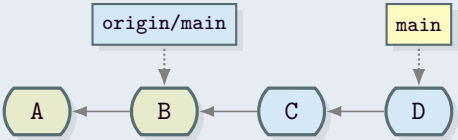
Prelude

Some assumptions in the following examples:

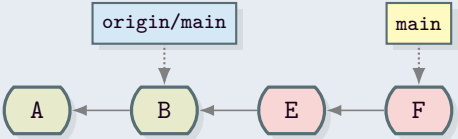
- There is always a remote repository that represents the official project
- Each developer has a local repository
- Unless otherwise stated, local repositories are clones of the official repository
- All operations are done **locally** - GitHub equivalents will be presented in the next section

Centralized workflow

Anne



Bob

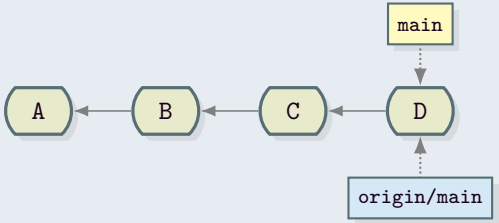


Centralized workflow

- Changes are published by pushing them to the official repository

Anne

```
$ git push origin main
```

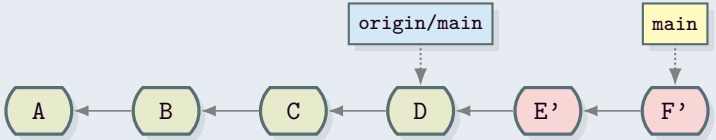


Centralized workflow

- Bob needs to get Anne's changes before publishing his own changes
- A rebase is necessary, otherwise the pull would fail

Bob

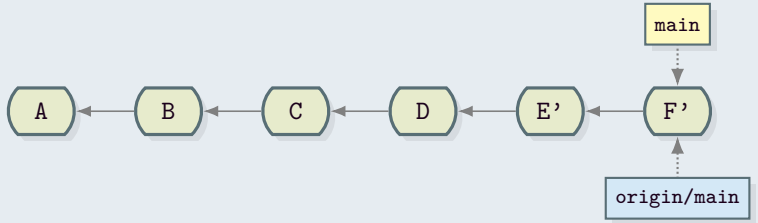
```
$ git pull --rebase
```



Centralized workflow

Bob

```
$ git push origin main
```

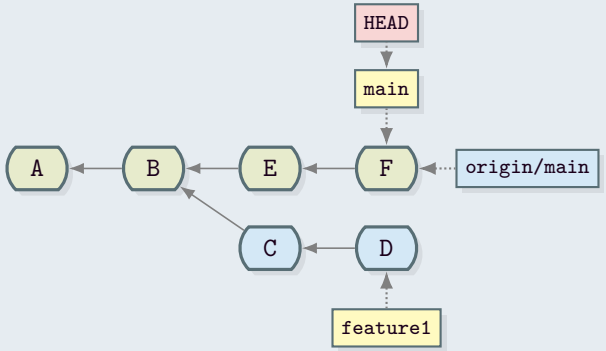


- Each developer should only work on one feature at a time

Feature branch workflow

- Feature branches are branched from main

Anne

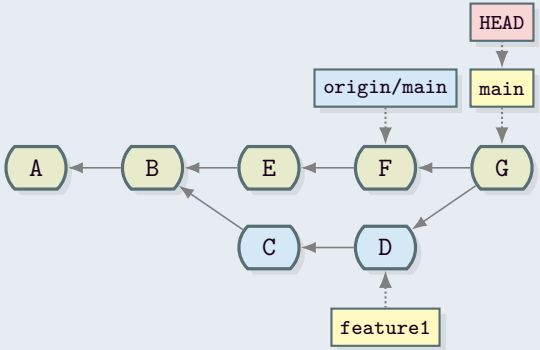


Feature branch workflow

- Feature branches are merged into main

Anne

```
$ git merge feature1
```

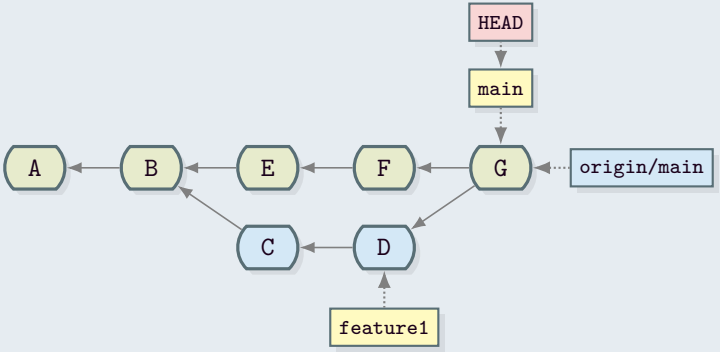


Feature branch workflow

- The result must be published to the official repository

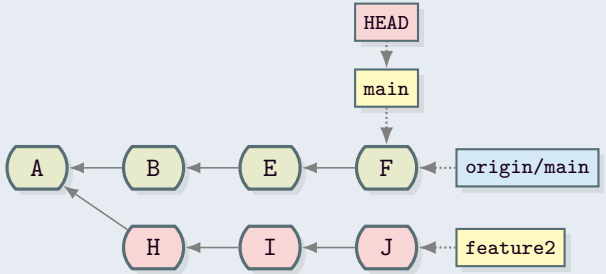
Anne

```
$ git push
```



Feature branch workflow

Bob

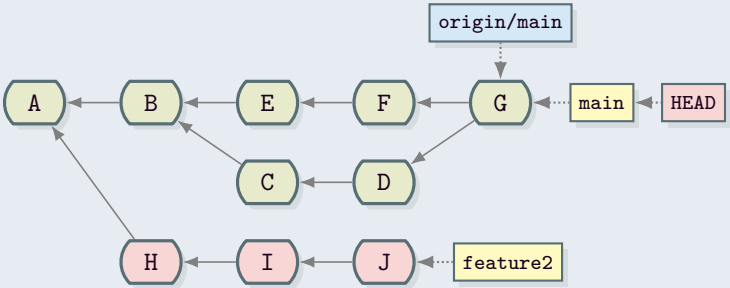


Feature branch workflow

- Always pull latest commits to main before merging!

Bob

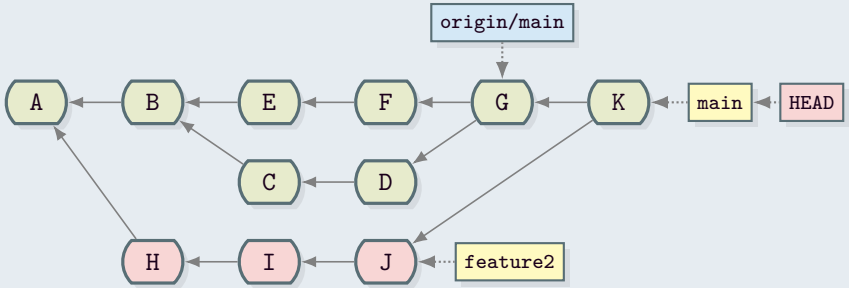
```
$ git pull
```



Feature branch workflow

Bob

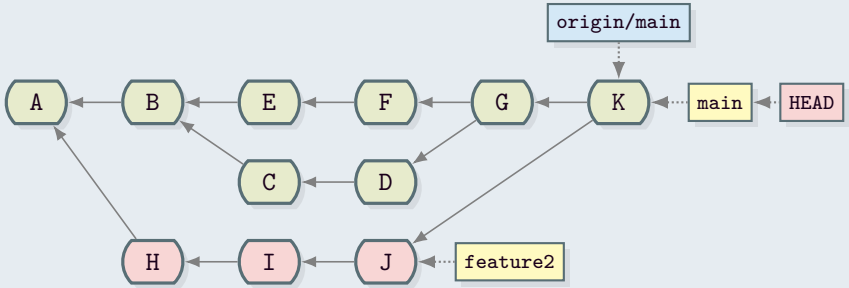
```
$ git merge feature2
```



Feature branch workflow

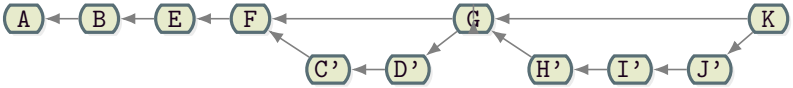
Bob

```
$ git push
```



Feature branch workflow

- Work on different features is independent
- Feature branches should be short lived
- Rebasing the feature branch just before merging keeps the history simpler:



- It's good practice to delete feature branches once they have been merged

GitHub

GitHub in a nutshell

- Hosting service for Git repositories
- Forks: special repositories linked to another repository
- Pull requests: special web interface for merges
- Issues: used for bug reports, feature requests, project tasks, etc
- Has its own CI service (Actions), but other CI services can be used

ACCESS
National Research Infrastructure

Forks and pull requests

- A fork is a server-side clone of a git repository
- Branches from a fork can easily be merged into the original repository using pull requests
- Allows developers to contribute to the original repository without having write permissions
- Can easily be integrated into workflows that use feature branches

ACCESS
National Research Infrastructure

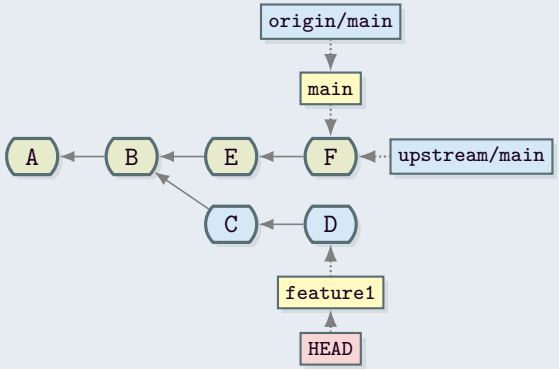
Feature branch workflow using forks and pull requests

- Each user has a fork of the official repository
- Users work locally on a clone of their fork
- In the following, the `origin` remote points to the fork
- We follow a common convention and call the remote pointing to the official repository `upstream`

Feature branch workflow using forks and pull requests

- Users still create the feature branch and commit to it locally

Anne

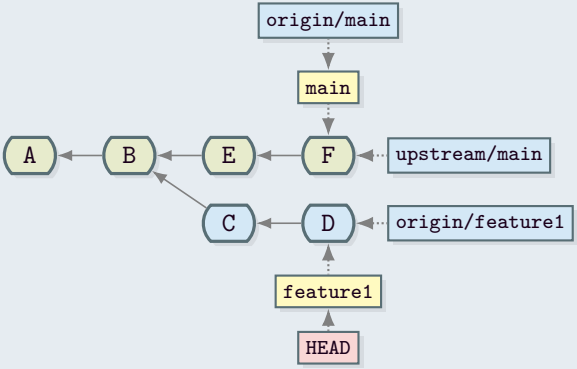


Feature branch workflow using forks and pull requests

- Feature branches are pushed to the fork

Anne

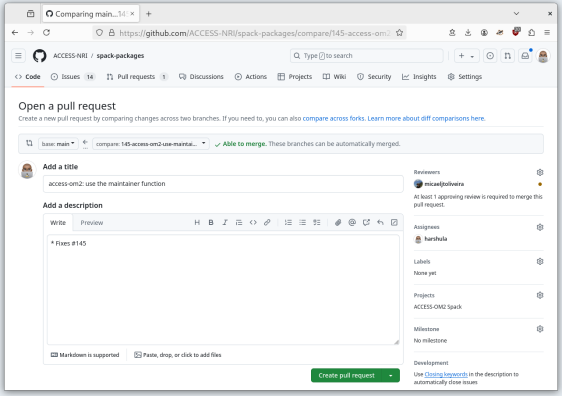
```
$ git push --set-upstream origin feature1
```



Feature branch workflow using forks and pull requests

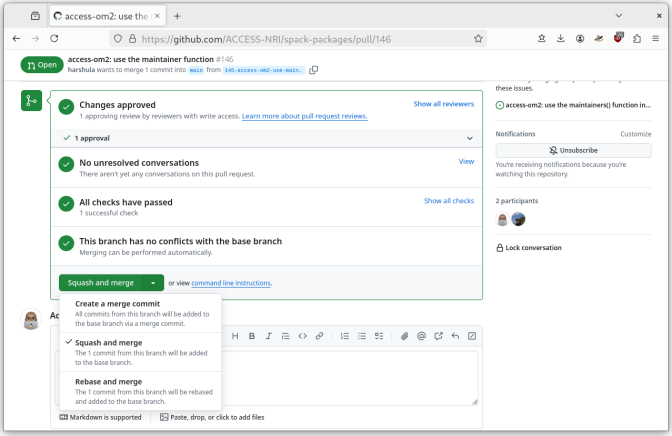
- Feature branches are merged into `main` through a pull request

Anne



Feature branch workflow using forks and pull requests

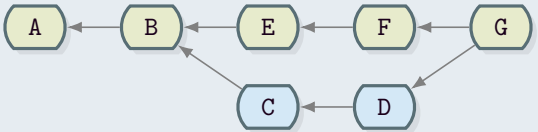
- GitHub provides different strategies to merge the feature branch into the target branch



Feature branch workflow using forks and pull requests

- GitHub provides different strategies to merge the feature branch into the target branch

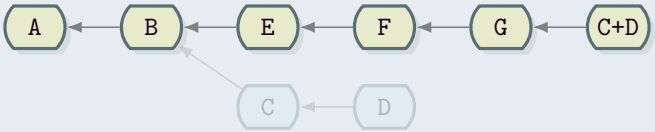
Merge commit



Feature branch workflow using forks and pull requests

- GitHub provides different strategies to merge the feature branch into the target branch

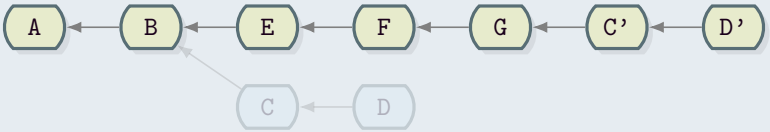
Squash and merge



Feature branch workflow using forks and pull requests

- GitHub provides different strategies to merge the feature branch into the target branch

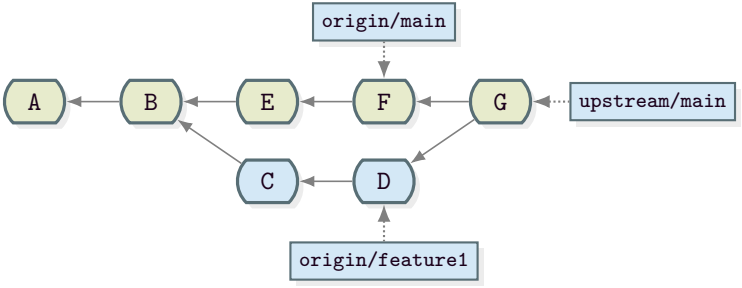
Rebase and merge



Note: GitHub will let you know if there is a potential conflict.

Feature branch workflow using forks and pull requests

- Once the pull request is merged, the new feature is included in the official repository



Note 1: `origin` does not track upstream automatically!
Note 2: User needs to update the `main` branch on `origin` and locally.
Note 3: Workflow works in the same way for all users.

GitHub best practices

Always protect your main branch:

- No force pushes
- No direct pushes to the branch, require pull requests instead
- Restrict users who can merge into the branch
- Require CI to pass before merging

ACCESS
National Research Infrastructure

GitHub best practices

Pull requests:

- Start by creating an issue explaining the problem you want to solve or the new feature you want to implement
- Get familiar with the project's guidelines and standards for contributions
- Take the time to write a good description and choose an appropriate title
- Link the pull request to the relevant issues (e.g., `Closes #23`)
- Document your code
- Add new tests or extend existing tests to cover your changes
- Make sure the Git history of your branch is easy to understand

ACCESS
National Research Infrastructure

GitHub best practices

Code review

- Be polite!
- It's okay to ask questions and/or clarifications
- Be constructive and provide suggestions to improve the code
- Mind that not every contributor has the same level of expertise
- Check documentation
- Check that code is covered by existing and/or new tests

ACCESS
National Research Infrastructure

Advanced Git Workflows

Releases and Hotfixes

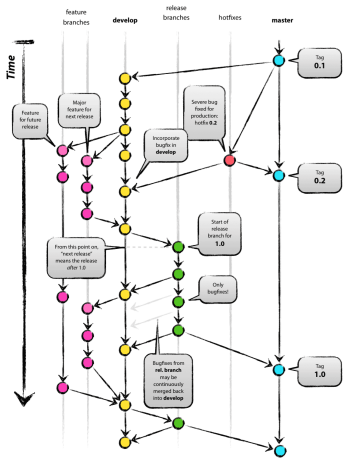
- Some software is periodically released for production: production release
- A production release usually includes new features and bugfixes
- Production releases are usually less frequent than the addition of new features (exception: continuous delivery)
- Hotfixes are releases that include only critical bugfixes
- Hotfix releases should not include new features

Releases and Hotfixes

How to do releases with Git?

- Tags are used to mark releases
- Workflow needs to incorporate some procedure to create the releases and the hotfixes
- Suitable procedure depends on several things. For example:
  - How often one does a new release
  - Is a new production release based on the previous release?
  - How many simultaneous releases are maintained?

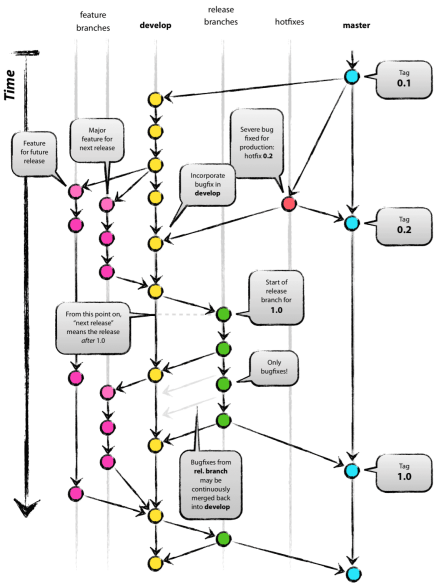
GitFlow



nvie.com/posts/a-successful-git-branching-model/

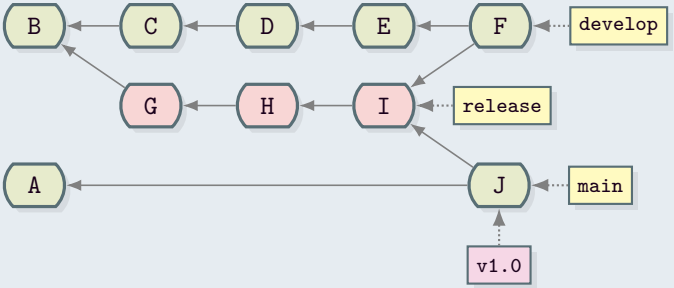
GitFlow

- There are two long-lived branches: main and develop
- There are three types of short-lived supporting branches:
  - Feature
  - Release
  - Hotfix
- Feature branches are branched from develop and merged into develop



GitFlow

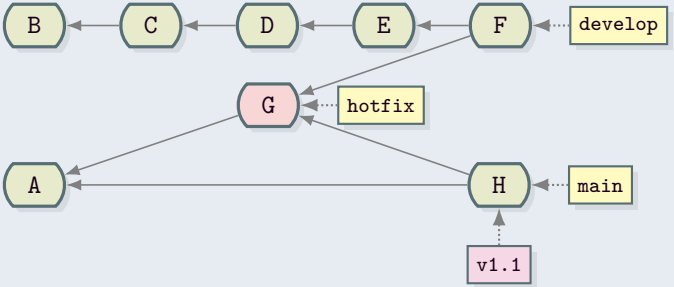
Release branches



- Release branch is branched from develop
- Branch is merged into main and develop with `--no--ff`
- Merged commit on main is tagged
- Release branch should only include bugfixes and changes needed to prepare the release

GitFlow

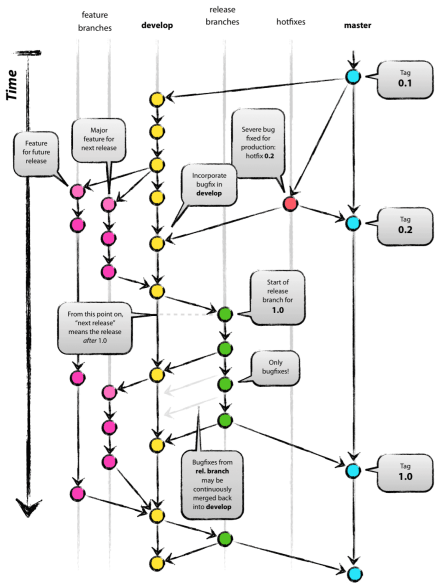
Hotfix branches



- Hotfix branch is branched from main to avoid including new features
- Branch is merged into main and develop with `--no--ff`
- Merged commit on main is tagged

GitFlow: a critique

- Quite complicated to implement and enforce
- Produces very complicated history
- Having two eternal branches is not really necessary
- No hotfix can be made for older releases



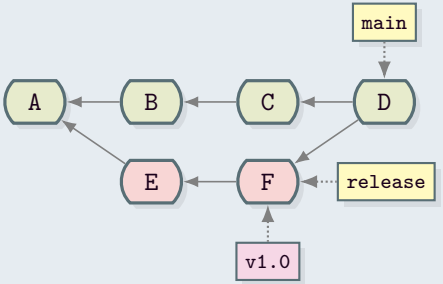
OneFlow

- Alternative to GitFlow
- Only one long-lived branch (main)
- Same support branches as GitFlow
- Feature branches are branched from and merged into main

`www.endoflineblog.com/oneflow-a-git-branching-model-and-workflow`

OneFlow

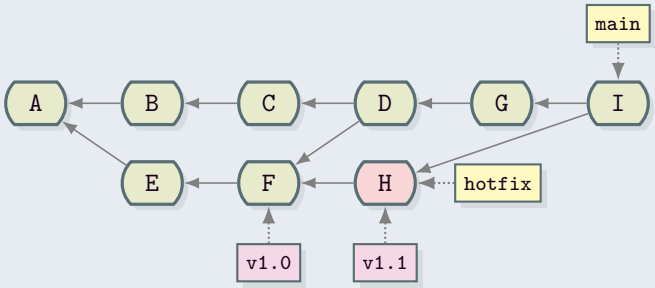
Release branches



- Release branch is branched from and merged into main
- Last commit of release branch is tagged

OneFlow

Hotfix branches



- Hotfix branch is branched from last release tag
- Hotfix branch is merged into main
- Last commit of hotfix branch is tagged

OneFlow

- Simpler than GitFlow
- Less merges than GitFlow
- It is able to do all that GitFlow can do
- It is possible to do hotfixes from older releases (exercise: think how this would work)