

# Git in real life

Alessandro Sciarra

CRC-TR 211 – Software Development Center  
PUNCH Young Academy – PUNCH4NFDI

24 May 2024

# Outline of the talk

- 1 Rewriting history
- 2 Git rebase
- 3 Git tag
- 4 Semantic versioning
- 5 Git-flow
- 6 Summary and outlook

Rewriting history

A very sharp knife!

# About rewriting history

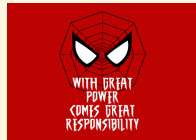
Yes, you can change the history of a repository!



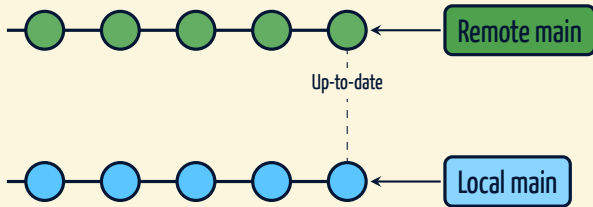
- **Very bad if there are different copies of the history** { E.g. branches, remote(s) }
- **If you rewrite shared history,**
  - it is generally hard to make the same change elsewhere and
  - merging (and hence pull/push) can lead to duplicated commits in history
- **Fine as long as**
  - yours is the only clone containing the rewritten history or
  - you work on a git project alone and you know what to do then
  - your **branch** is internal to the team and you inform people about rewriting

# About rewriting history

Yes, you can change the history of a repository!



- **Very bad if there are different copies of the history** { E.g. branches, remote(s) }
- If you rewrite shared history,
  - it is generally hard to make the same change elsewhere and
  - merging (and hence pull/push) can lead to duplicated commits in history

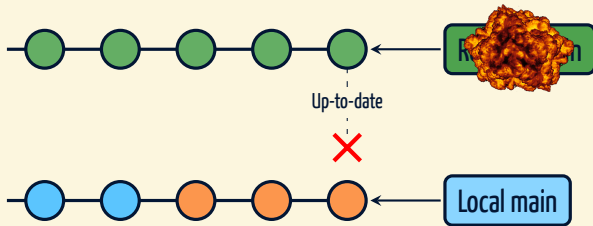


# About rewriting history

Yes, you can change the history of a repository!



- **Very bad if there are different copies of the history** { E.g. branches, remote(s) }
- If you rewrite shared history,
  - it is generally hard to make the same change elsewhere and
  - merging (and hence pull/push) can lead to duplicated commits in history



**As soon as you rewrite history contained elsewhere, you are in troubles!**

# Changing the last commit (message)

## 1 How can I fix a typo in the **last** commit message?

This procedure will change history!

Said differently, feel free to do it if you **did not** share history!

```
$ git commit -m "Added engine implementation"
[airplane f8b0c25] Added engine implementation
24 files changed, 1340 insertions(+), 476 deletions(-)
# Ops! I used a verb in past form, ✎not conforming!
```



# Changing the last commit (message)

## 1 How can I fix a typo in the last commit message?

This procedure will change history!

Said differently, feel free to do it if you **did not** share history!

```
$ git commit -m "Added engine implementation"
[airplane f8b0c25] Added engine implementation
24 files changed, 1340 insertions(+), 476 deletions(-)
# Ops! I used a verb in past form, ✎not conforming!
```

# With clean staging area:

```
$ git commit --amend -m "Add engine implementation"
[airplane 33f53f4] Add engine implementation
Date: Fri Nov 25 08:02:22 2022 +0100
24 files changed, 1340 insertions(+), 476 deletions(-)
```

# Changing the last commit (content)

2 I forgot some changes in the **last** commit! And now?

This procedure will change history!

Said differently, feel free to do it if you **did not** share history!

```
$ git commit -m "Review take-off system"
[airplane e1df32a] Review take-off system
1 file changed, 230 insertions(+), 61 deletions(-)
# Ops! I forgot a file!
```

# Changing the last commit (content)

2 I forgot some changes in the **last** commit! And now?

This procedure will change history!

Said differently, feel free to do it if you **did not** share history!

```
$ git commit -m "Review take-off system"
[airplane e1df32a] Review take-off system
1 file changed, 230 insertions(+), 61 deletions(-)
# Ops! I forgot a file!
```

```
$ git add wheels_electronic.h
$ git commit --amend --no-edit
[airplane c13e34f] Review take-off system
Date: Fri Nov 25 08:45:18 2022 +0100
2 files changed, 345 insertions(+), 88 deletions(-)
```

# Changing the last commit (content)

2 I forgot some changes in the **last** commit! And now?

This procedure will change history!

Said differently, feel free to do it if you **did not** share history!

```
$ git commit -m "Review take-off system"
[airplane e1df32a] Review take-off system
1 file changed, 230 insertions(+), 61 deletions(-)
# Ops! I forgot a file!
```

```
$ git add wheels_electronic.h
$ git commit --amend -C HEAD # use original commit timestamp
[airplane c13e34f] Review take-off system
Date: Fri Nov 25 08:33:01 2022 +0100
2 files changed, 345 insertions(+), 88 deletions(-)
```

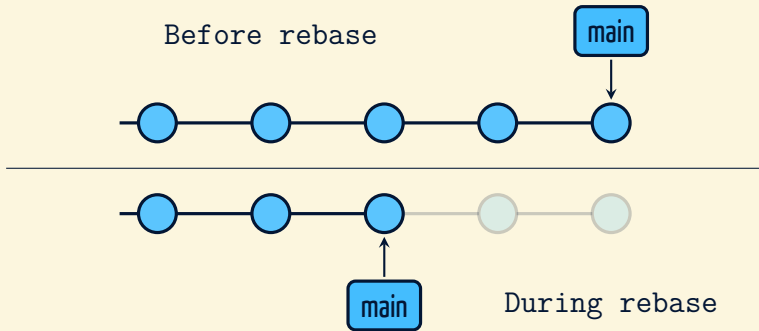
# Git rebase

# Which is the abstract idea of a rebase?



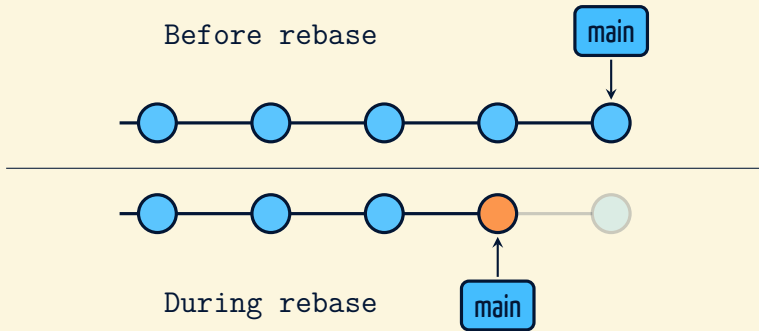
# Which is the abstract idea of a rebase?

- 1 Go back in history till a **given** point



# Which is the abstract idea of a rebase?

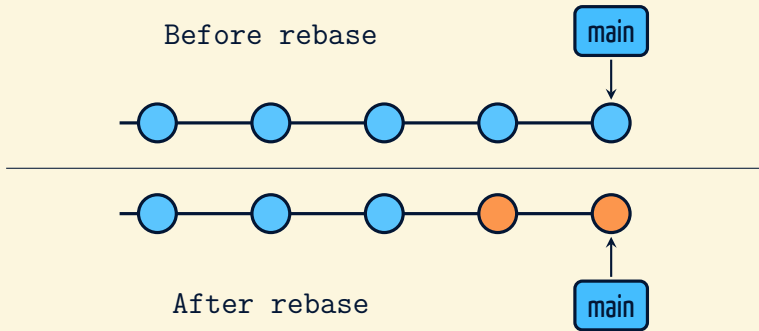
- 1 Go back in history till a **given** point
- 2 Replay **chosen** commits in their order
  - ↳ possibly taking action on applied commit as specified





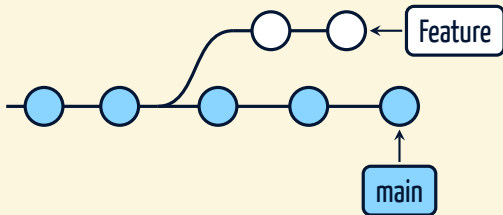
# Which is the abstract idea of a rebase?

- 1 Go back in history till a **given** point
- 2 Replay **chosen** commits in their order
  - ↳ possibly taking action on applied commit as specified

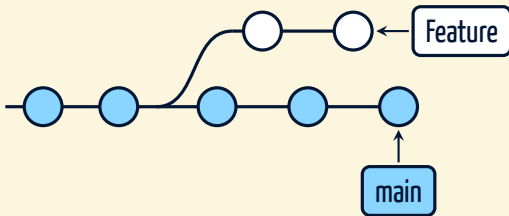


# Rebasing instead of (three-way) merging

Three way merge from Feature:

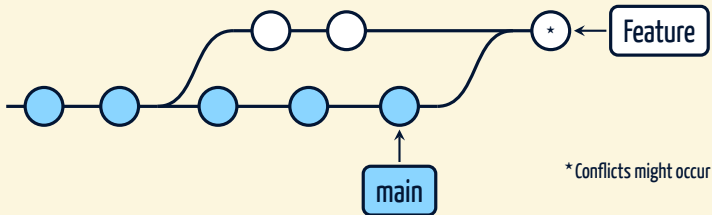


Rebase from Feature:

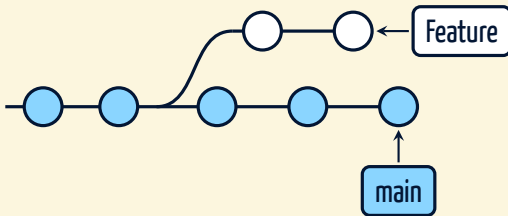


# Rebasing instead of (three-way) merging

Three way merge from Feature:

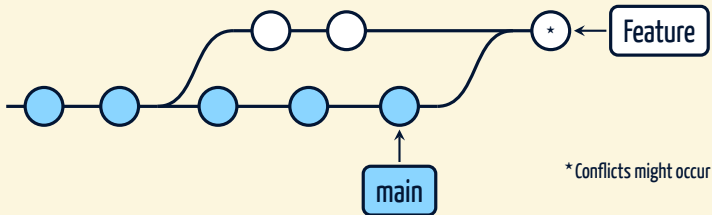


Rebase from Feature:

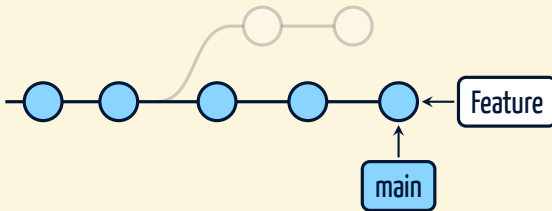


# Rebasing instead of (three-way) merging

Three way merge from Feature:

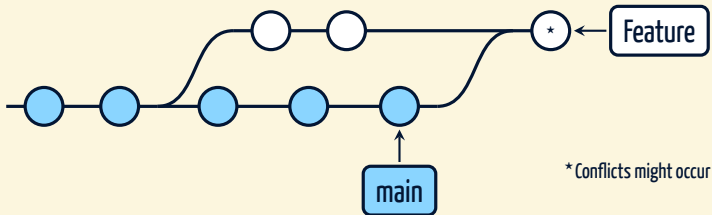


Rebase from Feature:

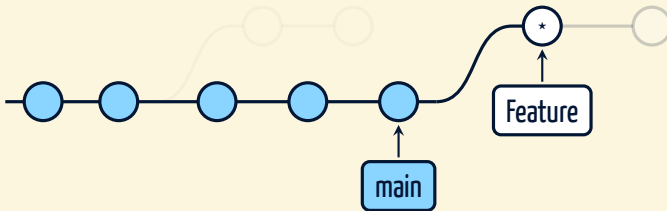


# Rebasing instead of (three-way) merging

Three way merge from Feature:

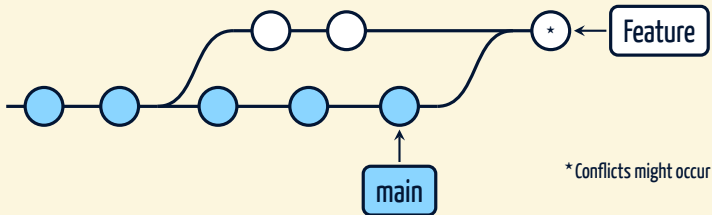


Rebase from Feature:

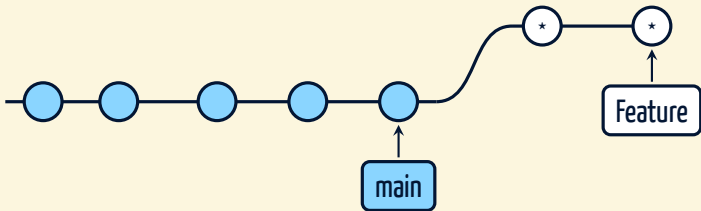


# Rebasing instead of (three-way) merging

Three way merge from Feature:



Rebase from Feature:



# Git rebase in its (almost) full glory

```
git rebase [-i] [--onto <newbase>] [<upstream> [<branch>]]
```

`-i` Set up the rebase interactively

`<newbase>` Where to apply the chosen commits  
↳ by default `<upstream>`

`<upstream>` Base history for the rebase to choose commits

`<branch>` The branch from which the rebase is done

## Conflicts might occur

- 1 Resolve them as usual and `git-add` the files
- 2 Run `git rebase --continue`

# Git rebase in its (almost) full glory

```
git rebase [-i] [--onto <newbase>] [<upstream> [<branch>]]
```

`-i` Set up the rebase interactively

`<newbase>` Where to apply the chosen commits  
↳ by default `<upstream>`

`<upstream>` Base history for the rebase to choose commits

`<branch>` The branch from which the rebase is done

## Conflicts might occur

- Run `git rebase --skip` to ignore commit
- Run `git rebase --abort` to end rebase

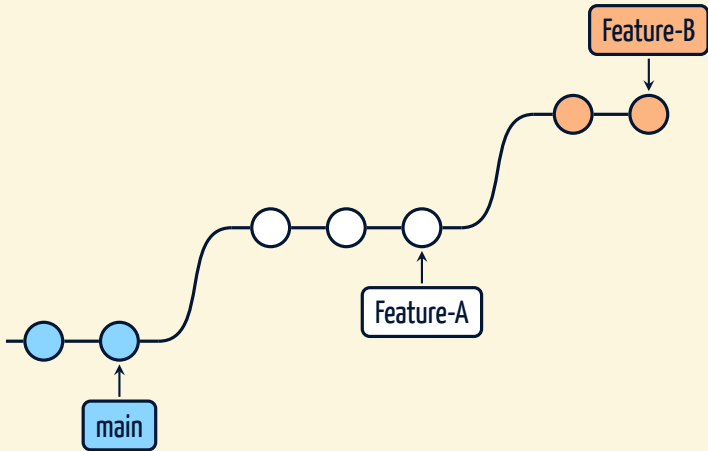


# Git rebase in its (almost) full glory

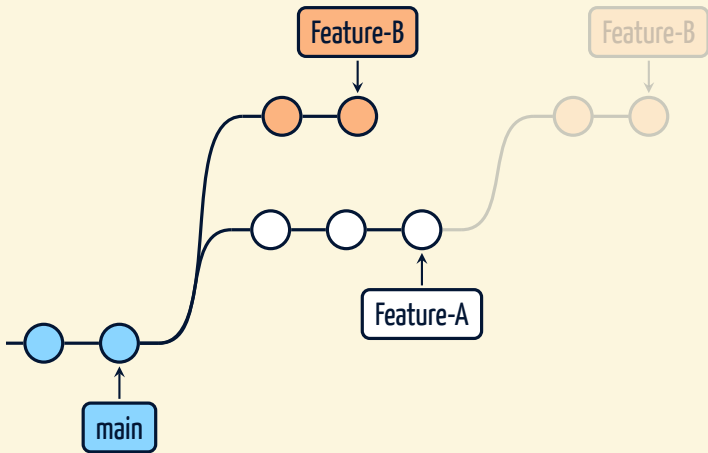
```
git rebase [-i] [--onto <newbase>] [<upstream> [<branch>]]
```

- 1 If `<branch>` is specified, git will switch to it
- 2 Commits in the current branch but that are not in `<upstream>` are saved to a temporary area
  - ↳ Use `git log <upstream>..HEAD` to see these commits
- 3 The current branch is reset to `<upstream>`
  - ↳ or `<newbase>` if the `--onto` option was supplied
- 4 The previously saved commits are then reapplied to the current branch, one by one, in order → **conflicts might occur!**
  - ↳ already existing commits are by default omitted

## Another example



# Another example



```
git rebase --onto main Feature-A Feature-B
```

# Back to rebasing instead of merging

Useful to keep history clean in repository

## If working **alone** on a branch

- 1 Get your work done
- 2 `git rebase main [your-branch]`
- 3 Then `git merge main` will be up-to-date
- 4 Switch to `main` and do a trivial merge

## Interactive rebase

- It allows to act on commits while re-applying them
- It offers further possibilities to tidy up work

# An example of interactive rebase

## Adjust history of recent local changes

```
$ git log --oneline -n 4
e499d89 Deploy engine turbo           # This should be rephrased
f8b0c25 Improve flaps of wings
dfb705b Make some wheels maintenance # Here we forgot a file
a0a3f28 Work on cockpit instruments
```

# An example of interactive rebase

## Adjust history of recent local changes

```
$ git log --oneline -n 4
e499d89 Deploy engine turbo          # This should be rephrased
f8b0c25 Improve flaps of wings
dfb705b Make some wheels maintenance # Here we forgot a file
a0a3f28 Work on cockpit instruments

$ git add wheel_test.cpp
$ git commit -m "This commit message does not matter"
[airplane 364ff12] This commit message does not matter
Date: Mon Nov 28 11:57:01 2022 +0100
1 files changed, 546 insertions(+), 810 deletions(-)
```

# An example of interactive rebase

## Adjust history of recent local changes

```
$ git log --oneline -n 4
e499d89 Deploy engine turbo          # This should be rephrased
f8b0c25 Improve flaps of wings
dfb705b Make some wheels maintenance # Here we forgot a file
a0a3f28 Work on cockpit instruments

$ git add wheel_test.cpp
$ git commit -m "This commit message does not matter"
[airplane 364ff12] This commit message does not matter
Date: Mon Nov 28 11:57:01 2022 +0100
1 files changed, 546 insertions(+), 810 deletions(-)

$ git rebase -i HEAD~5  # --> Edit, save, exit from editor
```

```

pick a0a3f28 Work on cockpit instruments #1
pick dfb705b Make some wheels maintenance #2
pick f8b0c25 Improve flaps of wings #3
pick e499d89 Deploy engine turbo #4
pick 364ff12 This commit message does not matter #5

# Rebase 9fdb3bd..e499d89 onto 9fdb3bd
#
# 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 [-C | -c] <commit> = like "squash" but keep only the previous
#                               commit's log message, unless -C is used, in which case
#                               keep only this commit's message; -c is same as -C but
#                               opens the editor
# 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.
# [...]

```



```

pick a0a3f28 Work on cockpit instruments #1
pick dfb705b Make some wheels maintenance #2
fixup 364ff12 This commit message does not matter #5
pick f8b0c25 Improve flaps of wings #3
edit e499d89 Deploy engine turbo #4

# Rebase 9fdb3bd..e499d89 onto 9fdb3bd
#
# 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 [-C | -c] <commit> = like "squash" but keep only the previous
#                               commit's log message, unless -C is used, in which case
#                               keep only this commit's message; -c is same as -C but
#                               opens the editor
# 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.
# [...]

```

# An example of interactive rebase

## Adjust history of recent local changes

```
$ git log --oneline -n 4
e499d89 Deploy engine turbo          # This should be rephrased
f8b0c25 Improve flaps of wings
dfb705b Make some wheels maintenance # Here we forgot a file
a0a3f28 Work on cockpit instruments

$ git add wheel_test.cpp
$ git commit -m "This commit message does not matter"
[airplane 364ff12] This commit message does not matter
Date: Mon Nov 28 11:57:01 2022 +0100
1 files changed, 546 insertions(+), 810 deletions(-)

$ git rebase -i HEAD~5 # --> Edit, save, exit from editor
# When asked for, rephrase commit as wished
Successfully rebased and updated refs/heads/airplane.
```

# An example of interactive rebase

## Adjust history of recent local changes

```
$ git log --oneline -n 4
e499d89 Deploy engine turbo          # This should be rephrased
f8b0c25 Improve flaps of wings
dfb705b Make some wheels maintenance # Here we forgot a file
a0a3f28 Work on cockpit instruments

$ git add wheel_test.cpp
$ git commit -m "This commit message does not matter"
[airplane 364ff12] This commit message does not matter
Date: Mon Nov 28 11:57:01 2022 +0100
1 files changed, 546 insertions(+), 810 deletions(-)

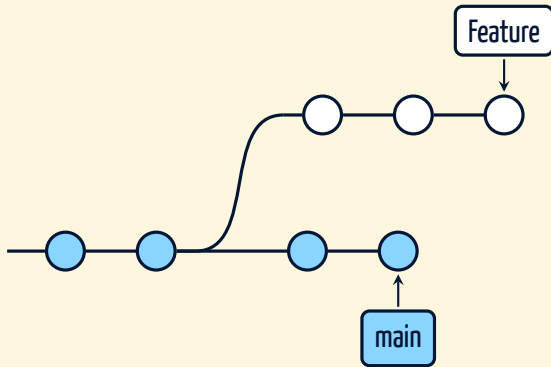
$ git rebase -i HEAD~5 # --> Edit, save, exit from editor
# When asked for, rephrase commit as wished
Successfully rebased and updated refs/heads/airplane.

$ git log --oneline -n 4
51a4b9b Deploy engine turbo and new pipes
afc765a Improve flaps of wings
9927a77 Make some wheels maintenance
a0a3f28 Work on cockpit instruments
```

History changed!

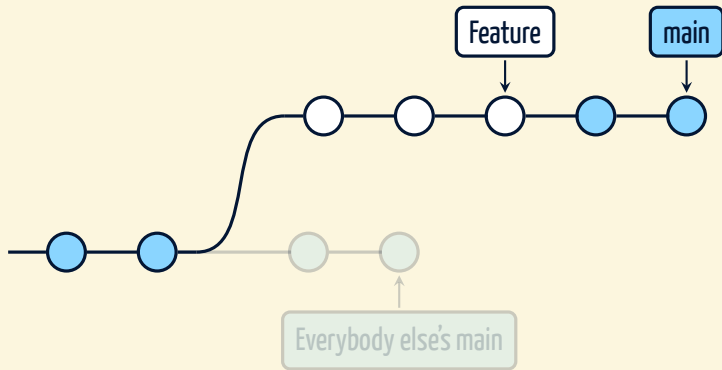
# Coming back to changing public history

The golden rule of rebasing: **Never use it on public branches**



# Coming back to changing public history

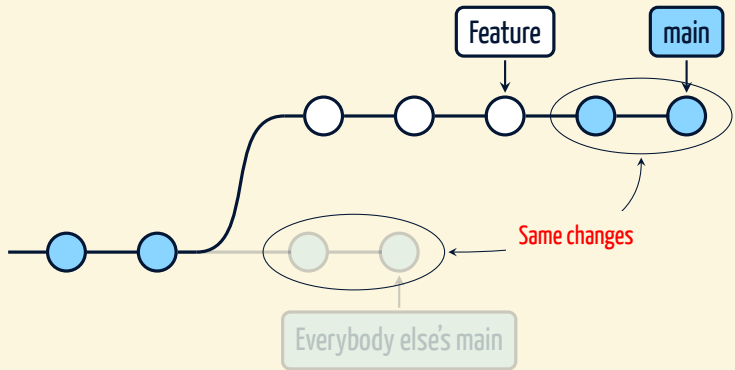
The golden rule of rebasing: **Never use it on public branches**



```
git rebase feature main # ...rebase on main, AAAAAAAAARGH!
```

# Coming back to changing public history

The golden rule of rebasing: **Never use it on public branches**



```
git rebase feature main # ...rebasing on main, AAAAAAAAARGH!
```

Git tag

# Tagging the history of a repository

## Lightweight tags

They simply are a name for an object (often a commit) and they are usually meant for private or temporary use

```
$ git tag v1.0
$ git tag
v1.0
$ git show v1.0
commit aa879f463acd41fc38c7e96090cc1eea279304df
Author: Alessandro Sciarra <sciarra@itp.uni-frankfurt.de>
Date:   Wed Aug 16 14:01:08 2025 +0200

Coffee machine production ready

# Commit differences
```



# Tagging the history of a repository

## Lightweight tags

They simply are a name for an object (often a commit) and they are usually meant for private or temporary use

## Annotated tags

Tag objects contain a creation date, the tagger name and e-mail, a tagging message, and an optional GnuPG signature

# Tagging the history of a repository

```
$ git tag -a v1.0 -m\  
> "Coffee machine software ready for production"  
$ git show v1.0  
tag v1.0  
Tagger: Alessandro Sciarra <sciarra@itp.uni-frankfurt.de>  
Date:   Wed Aug 17 09:12:24 2025 +0200  
  
Coffee machine software ready for production  
  
commit aa879f463acd41fc38c7e96090cc1eea279304df  
Author: Alessandro Sciarra <sciarra@itp.uni-frankfurt.de>  
Date:   Wed Aug 16 14:01:08 2025 +0200  
  
Coffee machine production ready  
  
# Commit differences
```

Prefer annotated tags for releases!

# Tagging the history of a repository

## Lightweight tags

They simply are a name for an object (often a commit) and they are usually meant for private or temporary use

## Annotated tags

Tag objects contain a creation date, the tagger name and e-mail, a tagging message, and an optional GnuPG signature

Tags as commits are by default local

Use `git push --tags` to push them

# Tagging a code: Let's speak about the **same** code!

## A milestone in development

A tag in the git history is to some extent a commitment, but it is also a statement of invaluable help about the codebase!

## Why should I?

- The codebase is released
- The codebase will be released
- The codebase is private but shared with colleagues
- The codebase will be (maybe) inherited
- **The software is used to produce data**

# Semantic versioning

# How should tags be named?

# How should tags be named?

[Prefix]X[.Y[.Z]]

According to the  Semantic Versioning, increase

**MAJOR** version **X** when you make incompatible API changes

**MINOR** version **Y** when you add functionality in a backwards-compatible manner

**PATCH** version **Z** when you make backwards-compatible bug fixes

# How should tags be named?

[Prefix]X[.Y[.Z]]

According to the  Semantic Versioning, increase

**MAJOR** version **X** when you make incompatible API changes

**MINOR** version **Y** when you add functionality in a backwards-compatible manner

**PATCH** version **Z** when you make backwards-compatible bug fixes

Choose your alternative, but give yourself a rule, e.g. increase

**MAJOR** version **X** when you introduce new big features

**MINOR** version **Y** when you add minor functionality or do big refactoring

**PATCH** version **Z** when you make bug fixes (without large changes)



# How should tags be named?

[Prefix]X[.Y[.Z]]

According to the  Semantic Versioning, increase

**MAJOR** version **X** when you make incompatible API changes

**MINOR** version **Y** when you add functionality in a backwards-compatible manner

**PATCH** version **Z** when you make backwards-compatible bug fixes

Choose your alternative, but give yourself a rule, e.g. increase

**MAJOR** version **X** when you introduce new big features

**MINOR** version **Y** when you add minor functionality or do big refactoring

**PATCH** version **Z** when you make bug fixes (without large changes)

**MAJOR** version **X** when you introduce new features

**MINOR** version **Y** when you do big refactoring or fix bugs

# How should tags be named?

[Prefix]X[.Y[.Z]]

According to the  Semantic Versioning, increase

**MAJOR** version **X** when you make incompatible API changes

**MINOR** version **Y** when you add functionality in a backwards-compatible manner

**PATCH** version **Z** when you make backwards-compatible bug fixes

Choose your alternative, but give yourself a rule, e.g. increase

**MAJOR** version **X** when you introduce new big features

**MINOR** version **Y** when you add minor functionality or do big refactoring

**PATCH** version **Z** when you make bug fixes (without large changes)

**MAJOR** version **X** when you introduce new features

**MINOR** version **Y** when you do big refactoring or fix bugs

However you do it, use a CHANGELOG file to list user-relevant changes!

# Ideally

```
$ git tag -n
v2.0.0    Coffee machine ready for milk drinks    Major
v1.3.1    Fix milk temperature problem           Patch
v1.3.0    Interface milk system with machine     Minor
v1.2.0    Add foam generator                     Minor
v1.1.0    Add milk container                     Minor
v1.0.0    Coffee machine production ready       Major
v0.7.1    Fix pipe failures                     Patch
v0.7.0    Interface all components              Minor
v0.6.0    Add coffee grounds container          Minor
v0.5.0    Add water tray                       Minor
v0.4.0    Add water tank                       Minor
v0.3.0    Add bean container                   Minor
v0.2.0    Add brew system and core engine       Minor
v0.1.0    Deploy coffee machine skeleton        Minor
```

- Clear evolution of the codebase for everybody
- **v1.0** usually refers to the first **production-ready** version

# Summary so far



# Git-flow

# 5

## Git-flow

- Motivation and background
- The main branches
- Feature branches
- Release branches
- Hotfix branches
- Git-flow: an additional tool

# Disclaimer and references


## Vincent Driessen back in 2010

In the following I will give you a summary of his blog page.  
All credit goes to the author and no original part is here contained.

## Vincent Driessen as nvie

The original author also implemented a collection of git extensions which allow to easily use his branching model (active till 2012).

## Alternative implementations

AVH Edition:  [gitflow-avh](#) (archived in June 2023)

CJS Edition:  [gitflow-cjs](#) (still active, but rarely)

# Motivation

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

 <https://www.xkcd.com/1296/>



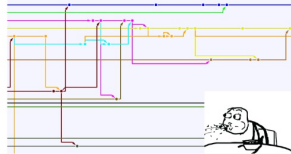
# Motivation

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAA	3 HOURS AGO
○	ADKFTSLKDFJSOKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

<https://www.xkcd.com/1296/>

Sometimes history becomes messy



[https://image.slidesharecdn.com/\[...\]](https://image.slidesharecdn.com/[...])

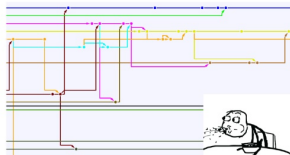
# Motivation

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAA	3 HOURS AGO
○	ADKFJSLKDFJSOKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

<https://www.xkcd.com/1296/>

Sometimes history becomes messy

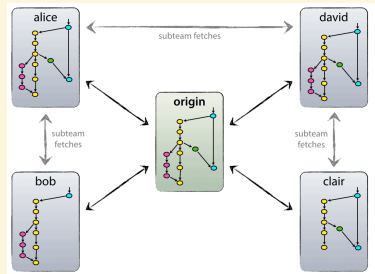


[https://image.slidesharecdn.com/\[...\]](https://image.slidesharecdn.com/[...])

- **Ordered** and **standardized** way to daily work
- Easy way to keep big projects history and development tidied up
- Sustainable work in team also **for new members**
- In larger projects, easy interaction within and between sub-teams
- A way to go for released software

# Background

- Central repository (e.g. GitHub)
- Developers pull/push from/to origin
- Sub-team fetches lead to high work efficiency
- It might be useful to work together with two or more developers on a big new feature, before pushing the work in progress to origin prematurely.
- **Never forget origin is public!**



Technically, this means nothing more than that, for example, Alice has defined a Git remote, named `bob`, pointing to Bob's repository, and vice versa.

# The main and develop branches

`origin/main`

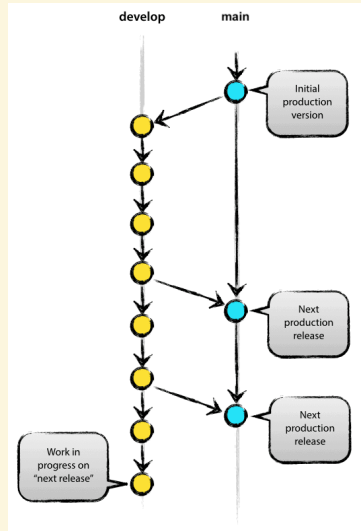
The source code of HEAD always reflects a production-ready state

`origin/develop`

The source code of HEAD always reflects a state with the latest delivered development changes for the next release

Each time when changes are merged back into main, this is a new production release by definition

→ hooks for enforced policies!

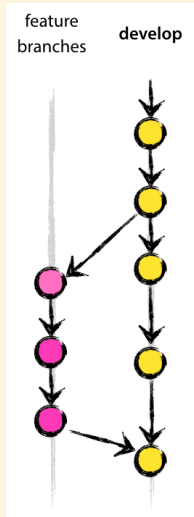


# The feature branches

## Properties

- May branch off from:  
`develop`
- Must merge back into:  
`develop`
- Any name except:  
`main`, `develop`  
`release-*`, `hotfix-*`

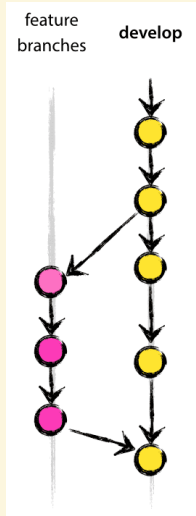
Feature branches typically exist in developer repository only, not in origin.



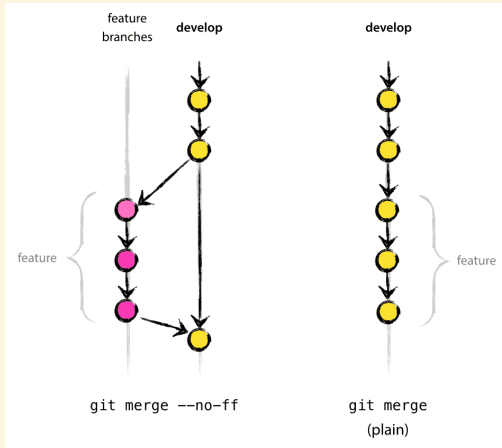
# The feature branches

## How it looks like:

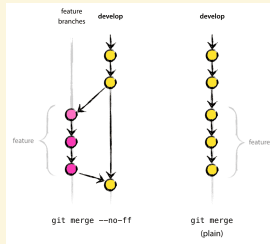
```
$ git switch -c myfeature develop
Switched to a new branch "myfeature"
#-----
# Some development, commits, etc.
#-----
$ git switch develop
Switched to branch 'develop'
$ git pull origin develop
Already up-to-date.
$ git merge --no-ff myfeature
Updating ea1b82a..05e9557
(Summary of changes)
$ git branch -d myfeature
Deleted branch myfeature (was 05e9557).
$ git push origin develop
[...]
```



# Why to avoid a fast-forward merge?



# Why to avoid a fast-forward merge?

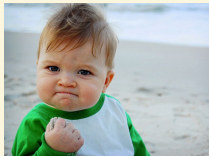


- ✓ No information lost about the historical existence of a feature
- ✓ Easy to see in history which commits have implemented a feature
- ✓ Easy to revert a whole feature (i.e. a group of commits)
- ✗ It will create (empty) commit objects
  - ↳ the gain is much bigger than the cost!



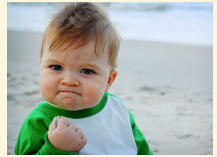
# The release branches

*Make you feel like*



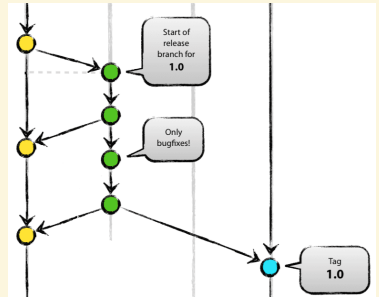
# The release branches

*Make you feel like*



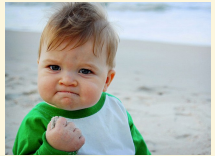
## Properties:

- May branch off from:  
**develop**
- Must merge back into:  
**develop, main**
- Branch naming convention:  
**release-\***



# The release branches

*Make you feel like*



- Release branches support preparation of a new production release
- Last-minute checks, minor bug fixes and preparing meta-data (version number, etc.) should be done on a release branch
- By doing all of this work on a release branch, the **develop** branch is cleared to receive features for the following release
- When a new release branch is created, all features that are targeted for this release must be merged into the **develop** branch

# The release branch: How it works

## Creation

```
Version 1.1.5 is the current production release
We are ready for "next release" -> NOW we decide v1.2
$ git switch -c release-1.2 develop
Switched to a new branch "release-1.2"
$ ./bump-version.sh 1.2 #Just modify some files
Files modified successfully, version bumped to 1.2.
$ git commit -a -m "Bump version number to 1.2"
[release-1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)
```

**Adding new features here is strictly prohibited!**  
The idea is to just make sure everything works and is  
ready to be published!

# The release branch: How it works

## Creation

```
$ git switch -c release-1.2 develop
$ ./bump-version.sh 1.2 #Just modify some files
$ git commit -a -m "Bump version number to 1.2"
```

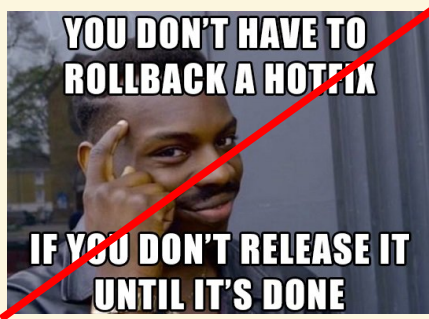
## Finalisation

```
$ git switch main
Switched to branch 'main'
$ git merge --no-ff release-1.2
Merge made by recursive. (Summary of changes)
$ git tag -a v1.2
$ git switch develop
Switched to branch 'develop'
$ git merge --no-ff release-1.2 #Maybe conflicts!
Merge made by recursive. (Summary of changes)
$ git branch -d release-1.2
Deleted branch release-1.2 (was ff452fe).
```

# The hotfix branch

# The hotfix branch

A hotfix is not something bad!



...and, after all, never forget Murphy's law!

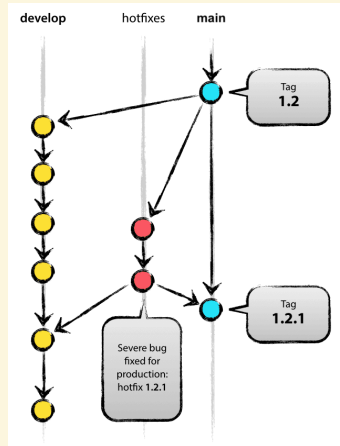
# The hotfix branch

## Properties:

- May branch off from:  
**main**
- Must merge back into:  
**develop, main**
- Branch naming convention:  
**hotfix-\***

## Main advantage

The essence is that work on the develop branch can continue, while a quick production fix is prepared





# The hotfix branch: How it works

## Creation

```
$ git switch -c hotfix-1.2.1 main
Switched to a new branch "hotfix-1.2.1"
$ ./bump-version.sh 1.2.1 #Just modify some files
Files modified successfully, version bumped to 1.2.1.
$ git commit -a -m "Bump version number to 1.2.1"
[hotfix-1.2.1 41e61bb] Bump version number to 1.2.1
1 files changed, 1 insertions(+), 1 deletions(-)
# Some work to fix the situation
$ git commit -m "Fix severe production problem"
[hotfix-1.2.1 abbe5d6] Fix severe production problem
5 files changed, 32 insertions(+), 17 deletions(-)
```

# The hotfix branch: How it works

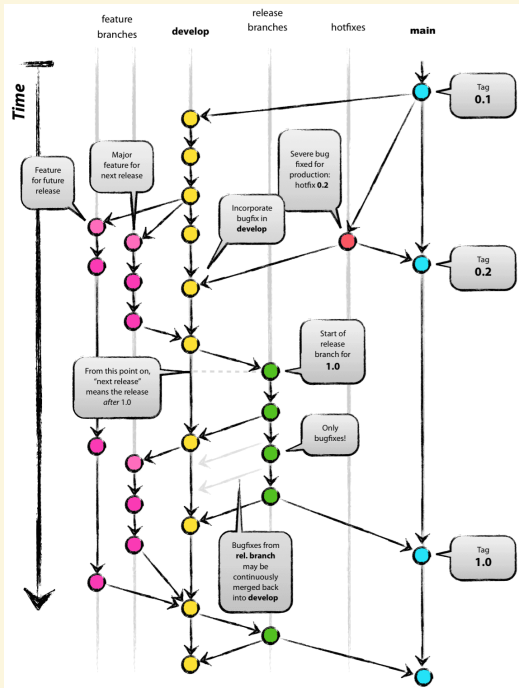
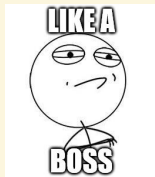
## Creation

```
$ git switch -c hotfix-1.2.1 main
$ ./bump-version.sh 1.2.1 #Just modify some files
$ git commit -a -m "Bump version number to 1.2.1"
# Some work to fix the situation
$ git commit -m "Fix severe production problem"
```

## Finalisation

```
$ git switch main
Switched to branch 'main'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive. (Summary of changes)
$ git tag -a v1.2.1
$ git switch develop
Switched to branch 'develop'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive. (Summary of changes)
$ git branch -d hotfix-1.2.1
Deleted branch hotfix-1.2.1 (was abbe5d6).
```

# Summary





But, wait, everything by hand?!

# Git-flow: How to easily apply what we learnt

Pick a set of high-level repository operations



**nvie**

or better



**gitflow-avh**



**gitflow-cjs**

# Git-flow: How to easily apply what we learnt

Pick a set of high-level repository operations



or better



gitflow-avh



gitflow-cjs

## The base syntax

→ Standard raw git commands can be used as usual, too

```
# To initialize a new repository, use:
git flow init [-d]
# To list/start/finish auxiliary branches, use:
git flow <type>
git flow <type> start <name> [<base>]
git flow <type> finish <name>
# To push/pull an auxiliary branch to the remote, use:
git flow <type> publish <name>
git flow <type> pull <remote> <name>
```

<type> can be either feature, release or hotfix

<name> is the name of the auxiliary branch

<base> must be a commit on develop or main

# Git-flow: How to easily apply what we learnt

Pick a set of high-level repository operations



or better



gitflow-avh



gitflow-cjs

Take-home message

If you have a released software or plan to have one, Git-flow branching model can help you to bring your product to another level!

Think about it

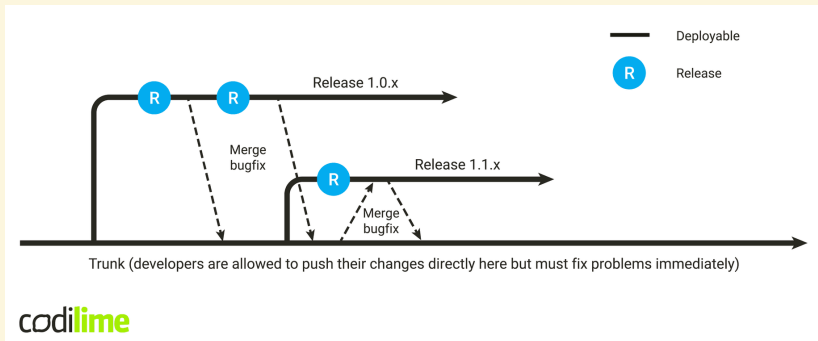
Even if you use a (slightly) different branching model, it might well be that available git extensions still partially fit your use case

I heard Git-flow is obsolete, is it?



# No, it is not!

## Trunk-based development



In the last years this alternative approach has become popular

# No, it is not!

## When does trunk-based development work best?

- When you need to iterate quickly
- When you work mostly with senior developers

## When does Git-flow work best?

- When you have a lot of junior developers
- When you have an established product
- When you run an open-source project

## In academia

- Development is rather slow and in small teams

# Summary and outlook

# What did we left out in this git trilogy?

<b>git mv</b>	Move or rename a file, a directory, or a symlink { pretty trivial }
<b>git rm</b>	Remove files from the working tree and from the index { pretty trivial }
<b>git restore</b>	Restore working tree files { we mentioned it already }
<b>git reset</b>	Reset current HEAD to the specified state
<b>git blame</b>	Find out who modified each line in each file
<b>git revert</b>	Create new commits to undo existing ones
<b>git bisect</b>	Use binary search to find the commit that introduced a bug
<b>git grep</b>	Print lines matching a pattern
<b>git [...]</b>	Some more technical commands

# What did we left out in this git trilogy?

<b>git mv</b>	Move or rename a file, a directory, or a symlink { pretty trivial }
<b>git rm</b>	Remove files from the working tree and from the index { pretty trivial }
<b>git restore</b>	Restore working tree files { we mentioned it already }
<b>git reset</b>	Reset current HEAD to the specified state
<b>git blame</b>	Find out who modified each line in each file
<b>git revert</b>	Create new commits to undo existing ones
<b>git bisect</b>	Use binary search to find the commit that introduced a bug
<b>git grep</b>	Print lines matching a pattern
<b>git [...]</b>	Some more technical commands

Go for it!

Read your favourite source, you'll be able to understand it alone!

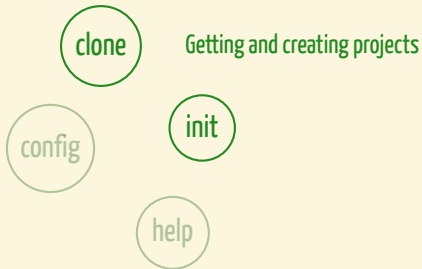
# Most used Git commands

## Setup and Config

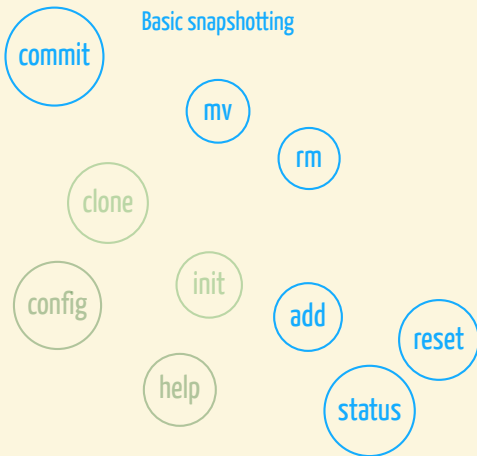
config

help

# Most used Git commands

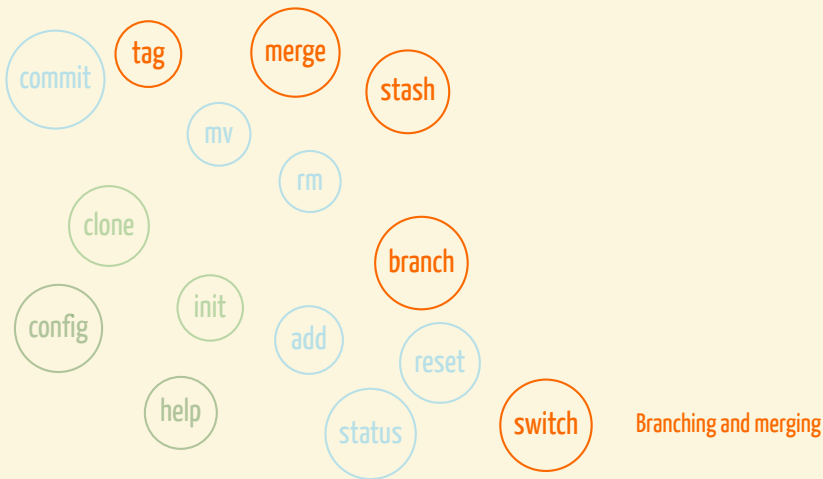


# Most used Git commands

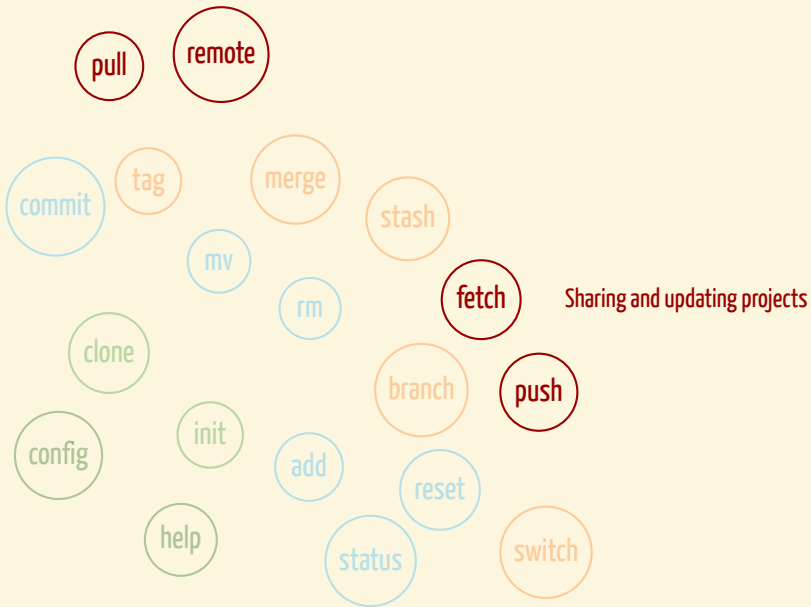




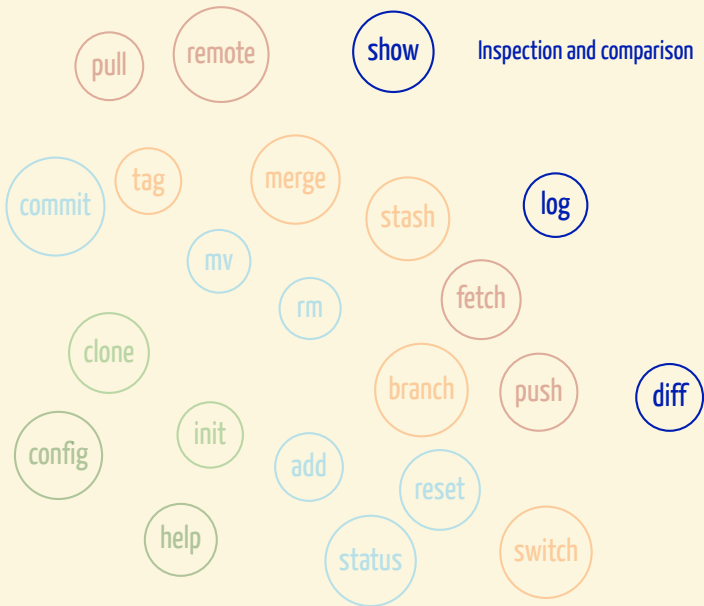
# Most used Git commands



# Most used Git commands

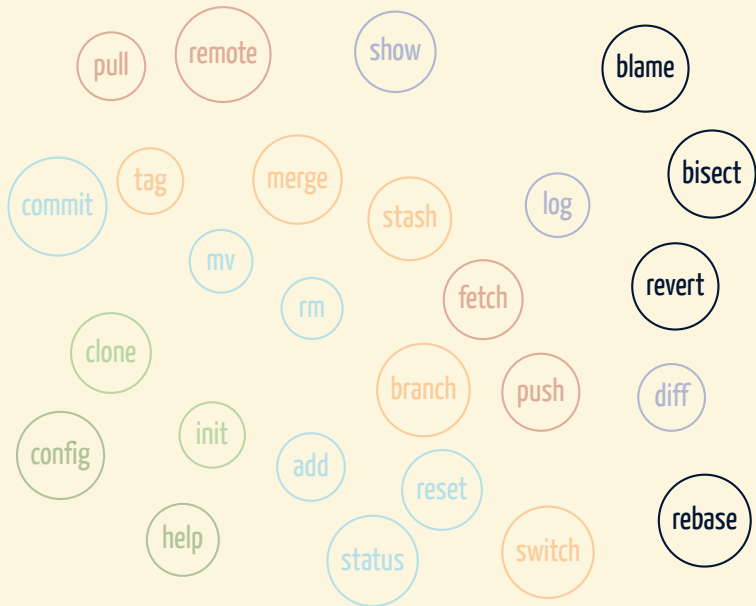


# Most used Git commands

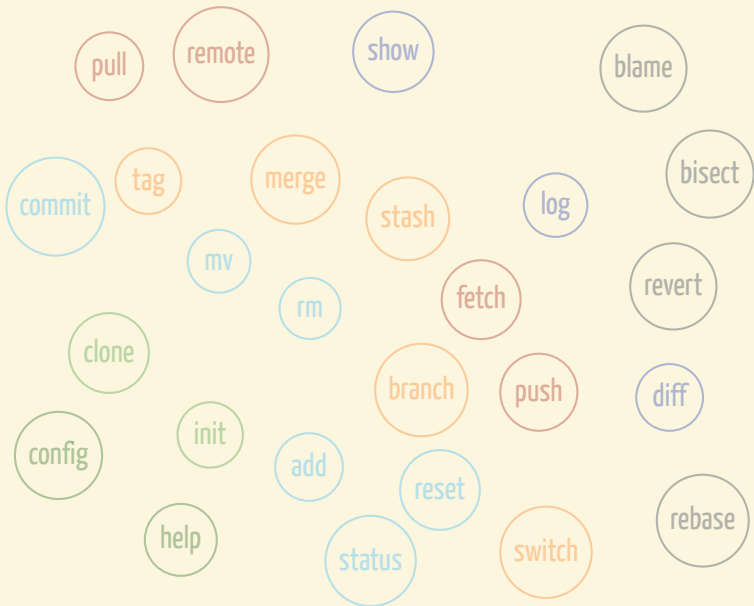


# Most used Git commands

Patching and debugging



# Most used Git commands



# Most used Git commands



# Thanks for listening and remember

In case of fire:

- 1 `git switch -c dfjaskjdsgk`
- 2 `git commit -am 'fvfsdavg'`
- 3 `git push`
- 4 Exit the building



Not sure whether I need an alias for that...

## Questions? Feedback?