Introduction to Git

Sébastien Varrette, PhD

UL HPC Management Team,

Parallel Computing and Optimization Group (PCOG),

University of Luxembourg (UL), Luxembourg

Latest version available on Github:

Beamer theme Falkor:	https://github.com/Falkor/beamerthemeFalkor
Generic Makefiles:	https://github.com/Falkor/Makefiles
LaTeX Sources of the slides (Markdown):	https://github.com/ULHPC/documents
UL HPC School:	https://hpc.uni.lu/hpc-school/
UL HPC Tutorials:	http://ulhpc-tutorials.readthedocs.org/

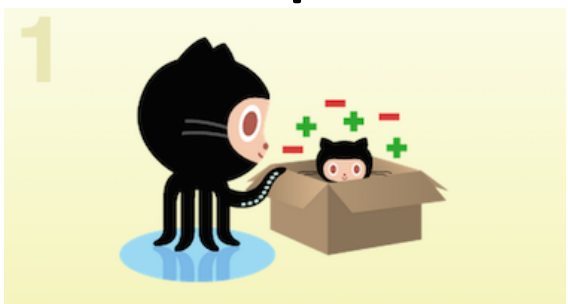
Summary

Github

github.com

(Reference) web-based Git repository hosting service

Set up Git



Create Repository

Fork repository

Work together

git-scm.com --everything-is-local

Git – the simple Guide

http://rogerdudler.github.io/git-guide/

git - the simple guide

just a simple guide for getting started with git. no deep shit ;)

Tweet 4,747

download the cheat sheet now. it's free!

by Roger Dudler

credits to @tfnico, @fhd and Namics

this guide in deutsch, español, français, indonesian, italiano, nederlands, polski, português, русский, türkçe, မြန်မာ, 日本語, 中文, 한국어 Vietnamese

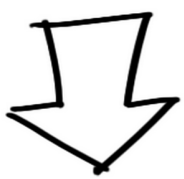
please report issues on github

want a simple but powerful git client for your mac?

Are You a Front-End Developer?
by Roger Dudler, Author of the Git Simple Guide

Try Frontify

Now Free with Github Integration!

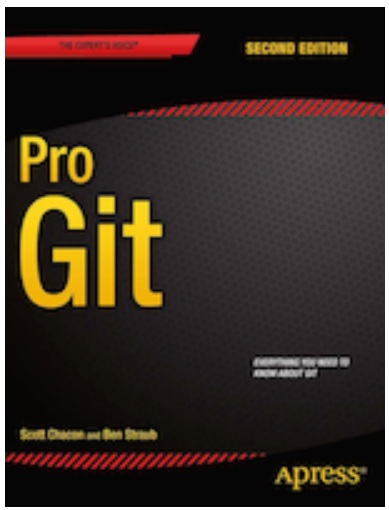


Atlassian Tutorials

https://www.atlassian.com/git/tutorials/

Pro Git Book – progit.org

- Open-Source Book on Git by S. Chacon and B. Straub
  - ↪ Sources (on Github)
  - ↪ Online Reading – PDF
- See also Git Internal, also by S. Chacon



- **Note**: Most images of this talk comes from this book
  - ↪ more precisely the first edition

Summary

Why use Version Control?

- Version Control = Revision Control = Source Control
  - ↪ lets you track your files over time.
- you probably cooked up your own!
  - ↪ ever get files like `main-v2.tex`, `CORE-proposal.doc.old` or `2015-03-cv.pdf`?

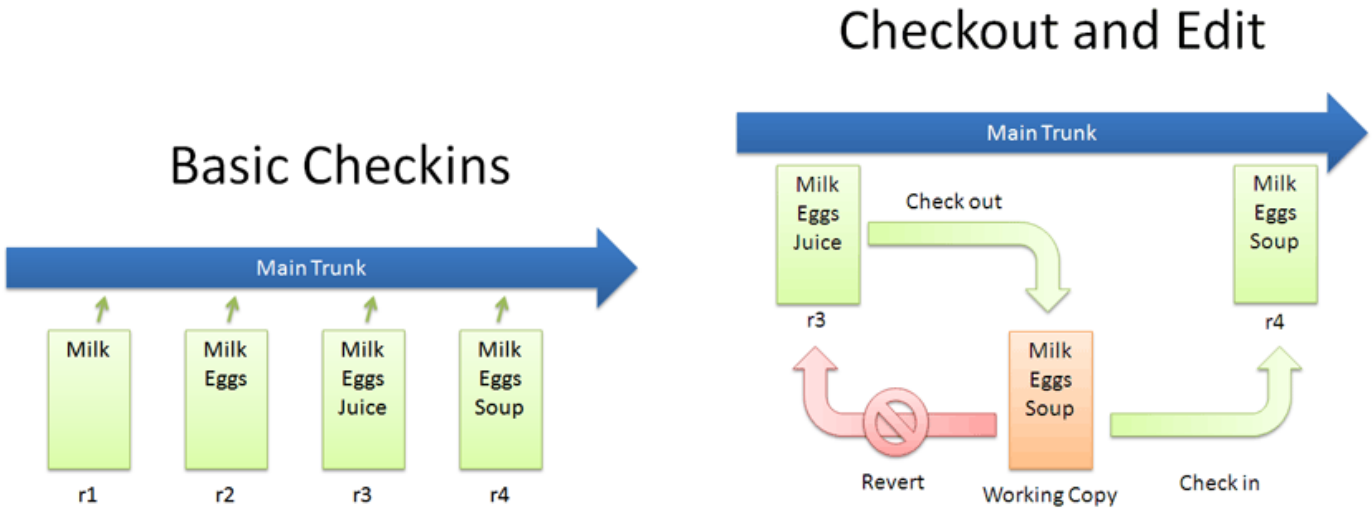
Why use Version Control?

- Version Control = Revision Control = Source Control
  - ↪ lets you track your files over time.
- you probably cooked up your own!
  - ↪ ever get files like `main-v2.tex`, `CORE-proposal.doc.old` or `2015-03-cv.pdf`?

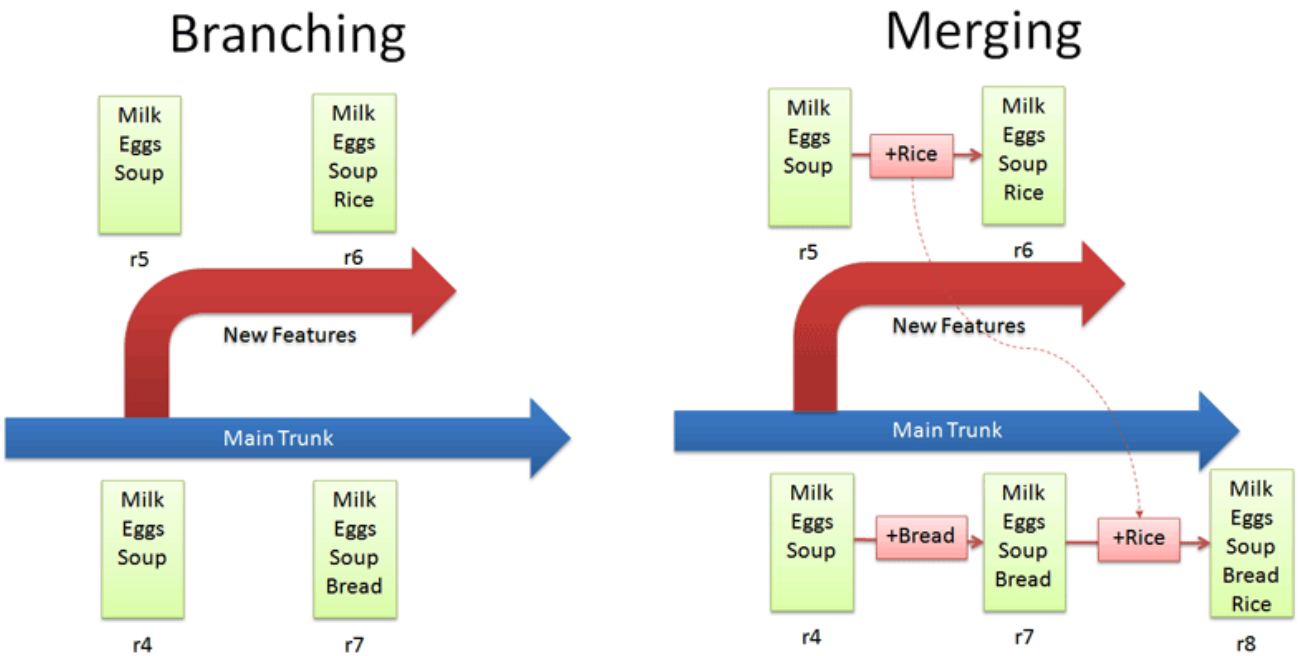
Version Control System (VCS)

- Integrated fool-proof framework for:
  - ↪ Backup and Restore
  - ↪ Synchronization / Collaborating
  - ↪ Short and long-term undo / Tracking changes
  - ↪ Sandboxing

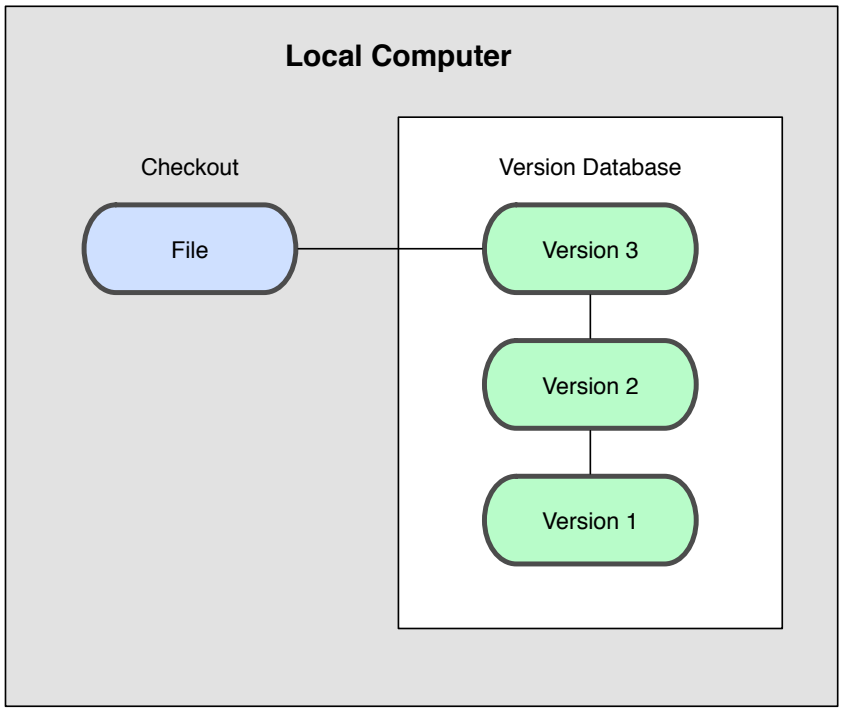
Typical VCS Workflow



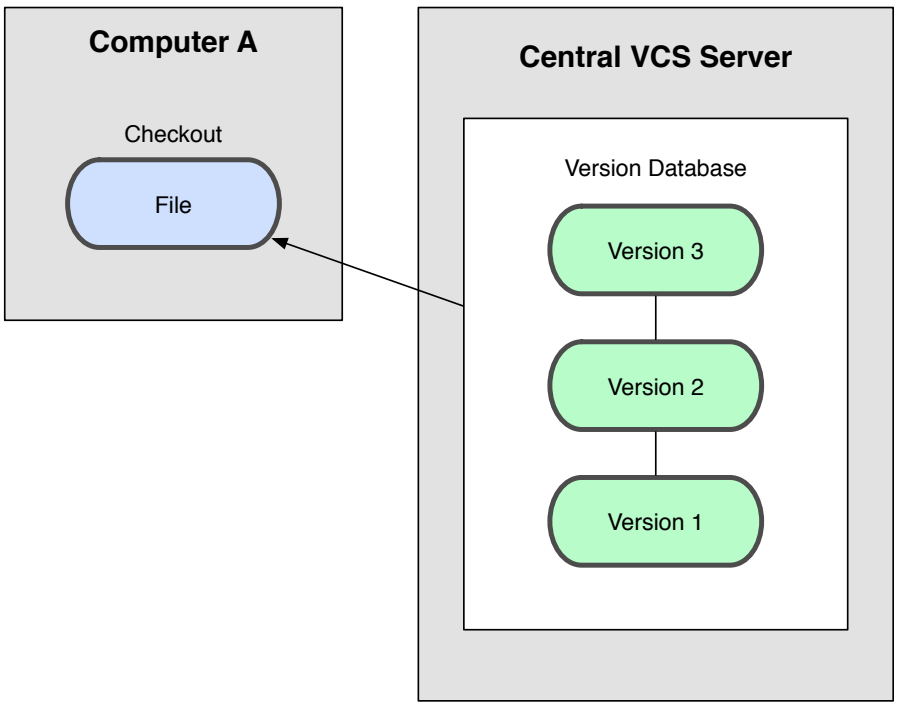
Typical VCS Workflow



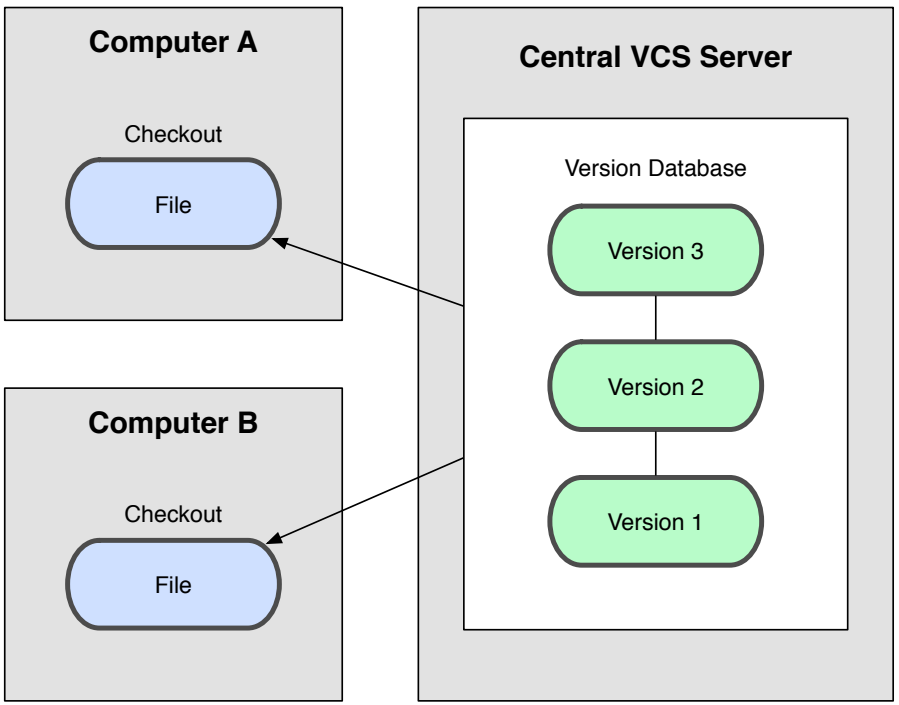
Local VCS – RCS, Mac OS Versions



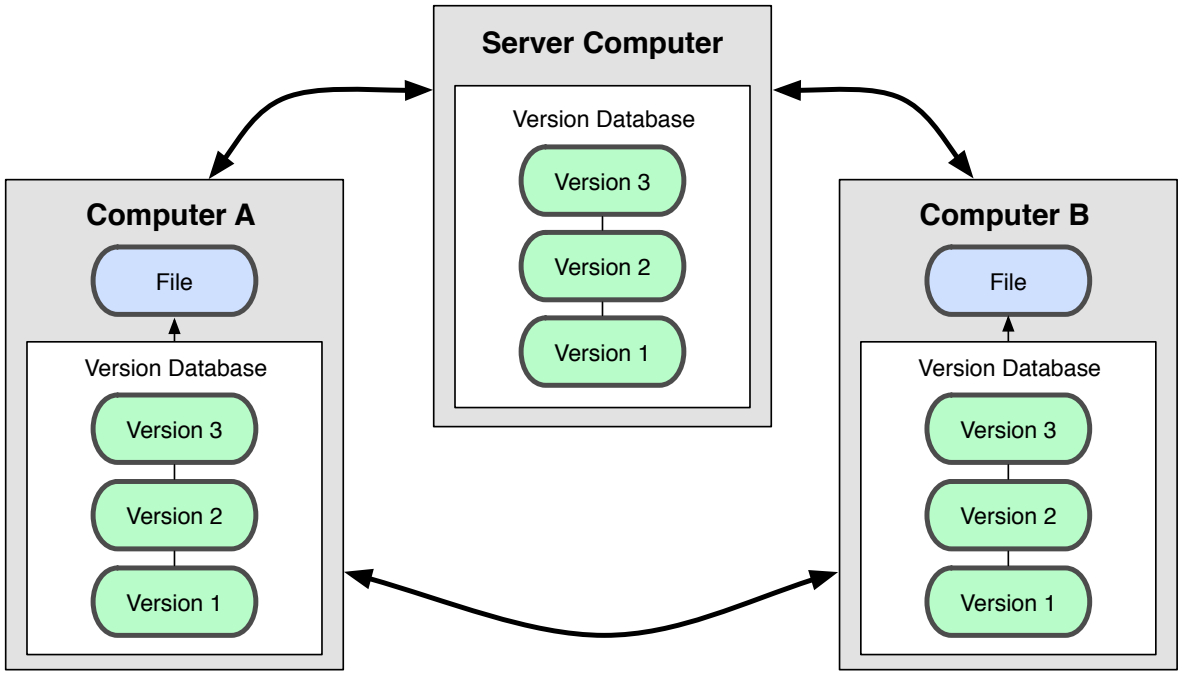
Centralized VCS – CVS, SVN



Centralized VCS – CVS, SVN

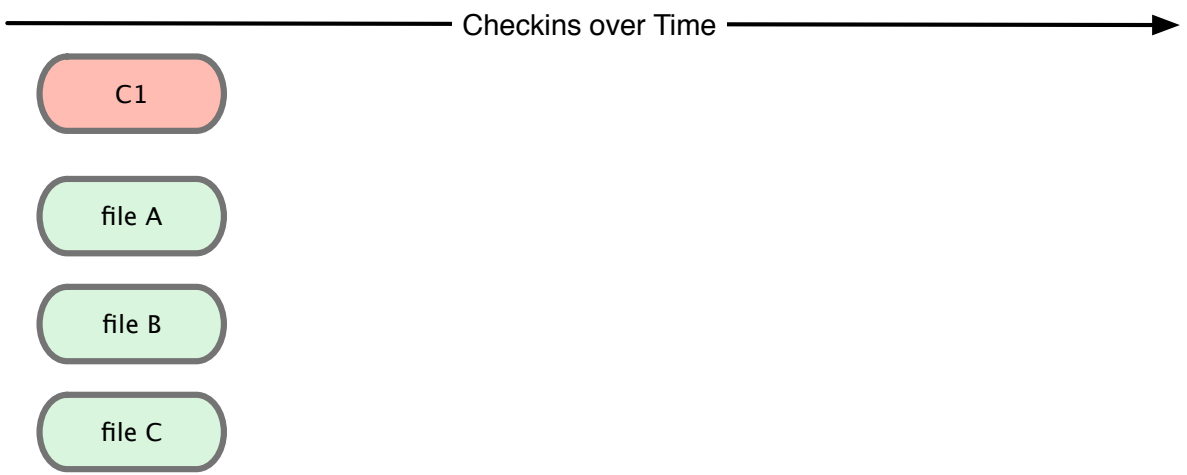


Distributed VCS – Git

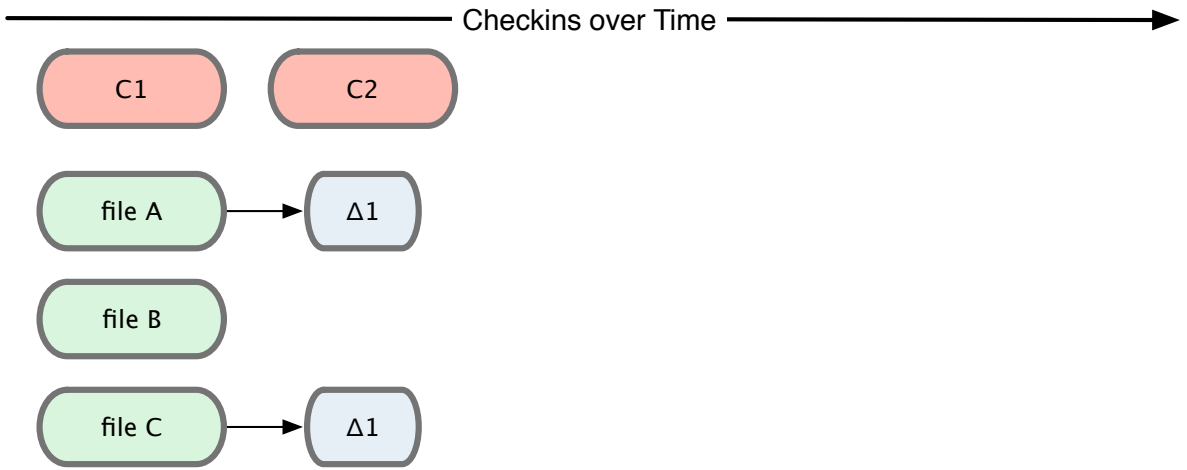


Everybody has the full history of commits

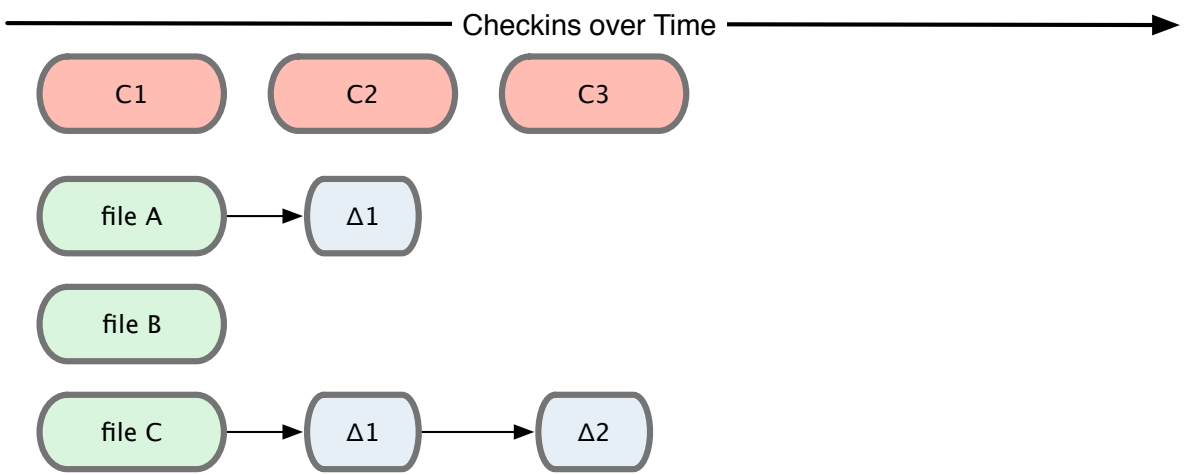
Tracking changes (most VCS)



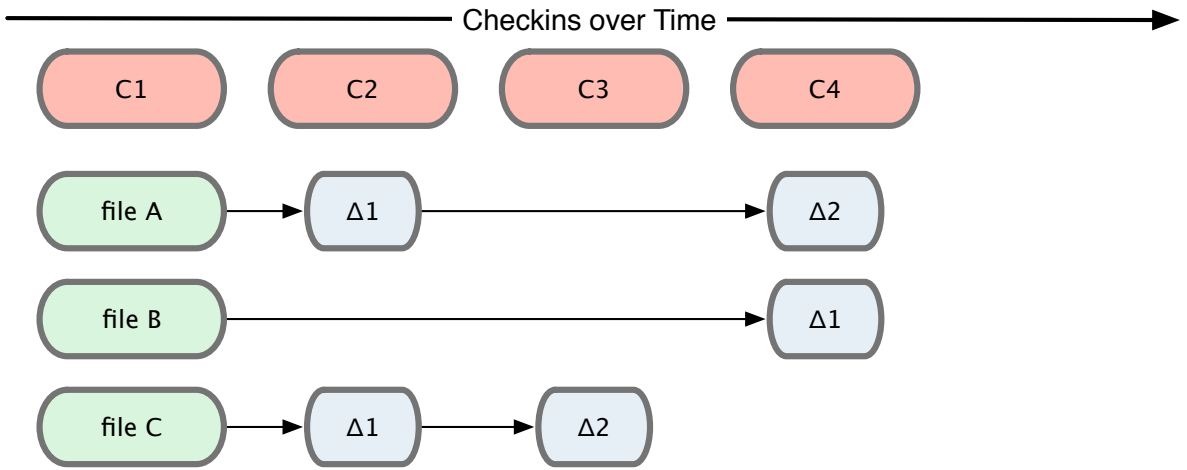
Tracking changes (most VCS)



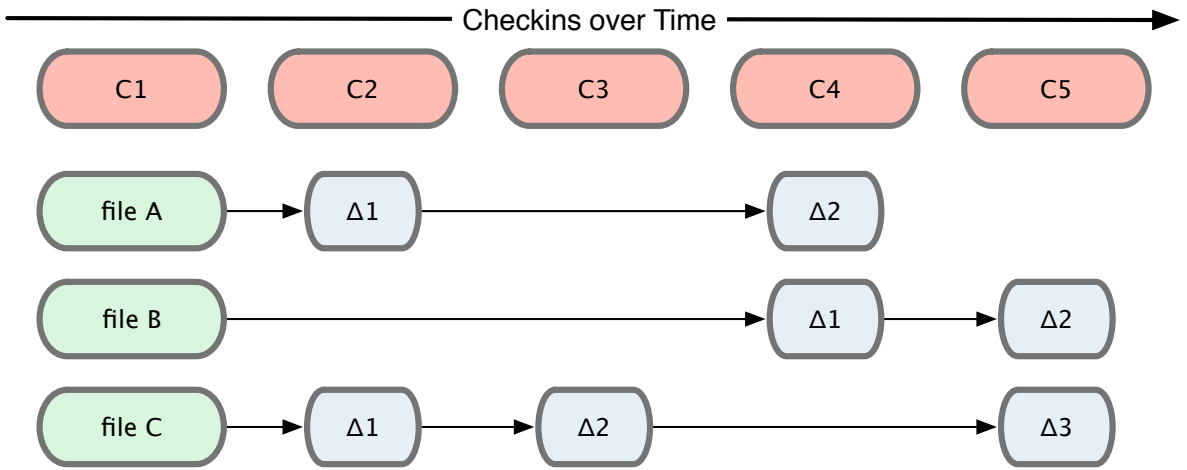
Tracking changes (most VCS)



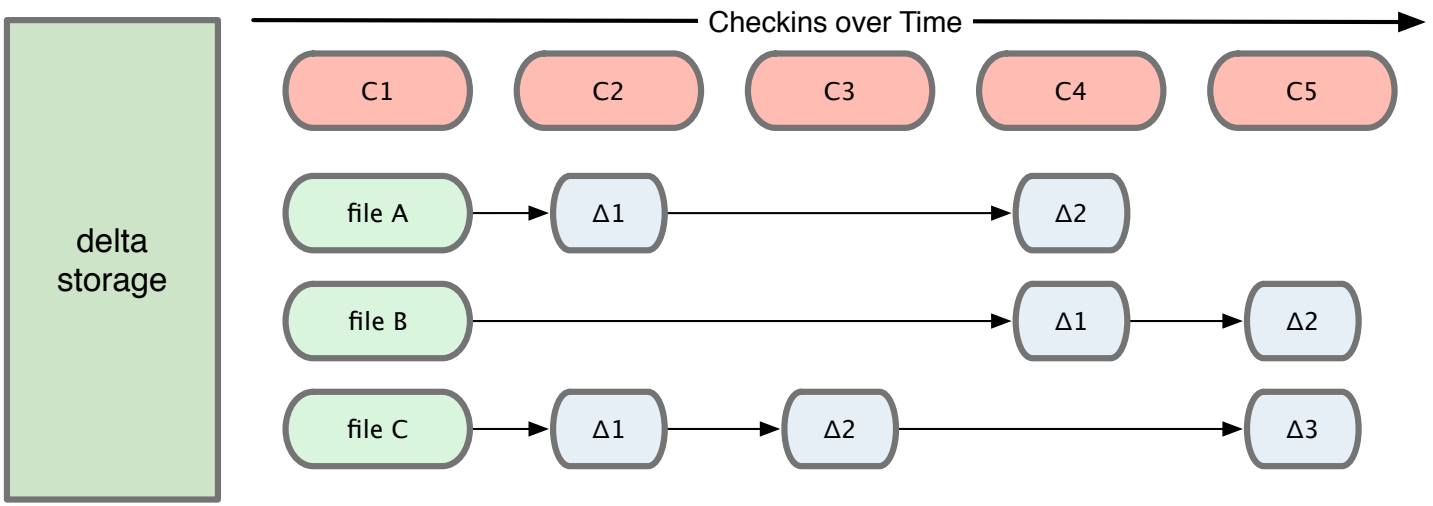
Tracking changes (most VCS)



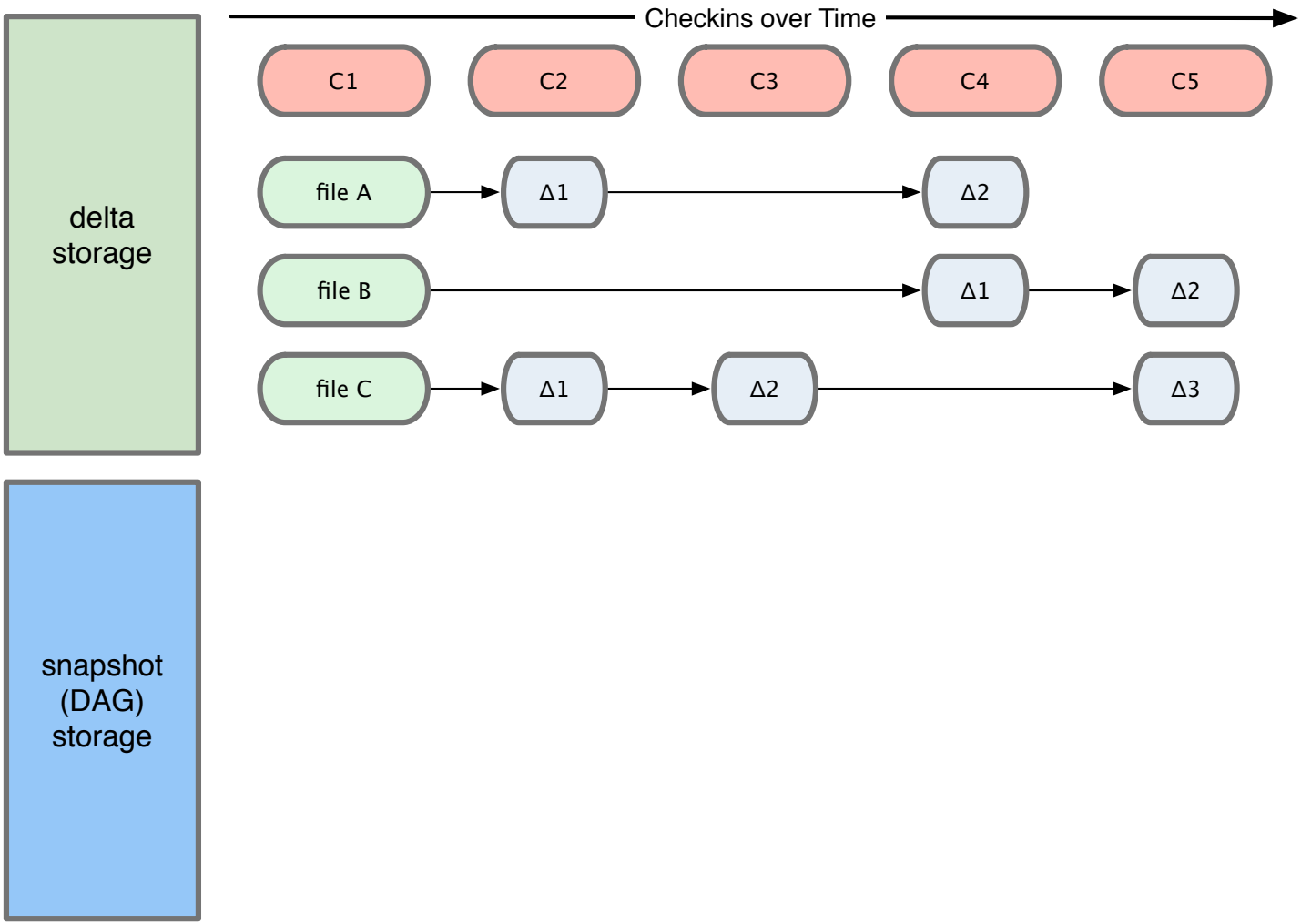
Tracking changes (most VCS)



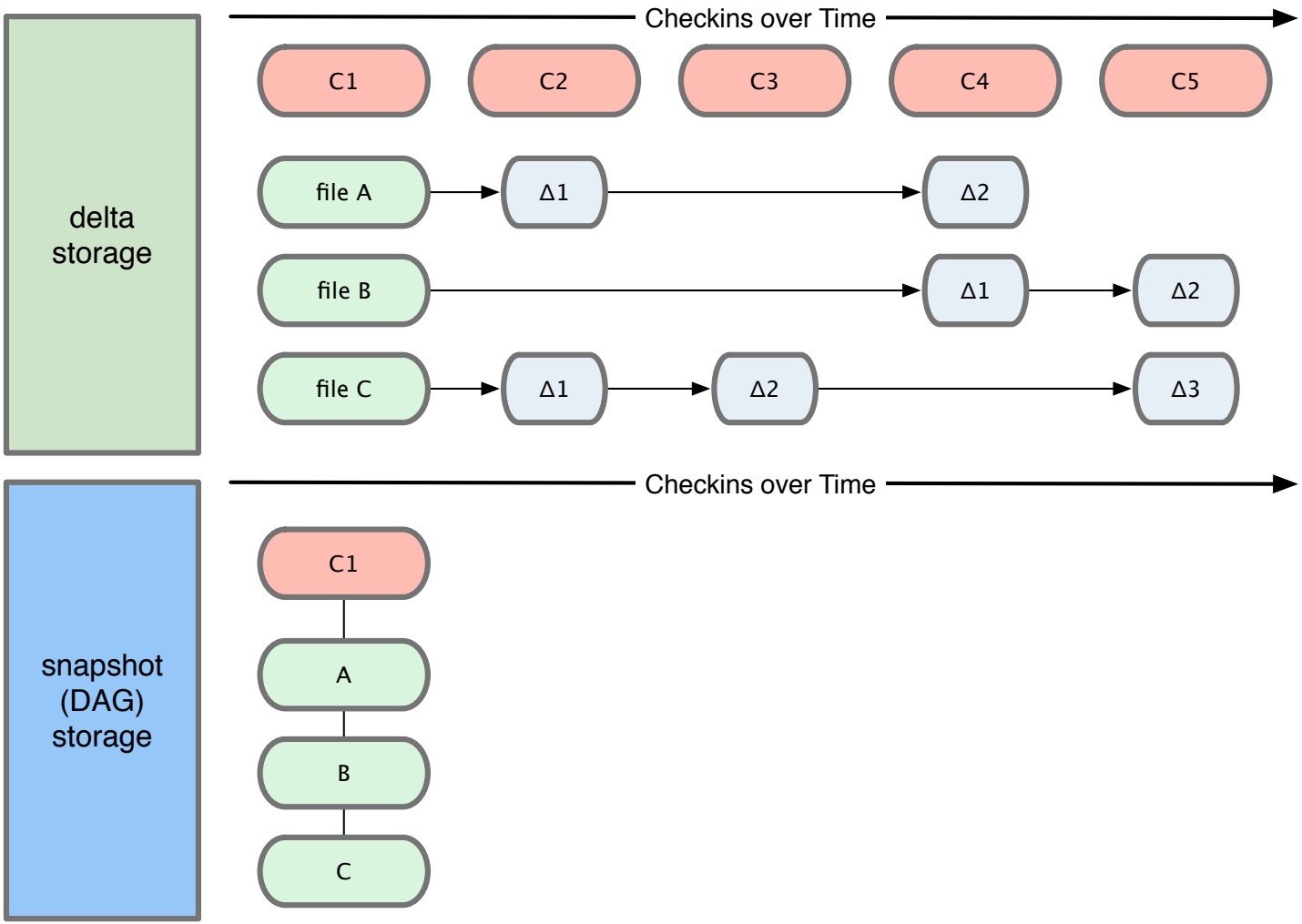
Tracking changes (most VCS)



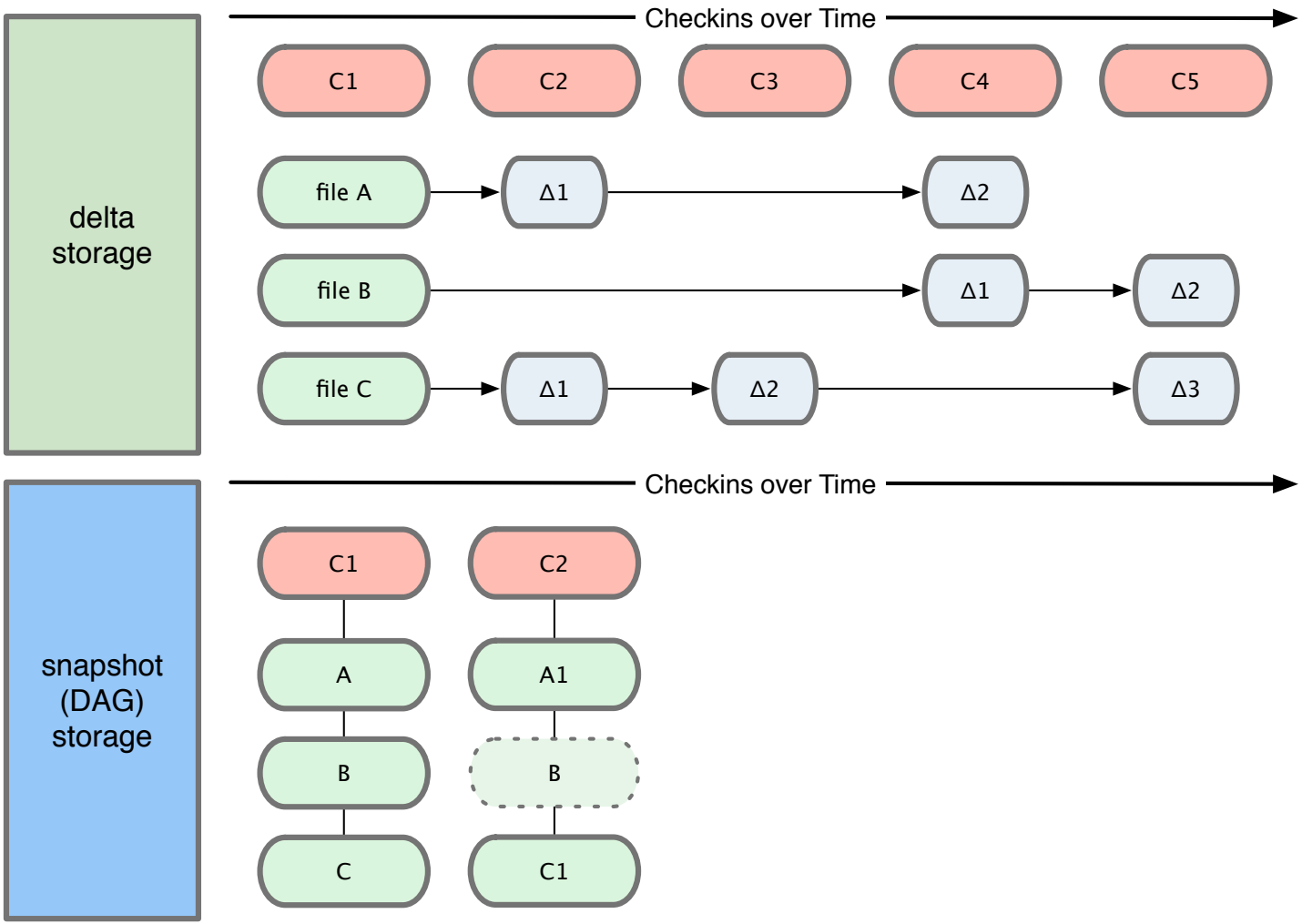
Tracking changes (Git)



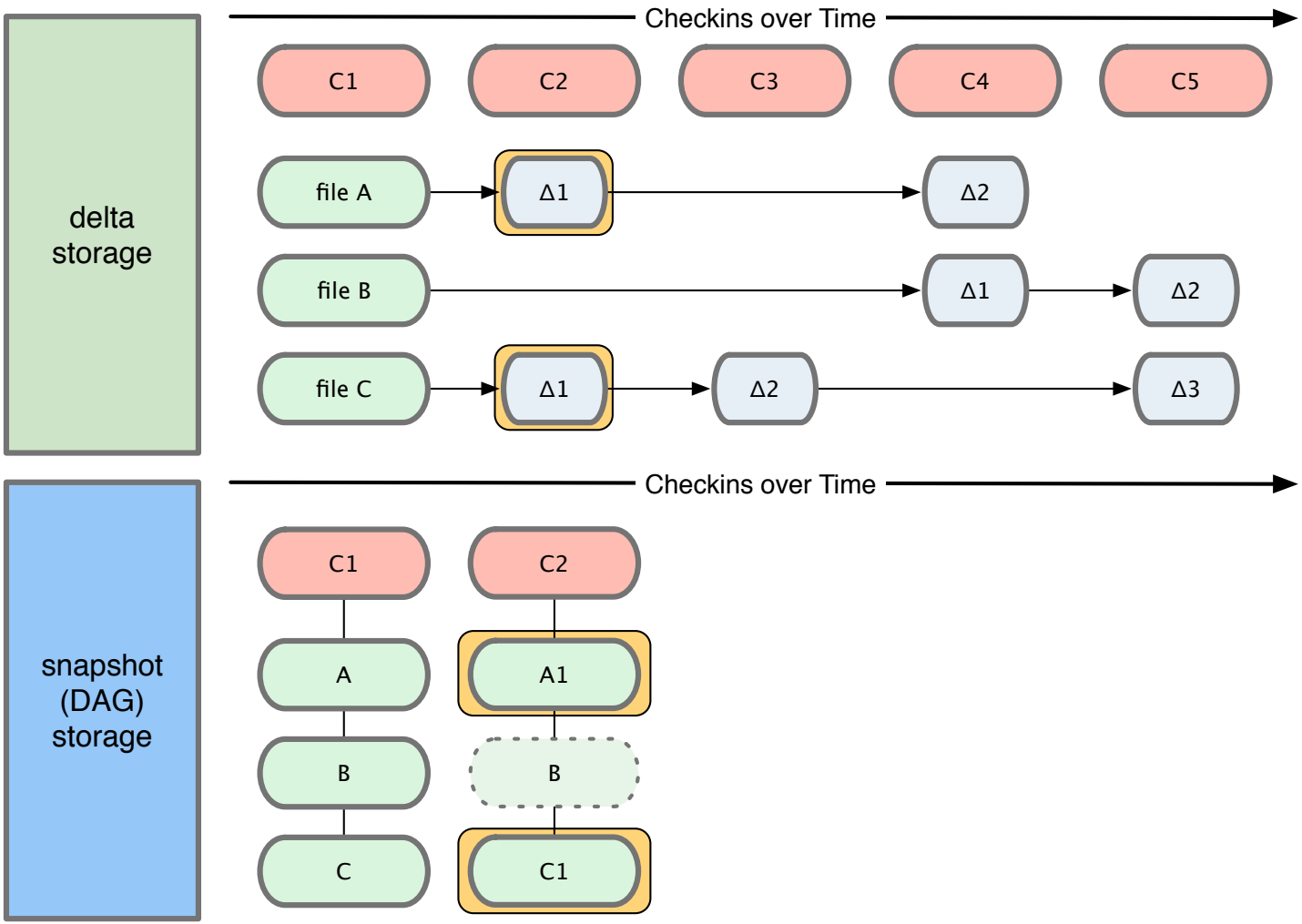
Tracking changes (Git)



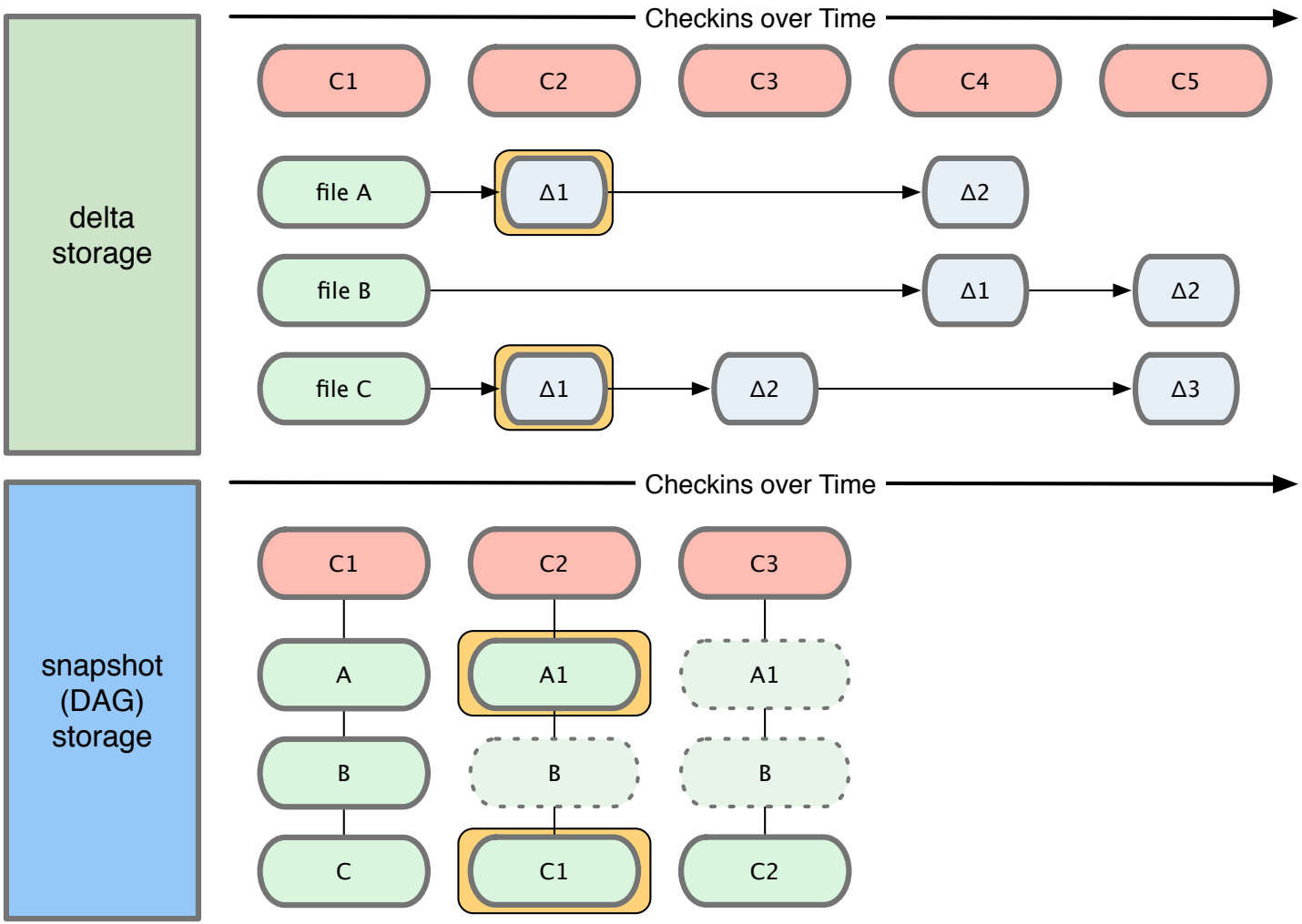
Tracking changes (Git)



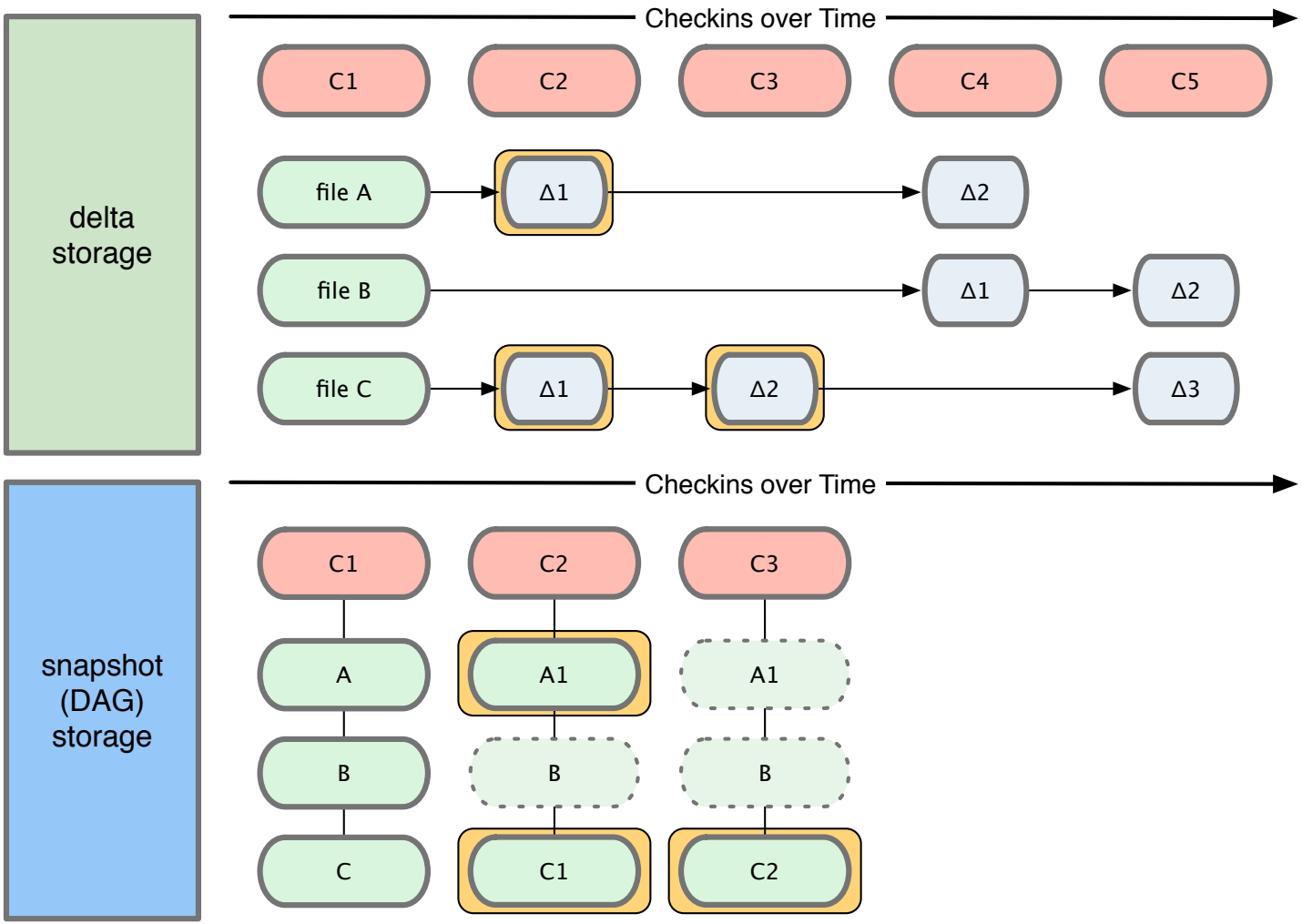
Tracking changes (Git)



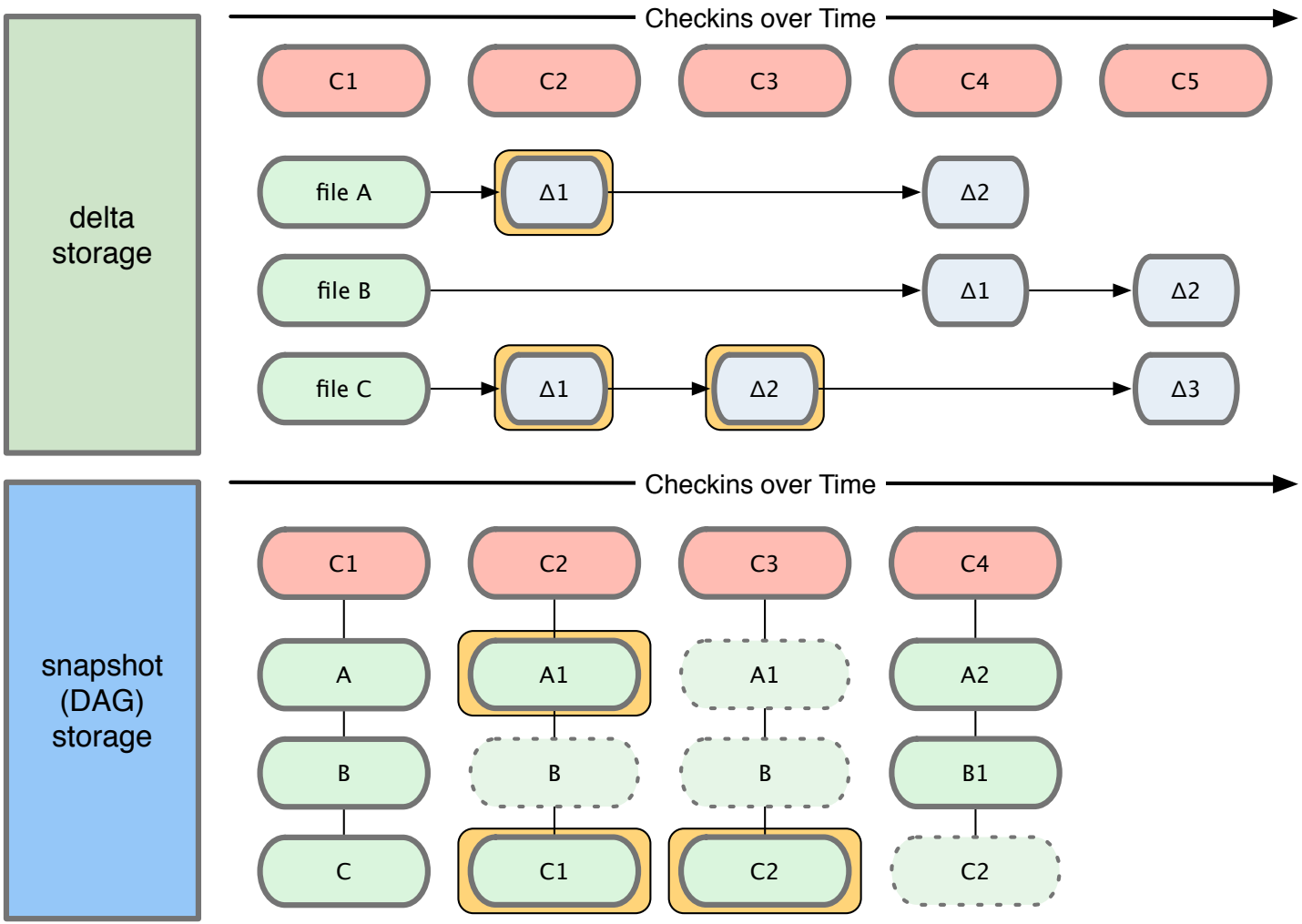
Tracking changes (Git)



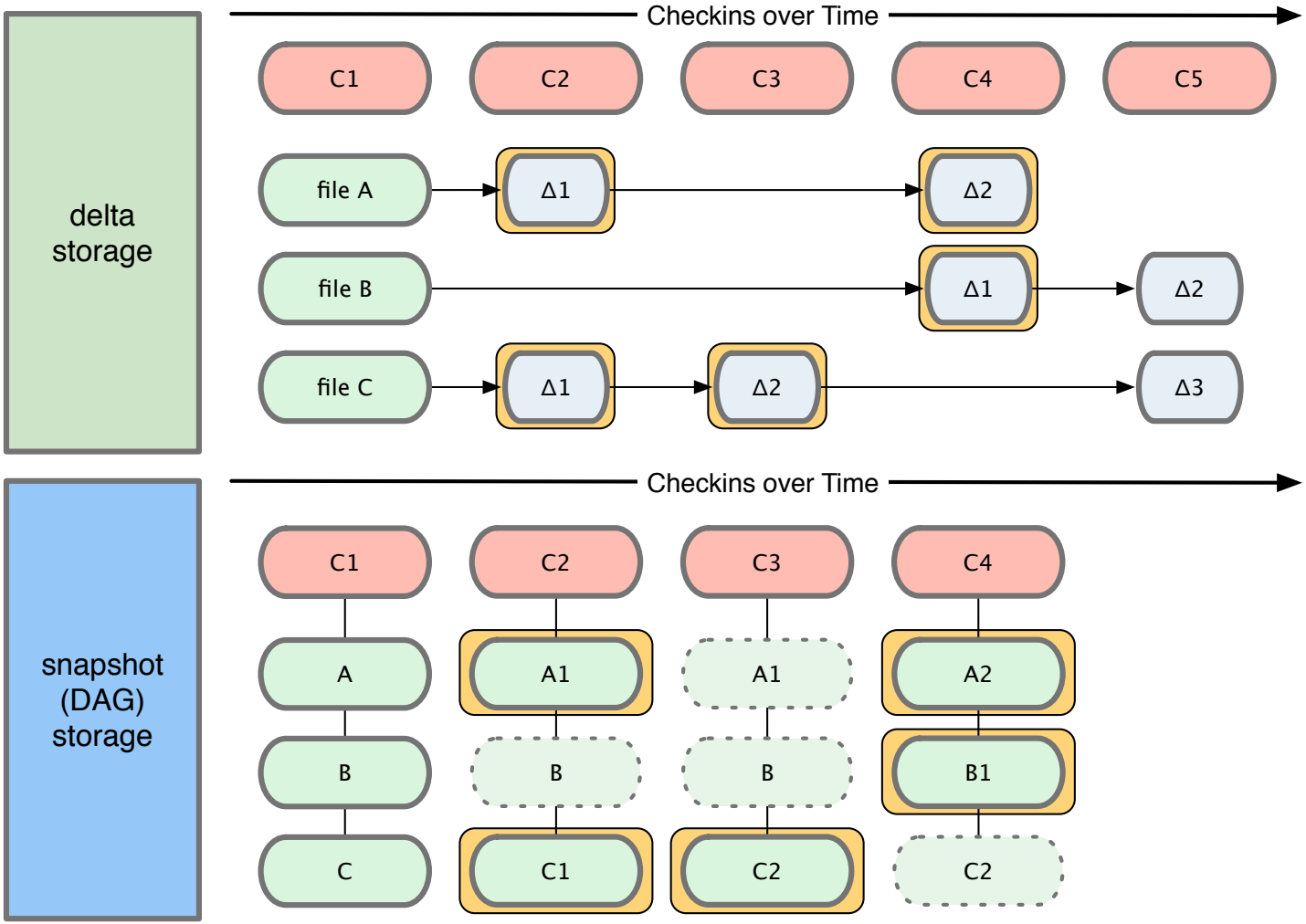
Tracking changes (Git)



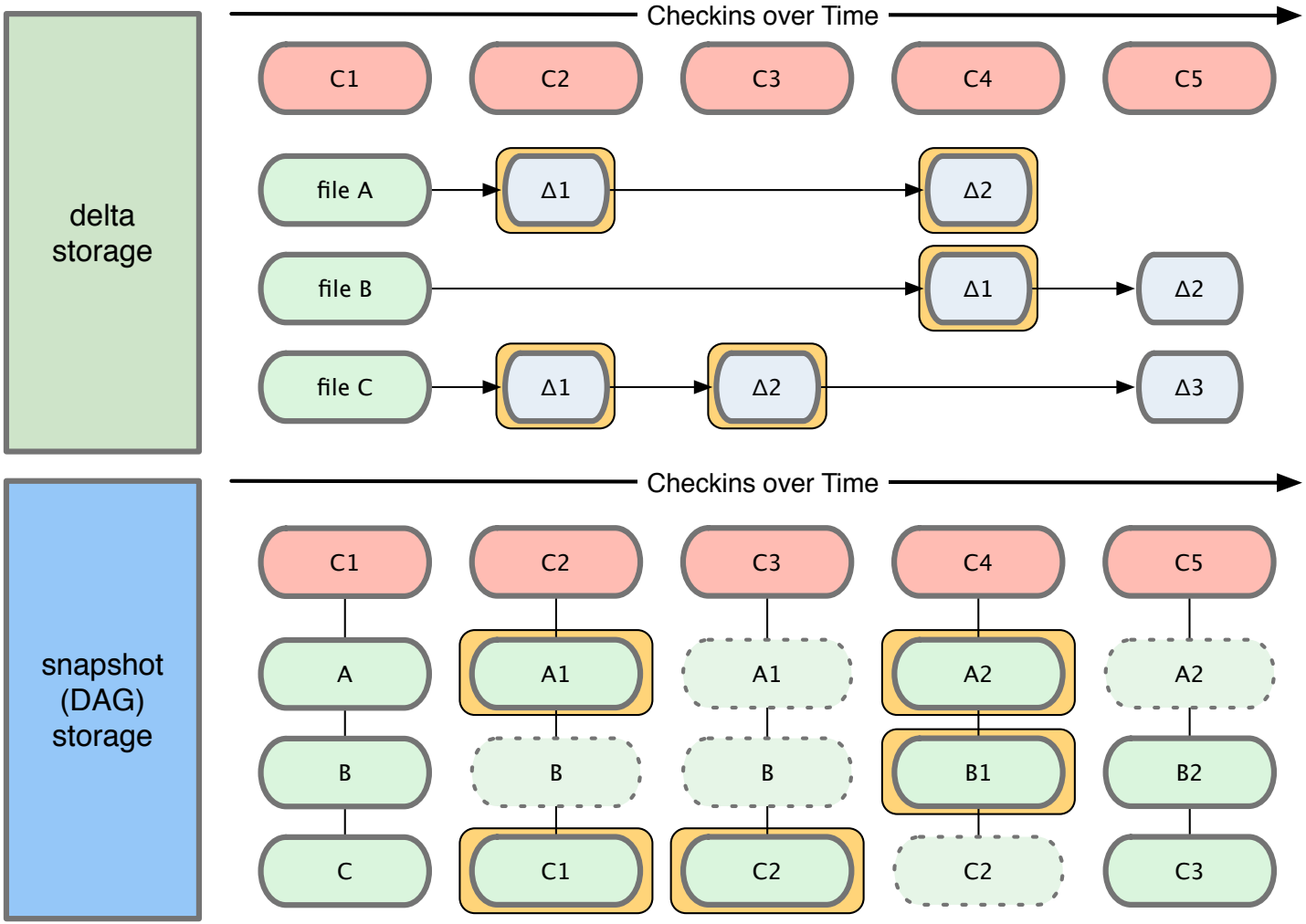
Tracking changes (Git)



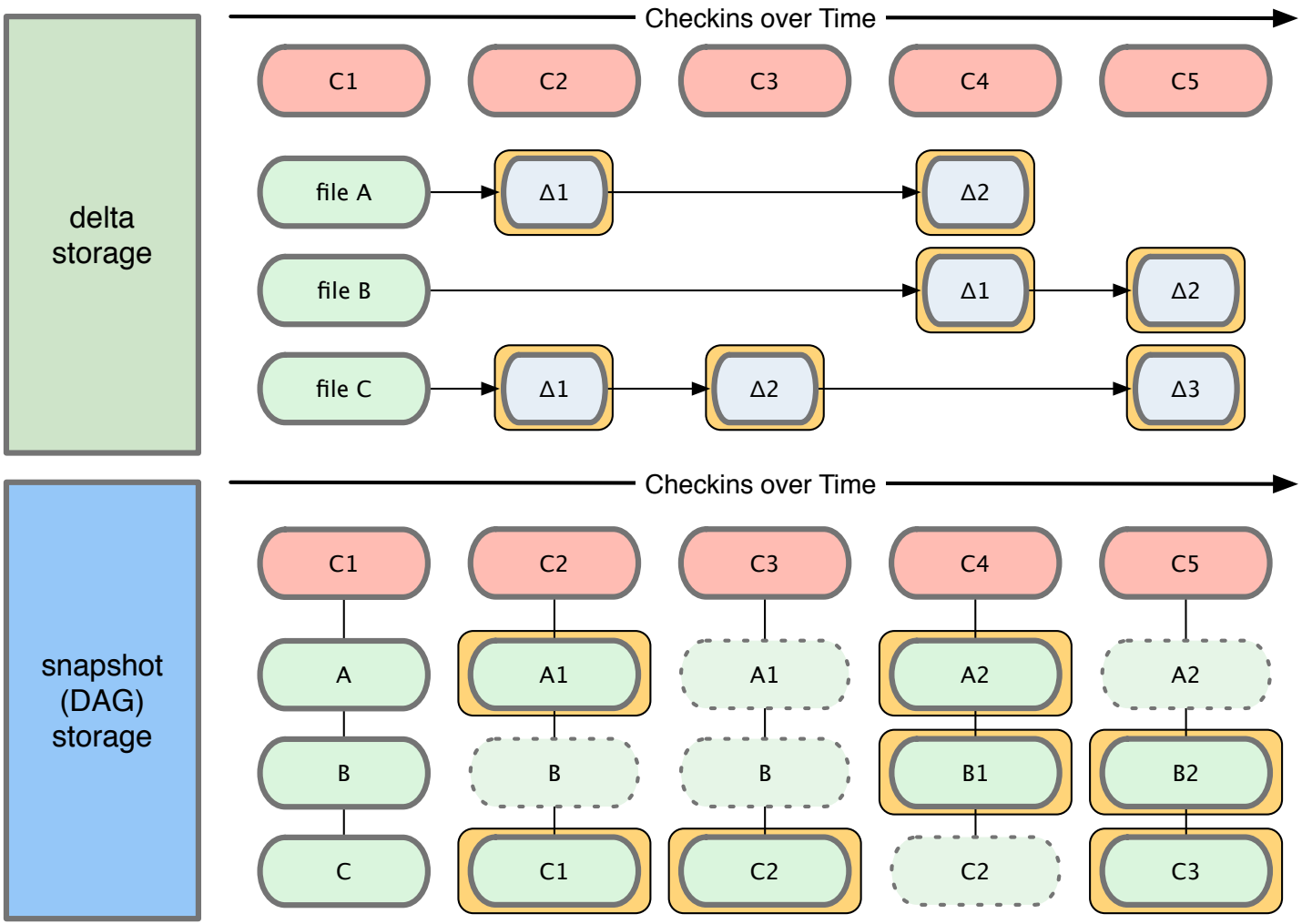
Tracking changes (Git)



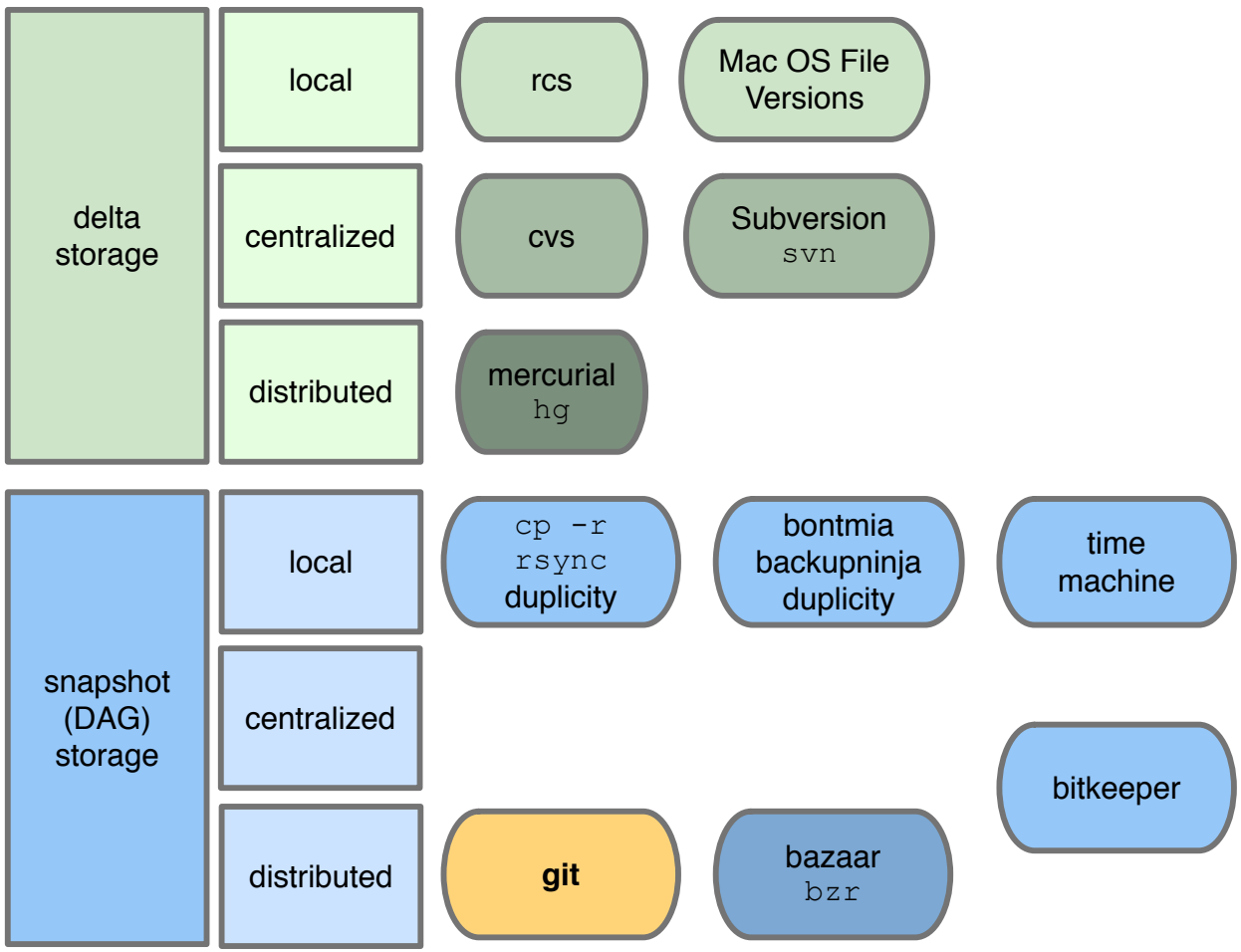
Tracking changes (Git)



Tracking changes (Git)



VCS Taxonomy



So what makes Git so useful?

(almost) Everything is local

- everything is fast
- every clone is a backup
- you work **mainly offline**

Ultra Fast, Efficient & Robust

- Snapshots, not patches (deltas)
- **Cheap branching and merging**
  - ↪ Strong support for thousands of parallel branches
- Cryptographic integrity everywhere

Other Git features

- **Git doesn't delete**
  - ↪ **Immutable** objects, Git generally only adds data
  - ↪ If you mess up, you can usually recover your stuff
    - ✓ Recovery can be tricky though

Other Git features

- **Git doesn't delete**
  - ↪ **Immutable** objects, Git generally only adds data
  - ↪ If you mess up, you can usually recover your stuff
    - ✓ Recovery can be tricky though

Git Tools / Extension

- cf. **Git submodules** or **subtrees**
- **Introducing git-flow**
  - ↪ workflow with a strict branching model
  - ↪ offers the `git` commands to follow the workflow

```
$> git flow init
$> git flow feature { start, publish, finish } <name>
$> git flow release { start, publish, finish } <version>
```

Summary

Summary

Installation Notes

git-scm.com

Linux / Mac OS

```
$> apt-get install git-core git-flow        # On Debian-like systems
$> yum install git gitflow                  # On CentOS-like systems
$> brew install git git-flow              # On Mac OS, using Homebrew
```

Installation Notes

git-scm.com

Linux / Mac OS

```
$> apt-get install git-core git-flow      # On Debian-like systems
$> yum install git gitflow                # On CentOS-like systems
$> brew install git git-flow              # On Mac OS, using Homebrew
```

Windows MsysGit

- Incl. Git Bash/GUI & Shell Integration
  - ↪ use PLINk from Putty
  - ↪ install **Git bash** + command prompt
  - ↪ select checkout windows / commit unix

Installation Notes

git-scm.com

Linux / Mac OS

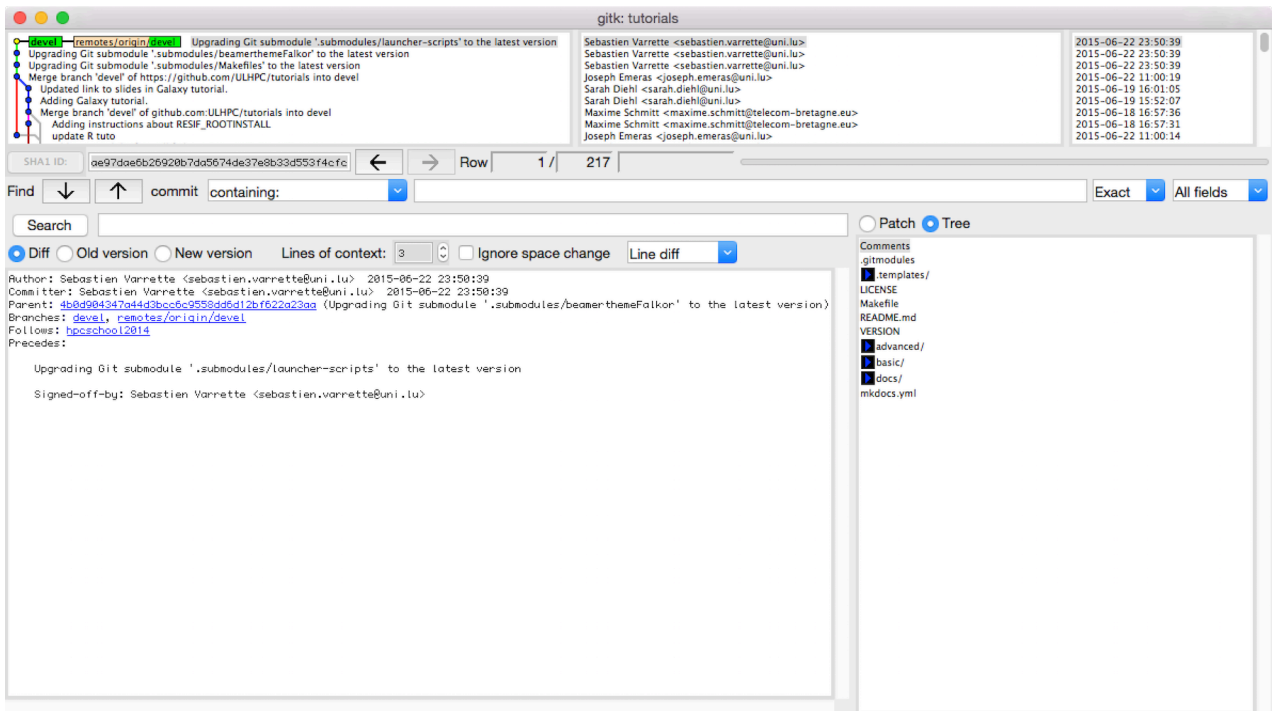
```
$> apt-get install git-core git-flow      # On Debian-like systems
$> yum install git gitflow                # On CentOS-like systems
$> brew install git git-flow              # On Mac OS, using Homebrew
```

Windows MsysGit

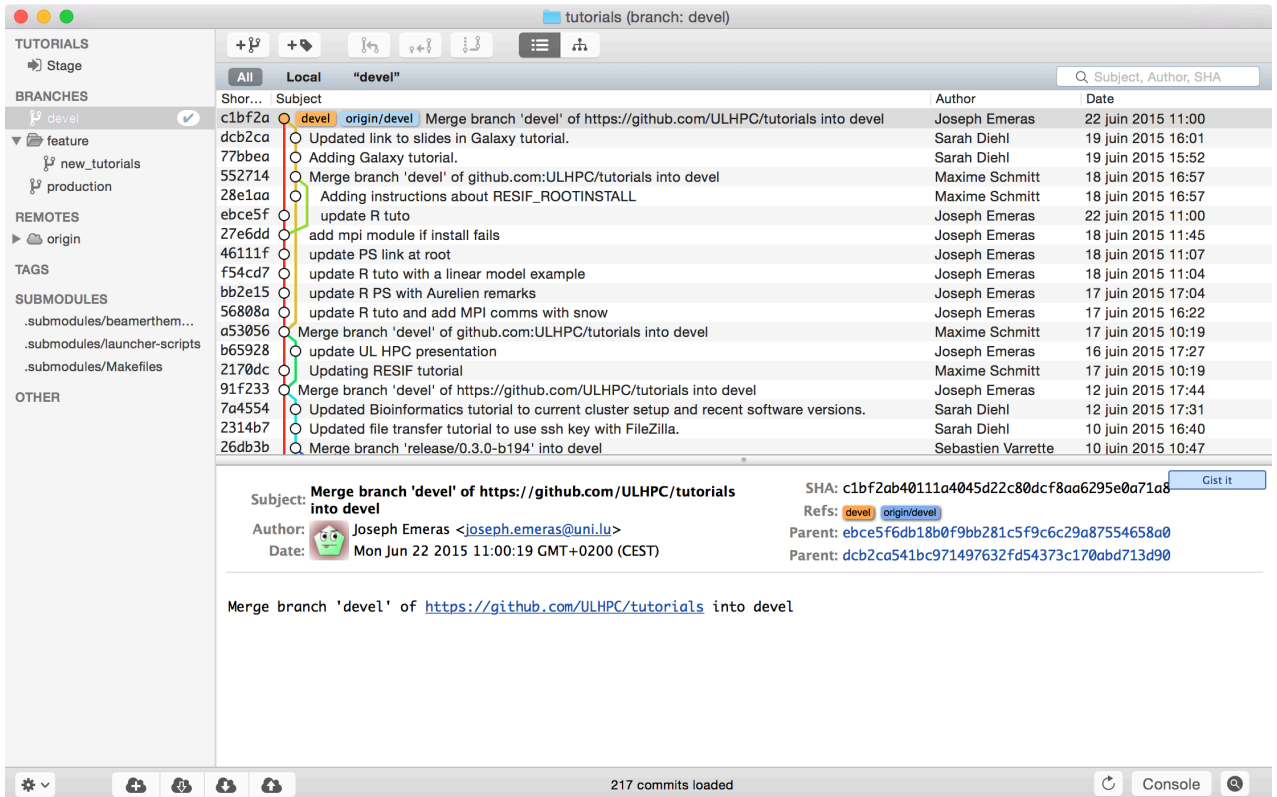
- Incl. Git Bash/GUI & Shell Integration
  - ↪ use PLINk from Putty
  - ↪ install **Git bash** + command prompt
  - ↪ select checkout windows / commit unix

Your Turn! Ensure you have git installed

Git GUI (default) Gitk

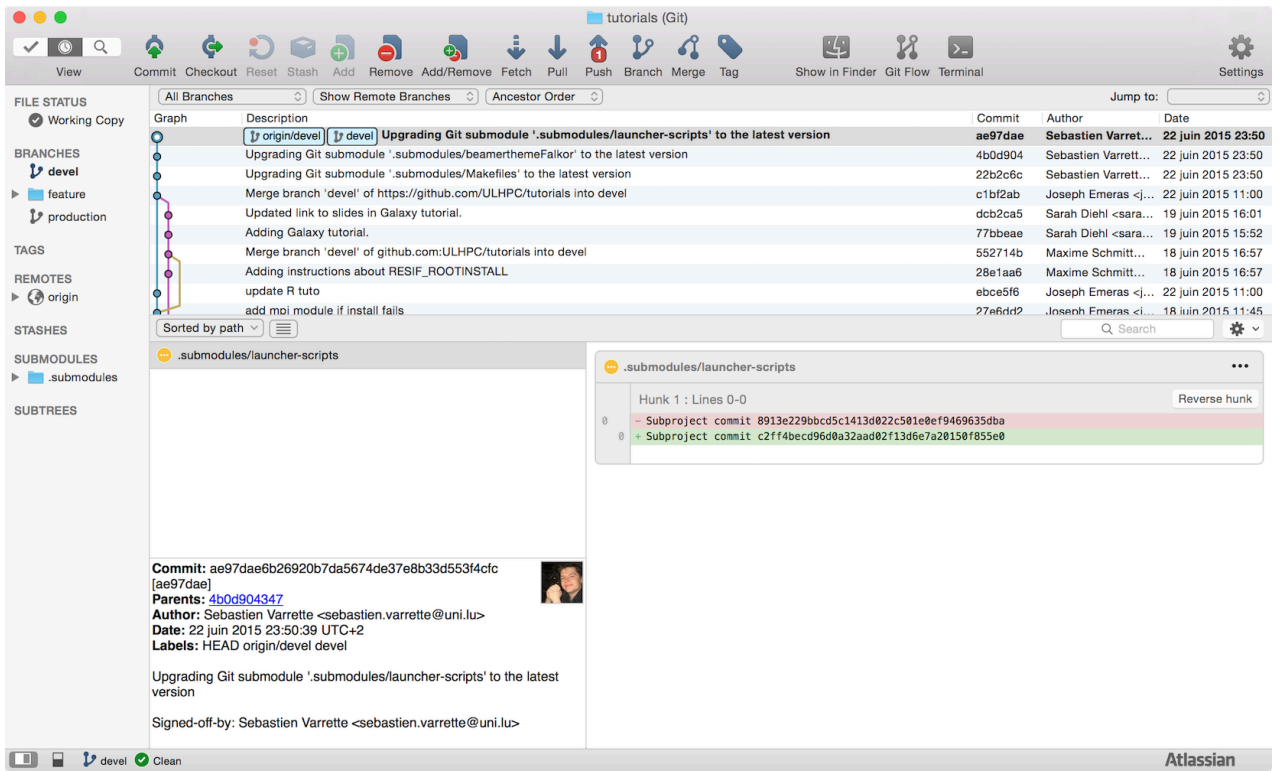


Git GUI (Mac OS) GitX-dev



http://rowanj.github.io/gitx/

Git GUI (Windows/Mac) SourceTree



http://www.sourcetreeapp.com/

1. Let it install a default git ignore file
2. make it load your SSH key created with Putty

Preliminary Configurations

- Global Git configuration are stored in `~/.gitconfig`
  - ↪ **Ex**: see my personal `.gitconfig`
- You **SHOULD** at least configure your name and email to commit
  - ↪ open a terminal (Git bash under windows) for the below commands

```
$> git config --global user.name "Firstname LastName"
$> git config --global user.email "Firstname.LastName@uni.lu"
$> git config --global color.ui true                    # Colors
$> git config --global core.editor vim                  # Editor
```

Preliminary Configurations

- Global Git configuration are stored in `~/.gitconfig`
  - ↪ **Ex**: see my personal `.gitconfig`
- You **SHOULD** at least configure your name and email to commit
  - ↪ open a terminal (Git bash under windows) for the below commands

```
$> git config --global user.name "Firstname LastName"
$> git config --global user.email "Firstname.LastName@uni.lu"
$> git config --global color.ui true                    # Colors
$> git config --global core.editor vim                  # Editor
```

Your Turn!

- Then check the changes by: `git config -l | grep user`

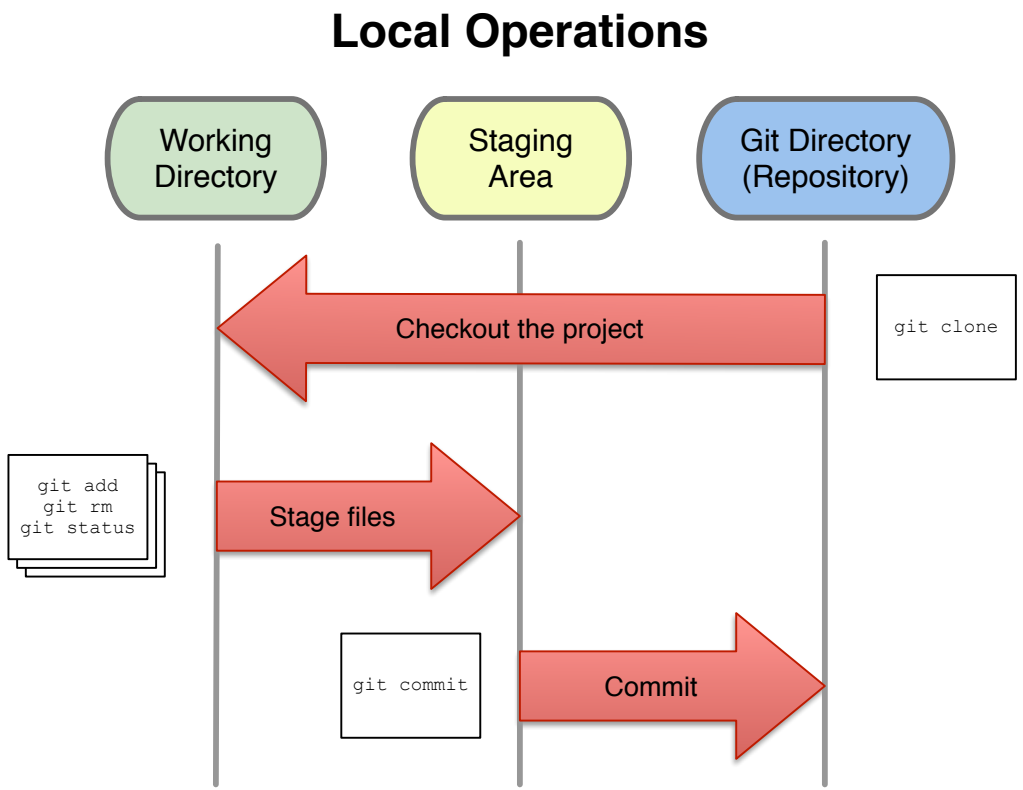
Git Commands Aliases

- You can also create git command aliases in `~/.gitconfig`.
  - ↪ **Ex** copy/paste from my personal `.gitconfig`

```
[alias]
    up = pull origin
    pu = push origin
    st = status
    df = diff
    ci = commit -s
    co = checkout
    br = branch
    w  = whatchanged --abbrev-commit
    ls = ls-files
    gr = log --graph --oneline --decorate
    amend = commit --amend
```

Summary

The Three (local) States



- The local repository lives in the `.git` directory.
- The **staging area** tracks what will go into the next commit
  - ↪ AKA "the index"

Summary

Creating a Repository

```
$> git [flow] init
```

- Initializes a new git (flow) repository in the current directory

Creating a Repository

```
$> git [flow] init
```

- Initializes a new git (flow) repository in the current directory

Your Turn!

```
$> cd /tmp
$> mkdir firstproject
$> cd firstproject

$> git init
Initialized empty Git repository in /private/tmp/firstproject/.git/
```

Cloning a Repository

```
$> git clone [--recursive] <url> [<path>]
```

Type	URL Format / Example	Port
Local	`/path/to/project.git`	n/a
SSH	`git+ssh://user@server:port/project.git`	22
Git	`git://server/project.git`	9418
HTTPS	`https://github.com/Falkor/falkorlib.git`	443

Cloning a Repository

```
$> git clone [--recursive] <url> [<path>]
```

Your Turn!

```
$> cd /tmp
$> git clone https://github.com/ULHPC/tutorials.git
Cloning into 'tutorials'...
remote: Counting objects: 1247, done.
remote: Compressing objects: 100% (63/63), done.
remote: Total 1247 (delta 32), reused 0 (delta 0), pack-reused 1181
Receiving objects: 100% (1247/1247), 15.74 MiB | 3.08 MiB/s, done.
Resolving deltas: 100% (588/588), done.
Checking connectivity... done.
$> git clone --recursive \
      https://github.com/ULHPC/tutorials.git /tmp/tutorials2
```

Inspecting a Repository

```
$> git status [-s]          # -s:  short / simplified output
```

Inspecting a Repository

```
$> git status [-s]          # -s:  short / simplified output
```

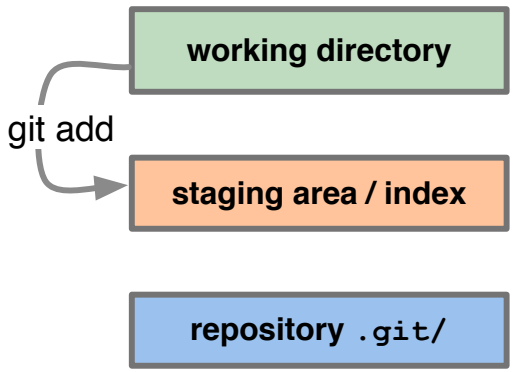
Your Turn!

```
$> cd /tmp/firstproject
$> git status
On branch master

Initial commit

nothing to commit

# Create an empty file
$> touch README.md
```

```
$> git status
On branch master

Initial commit

Untracked files:
    README

nothing added to commit but untracked
files present
$> git status -s
?? README
```

Add / Tracking [new] file(s)

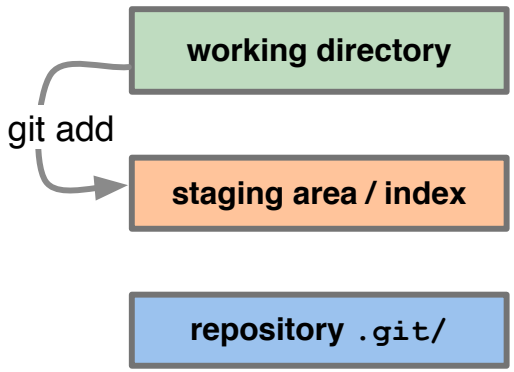
```
$> git add [-f] <pattern>
```



- Adds changes to the index
  - ↪ Add a specific file: `git add README`
  - ↪ Add a set of files: `git add *.py`
- Beware that empty directory cannot be added **directly**
  - ↪ due to the internal file representation (**blobs**)
  - ↪ **Tips**: add an hidden file `.empty` (or `.gitignore`)

Add / Tracking [new] file(s)

```
$> git add [-f] <pattern>
```



- Adds changes to the index
  - ↪ Add a specific file: `git add README`
  - ↪ Add a set of files: `git add *.py`
- Beware that empty directory cannot be added **directly**
  - ↪ due to the internal file representation (**blobs**)
  - ↪ **Tips**: add an hidden file `.empty` (or `.gitignore`)

Your Turn!

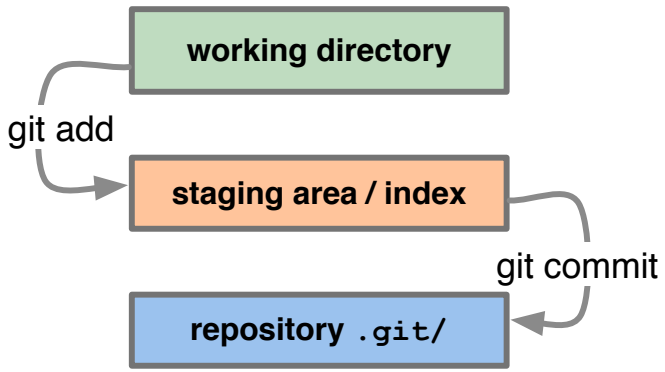
```
$> cd /tmp/firstproject
$> git status -s
?? README
```

```
$> git add README
$> git status -s
A  README
```

Committing your changes

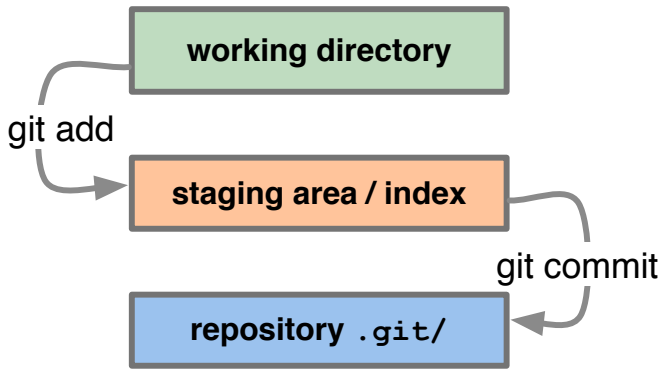
```
$> git commit [-s] [-m "msg"]
```

- Commit all changes: `git commit -a`



Committing your changes

```
$> git commit [-s] [-m "msg"]
```



- Commit all changes: `git commit -a`

Your Turn!

```
$> cd /tmp/firstproject
$> git commit -s -m "add README"    # OR git ci -m "add README"
[master (root-commit) ee60f53] add README
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README
$> git status               # OR git st
On branch master
nothing to commit, working directory cleant
```

Removing Files

```
$> git rm [-rf] [--cached] <file>
```

- `--cached`: remove from Staging area
  - ↪ otherwise (default): from index **and** file system

Ignoring Files

Ignoring files from staging: '.gitignore'

- you can create a `.gitignore` file listing patterns to ignore
  - ↪ Blank lines or lines starting with \# are ignored
  - ↪ End pattern with slash (/) to specify a directory
  - ↪ Negate pattern with exclamation point (!)
- Collection of useful `.gitignore` templates

```
.DS_Store
*~
```

```
*.asv
*.m~
*.mex*
tmp/*
```

- LaTeX `.gitignore`
- Python `.gitignore`
- Ruby `.gitignore`

Moving Files

```
$> git mv <source> <destination>
```

```
# Equivalent of:
mv <source> <destination>
git rm <source>
git add <destination>
```

Moving Files

```
$> git mv <source> <destination>
```

```
# Equivalent of:
mv <source> <destination>
git rm <source>
git add <destination>
```

Your Turn!

```
$> cd /tmp/firstproject
$> git mv README README.md
$> git status
On branch master
Changes to be committed:
    renamed:    README -> README.md
$> git commit -m "a first move"
```

Check the Commit History

```
$> git log [-p] [--stat] [--graph --oneline --decorate]
```

- `-p` / `--stat`: show the differences introduced in each commit
- You can also perform some date filtering

  ```
  $> git log --since=2.weeks
  ```

- Ncurses-based text-mode interface: `tig`

Check the Commit History

```
$> git log [-p] [--stat] [--graph --oneline --decorate]
```

- `-p` / `--stat`: show the differences introduced in each commit
- You can also perform some date filtering

  ```
  $> git log --since=2.weeks
  ```

- Ncurses-based text-mode interface: `tig`

Your Turn!

```
$> cd /tmp/firstproject
$> git log --oneline --graph --decorate     # OR git gr
* f1f0c27 (HEAD -> master) a first move
* ee60f53 add README
$> git log -p -1          # only the last commit OR git show
$> tig
```

Show differences

```
$> git diff [--cached] [<ref>]
```

- Check **un-staged** changes: `git diff`
  - ↪ `--cached`: check **staged** changes
- Relative to a specific revision:

```
$> git diff 1776f5
$> git diff HEAD^
```

Undoing Things

```
$> git commit --amend                    # Change the last commit
```

Undoing Things

```
$> git commit --amend                    # Change the last commit
```

```
$> git unstage <file>                    # or git reset HEAD <file>
```

Undoing Things

```
$> git commit --amend                    # Change the last commit
```

```
$> git unstage <file>                    # or git reset HEAD <file>
```

```
$> git checkout -- <file>   # DANGER! Un-modify modified file
```

- Restore to the last committed/cloned version: **all** changes are lost!

Undoing Things

```
$> git commit --amend                    # Change the last commit
```

```
$> git unstage <file>                    # or git reset HEAD <file>
```

```
$> git checkout -- <file>   # DANGER! Un-modify modified file
```

```
$> git revert <commit>                   # revert a <commit>
```

- Make a new commit that undoes all changes made in <commit>

Undoing Things

```
$> git commit --amend                    # Change the last commit
```

```
$> git unstage <file>                    # or git reset HEAD <file>
```

```
$> git checkout -- <file>   # DANGER! Un-modify modified file
```

```
$> git revert <commit>                   # revert a <commit>
```

Your Turn!

```
$> cd /tmp/firstproject
$> git commit --amend
$> echo 'toto' >> README.md
```

```
$> cat README.md && git status
$> git checkout -- README
$> git status
```

Summary

Basic Workflow

Edit files	`vim` / `emacs` / `subl` ...
Stage the changes	`git add`
Review your changes	`git status`
Commit the changes	`git commit`

Summary

For cheaters: A Basicerer Workflow

Edit files	`vim` / `emacs` / `subl` ...
Stage & commit the changes	`git commit -a`

Summary

For cheaters: A Basicerer Workflow

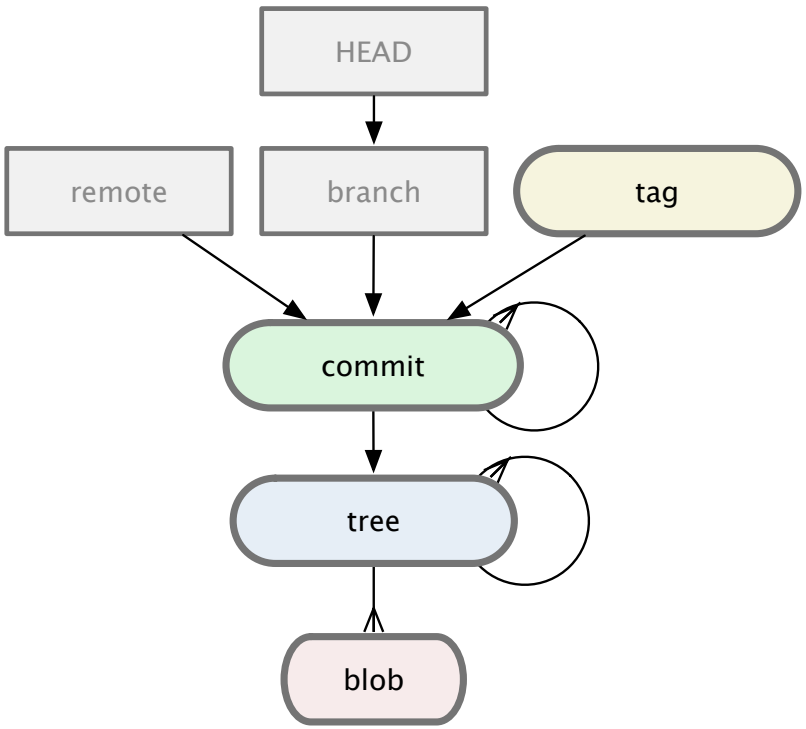
Edit files	`vim` / `emacs` / `subl` ...
Stage & commit the changes	`git commit -a`

Advices

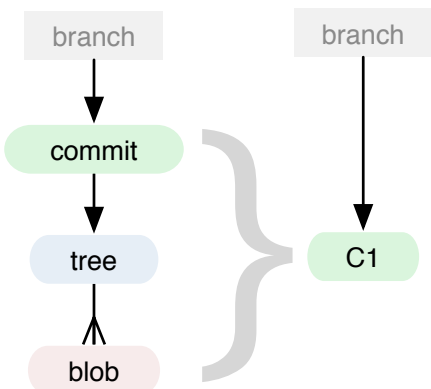
- **Commit early, commit often!**
  - ↪ commits = save points
  - ↪ use descriptive commit messages
- Don't get out of sync with your collaborators
- Commit the sources, not the derived files

Summary

Data Model



- **Immutable** objects
  - ↪ **Blob**: File content
  - ↪ **Tree**: Directory List
  - ↪ **Commit**: Pointer to a snapshot / tree
  - ↪ **Tag**: Pointer to commit
- **Git Branch**: Lighweight, movable pointer to a commit (`HEAD`: current branch)



Data Model Example

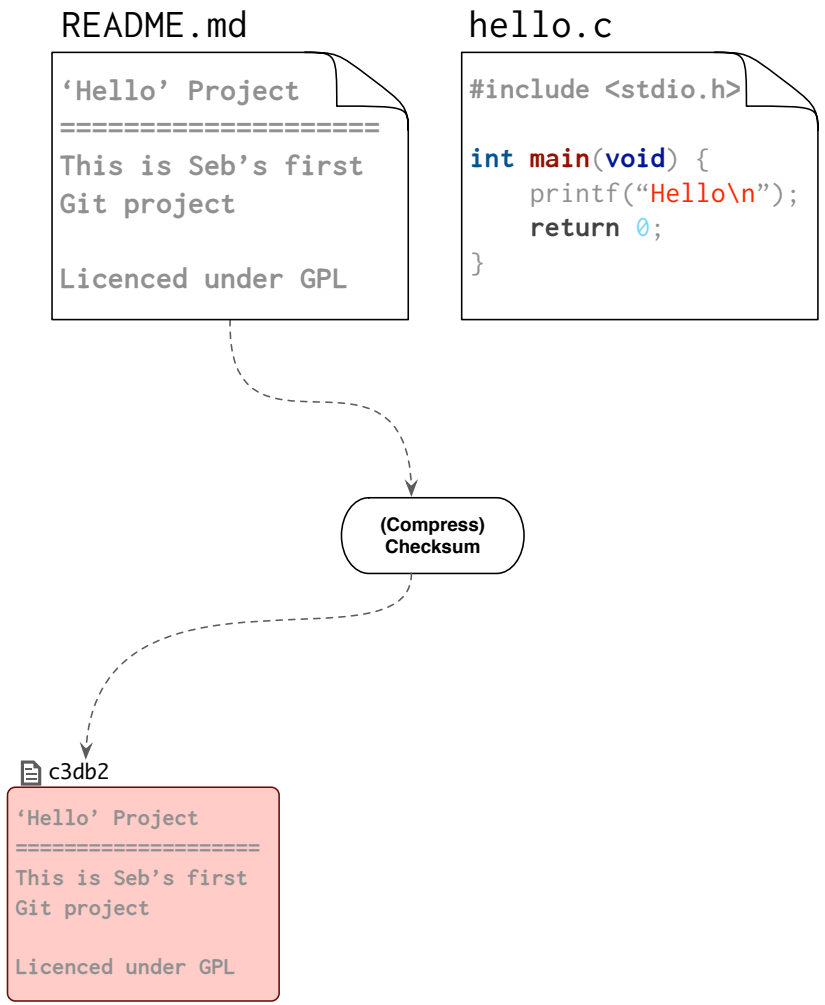
README.md

```
'Hello' Project
===================
This is Seb's first
Git project

Licenced under GPL
```

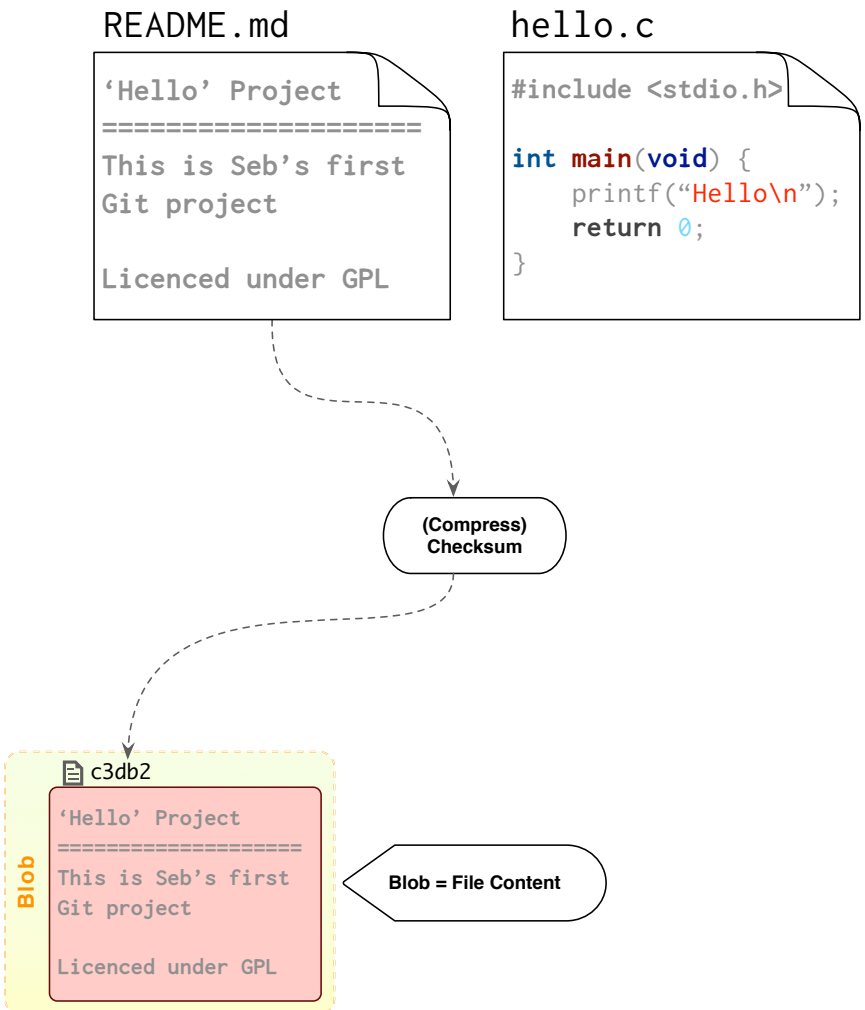
hello.c

```c
#include <stdio.h>

int main(void) {
    printf("Hello\n");
    return 0;
}
```

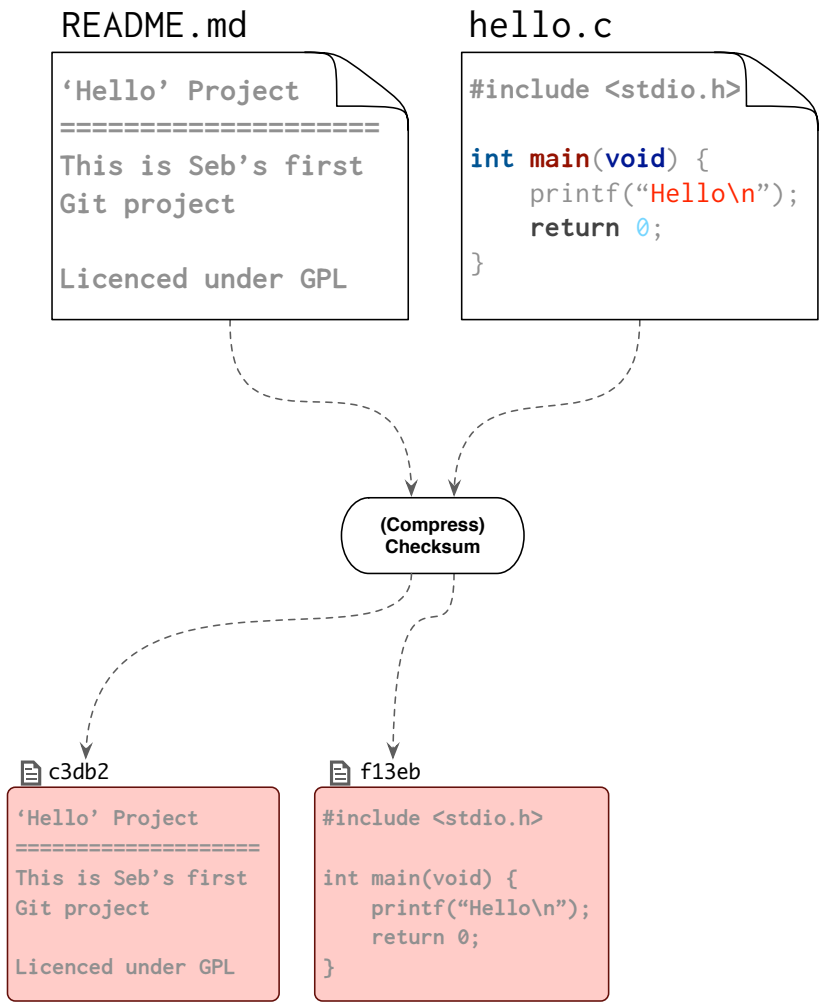
Data Model Example



Data Model Example



Data Model Example



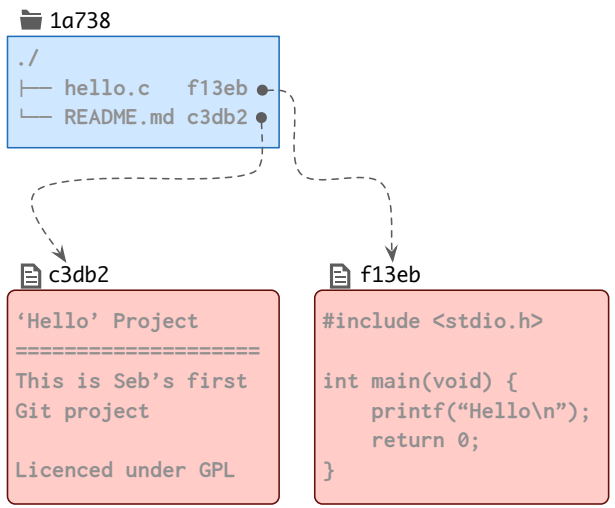
Data Model Example

README.md

```
'Hello' Project
===============
This is Seb's first
Git project

Licenced under GPL
```

hello.c

```
#include <stdio.h>

int main(void) {
    printf("Hello\n");
    return 0;
}
```



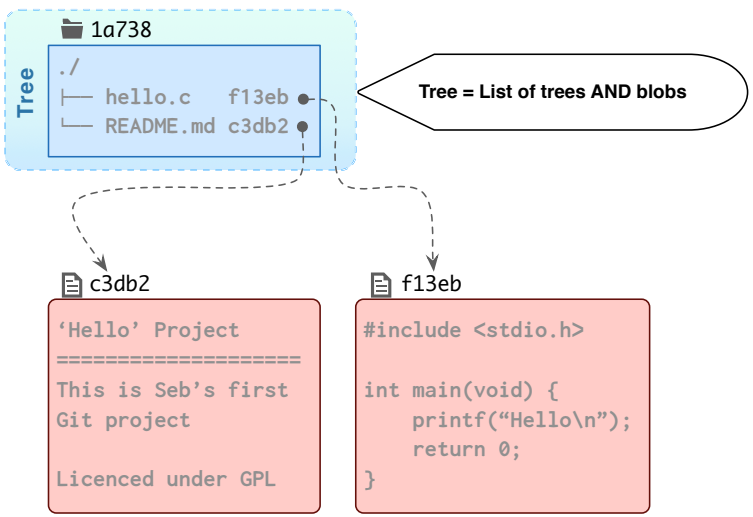
Data Model Example

README.md

```
'Hello' Project
===============
This is Seb's first
Git project

Licenced under GPL
```

hello.c

```
#include <stdio.h>

int main(void) {
    printf("Hello\n");
    return 0;
}
```



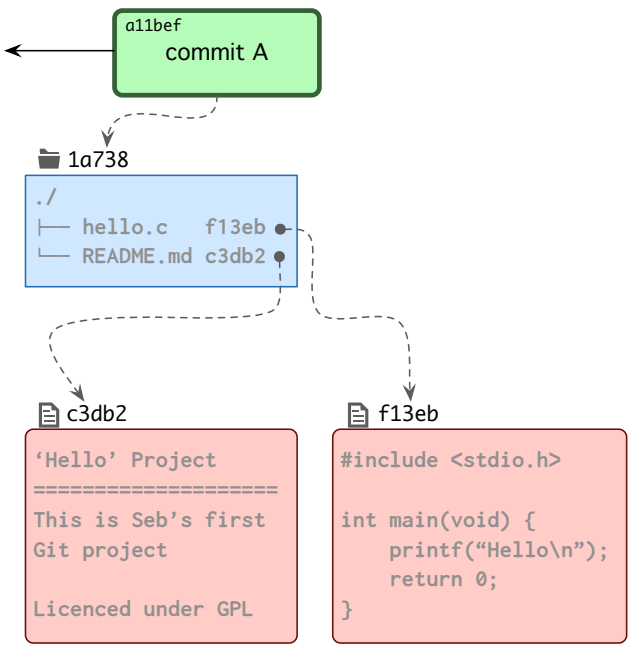
Data Model Example

README.md

```
'Hello' Project
===============
This is Seb's first
Git project

Licenced under GPL
```

hello.c

```
#include <stdio.h>

int main(void) {
    printf("Hello\n");
    return 0;
}
```



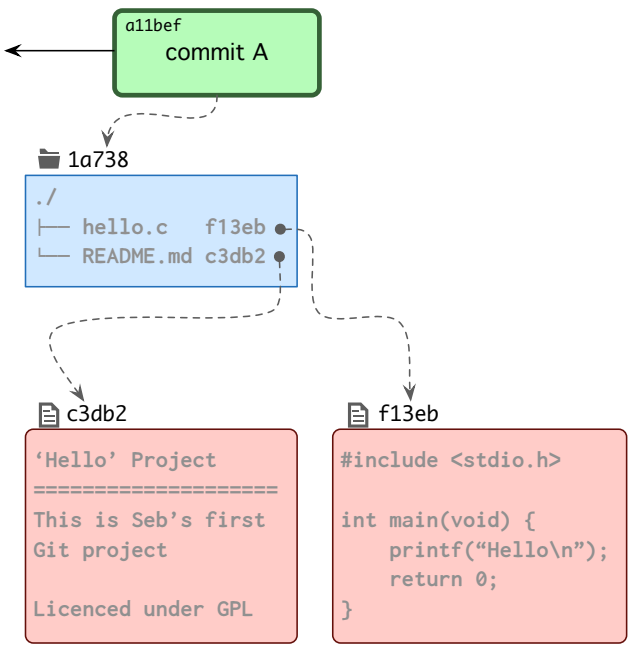
Data Model Example

README.md

```
'Hello' Project
===============
This is Seb's first
Git project

Licenced under GPL
```

hello.c

```
#include <stdio.h>

int main(void) {
    printf("Hello\n");
    return 0;
}
```



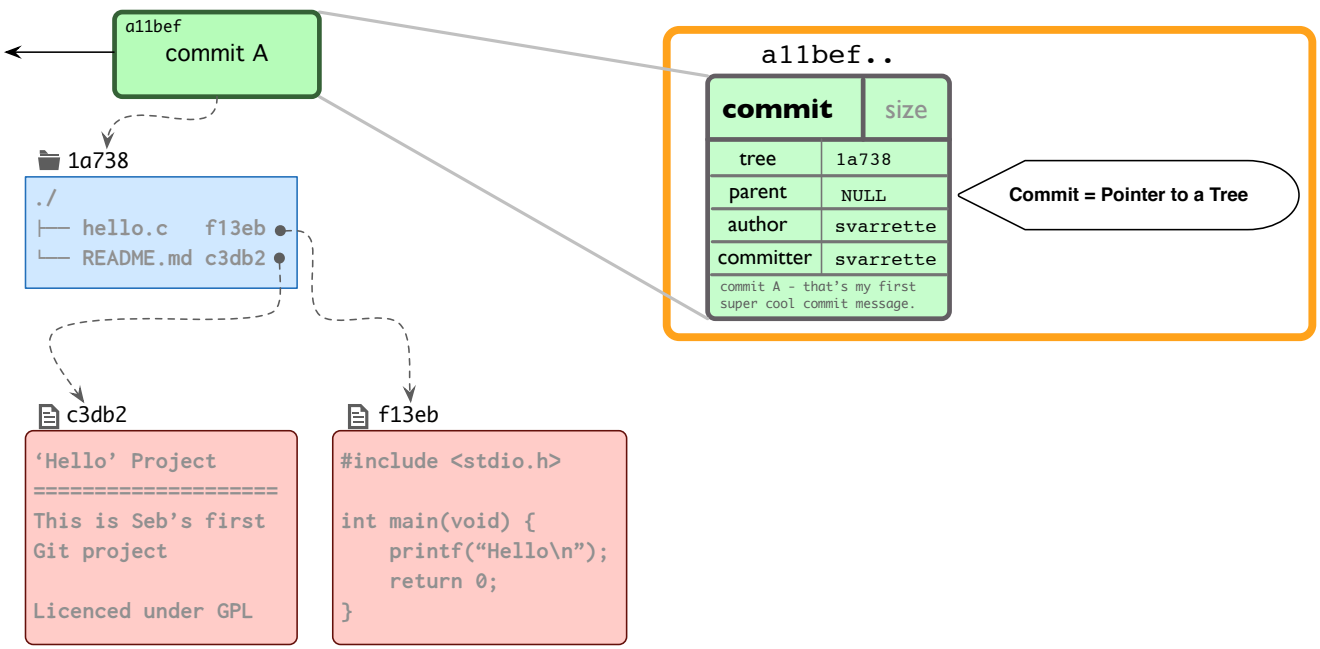
Data Model Example

README.md

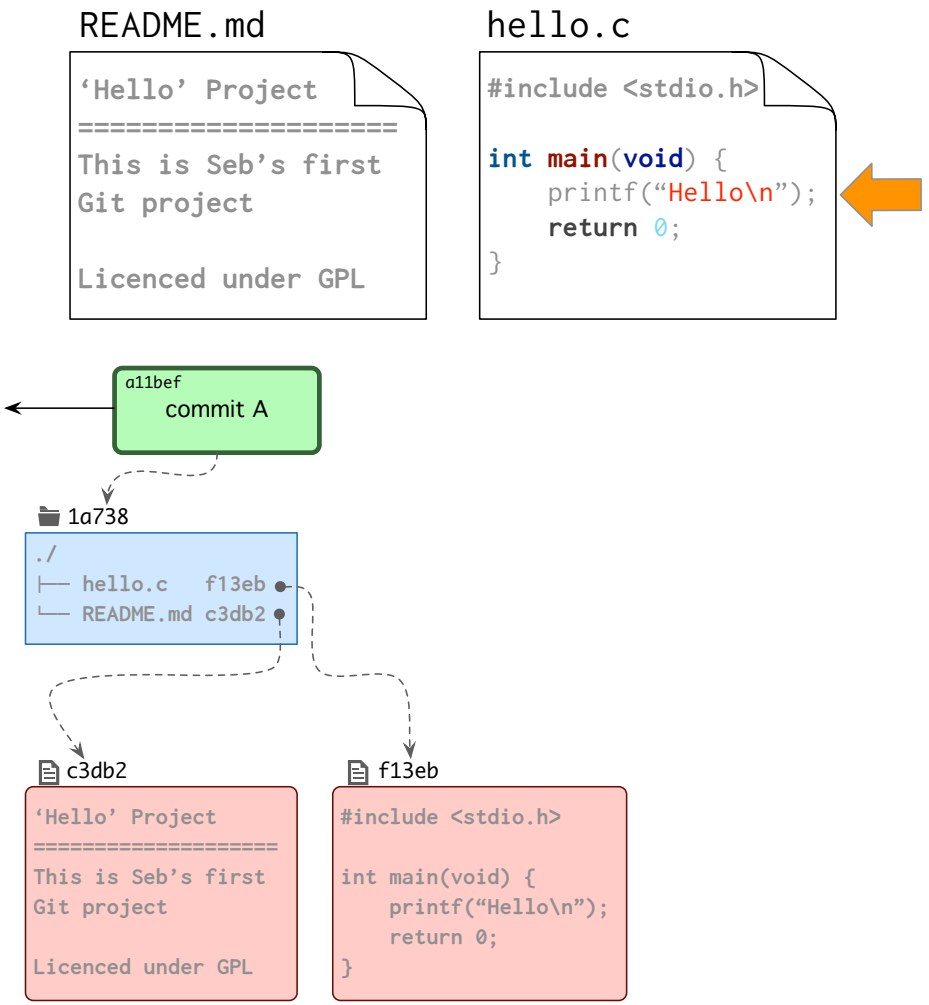
```
'Hello' Project
===============
This is Seb's first
Git project

Licenced under GPL
```

hello.c

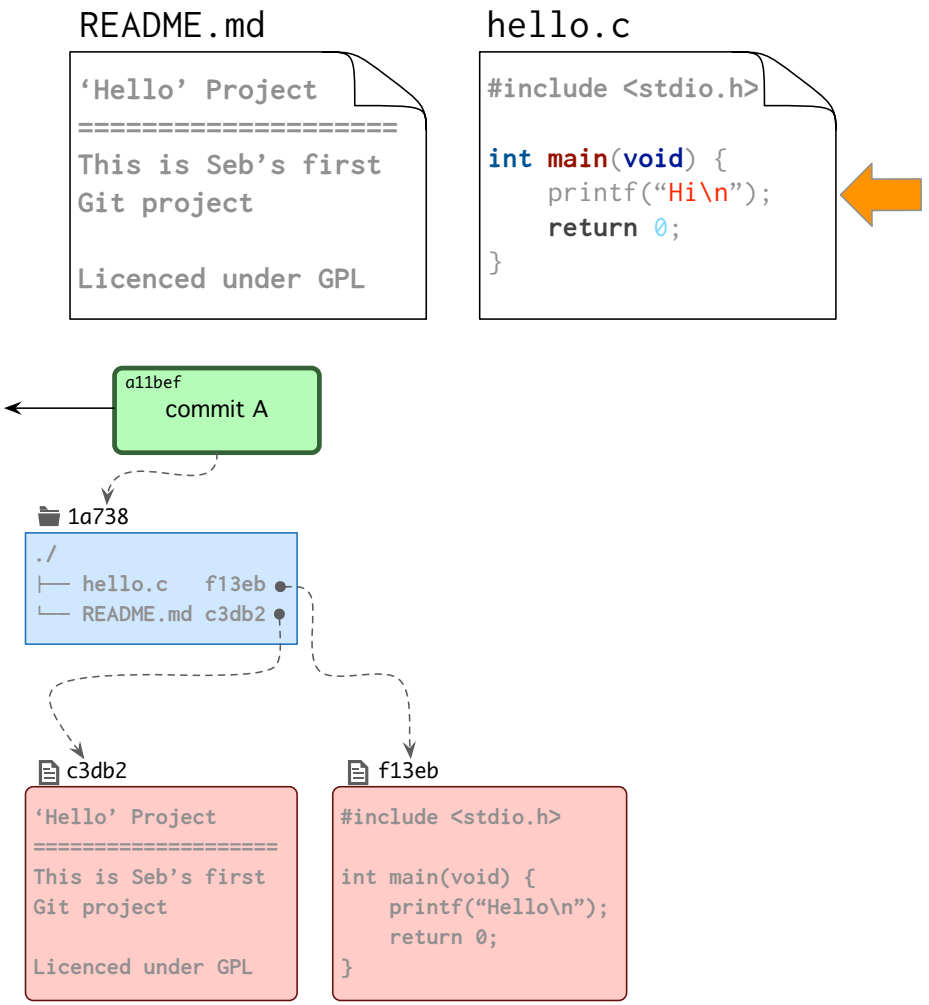
```
#include <stdio.h>

int main(void) {
    printf("Hello\n");
    return 0;
}
```



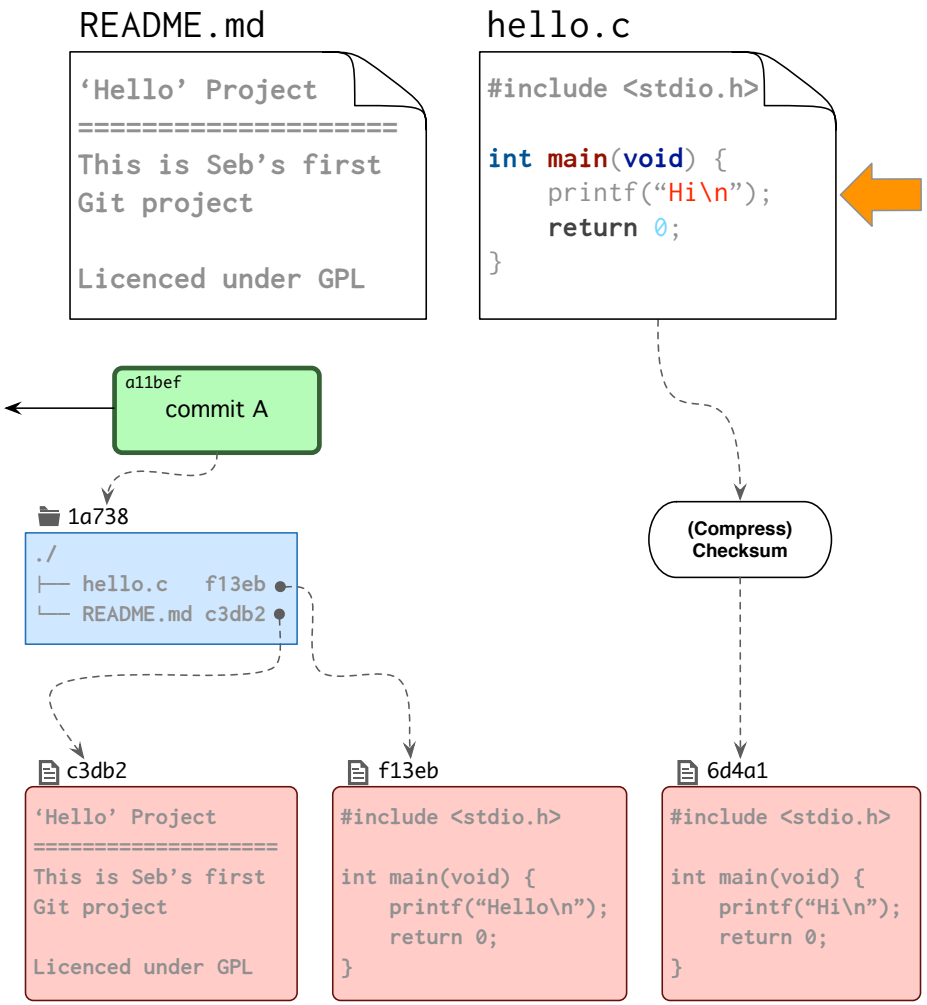
Data Model Example



Data Model Example



Data Model Example



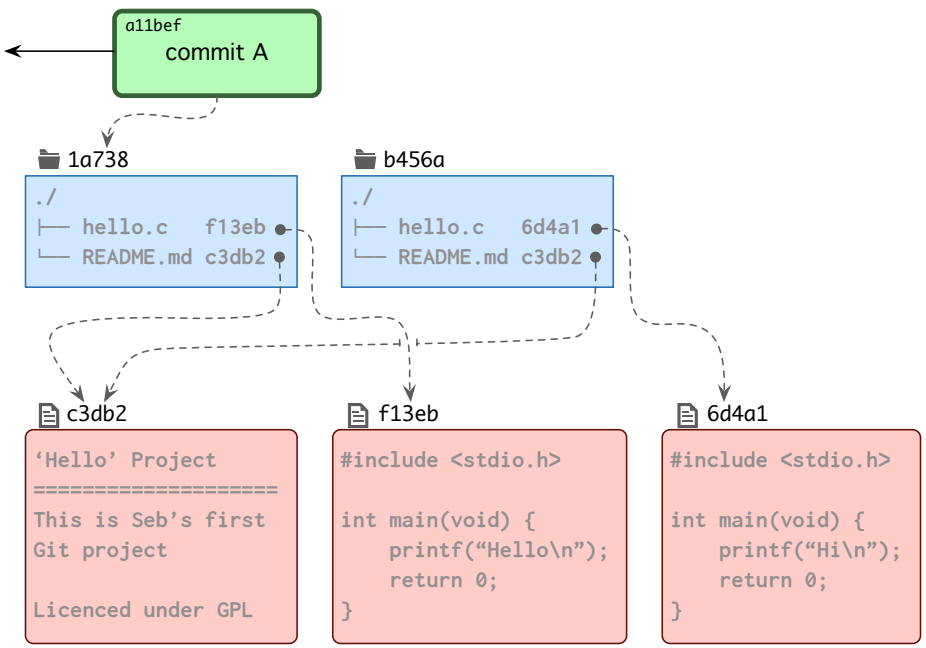
Data Model Example

README.md

```
'Hello' Project
===================
This is Seb's first
Git project

Licenced under GPL
```

hello.c

```
#include <stdio.h>

int main(void) {
    printf("Hi\n");
    return 0;
}
```



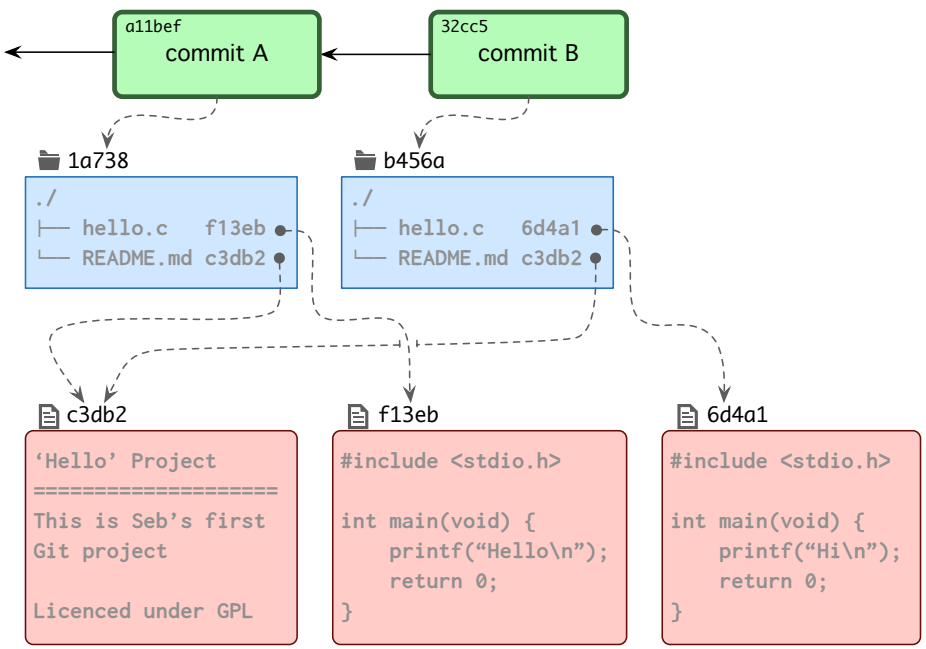
Data Model Example

README.md

```
'Hello' Project
===============
This is Seb's first
Git project

Licenced under GPL
```

hello.c

```
#include <stdio.h>

int main(void) {
    printf("Hi\n");
    return 0;
}
```



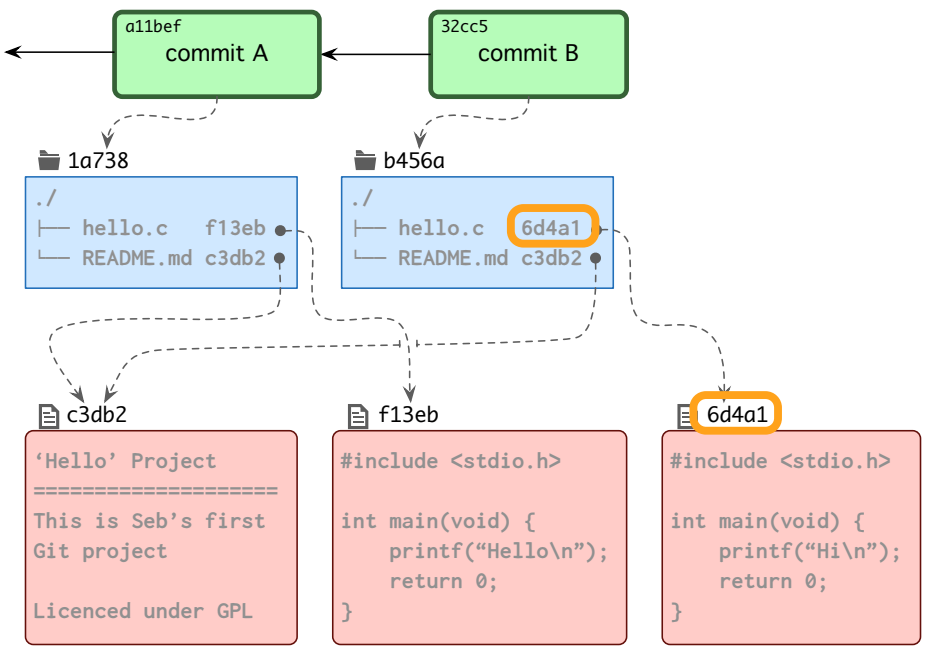
Data Model Example

README.md

```
'Hello' Project
===============
This is Seb's first
Git project

Licenced under GPL
```

hello.c

```
#include <stdio.h>

int main(void) {
    printf("Hi\n");
    return 0;
}
```



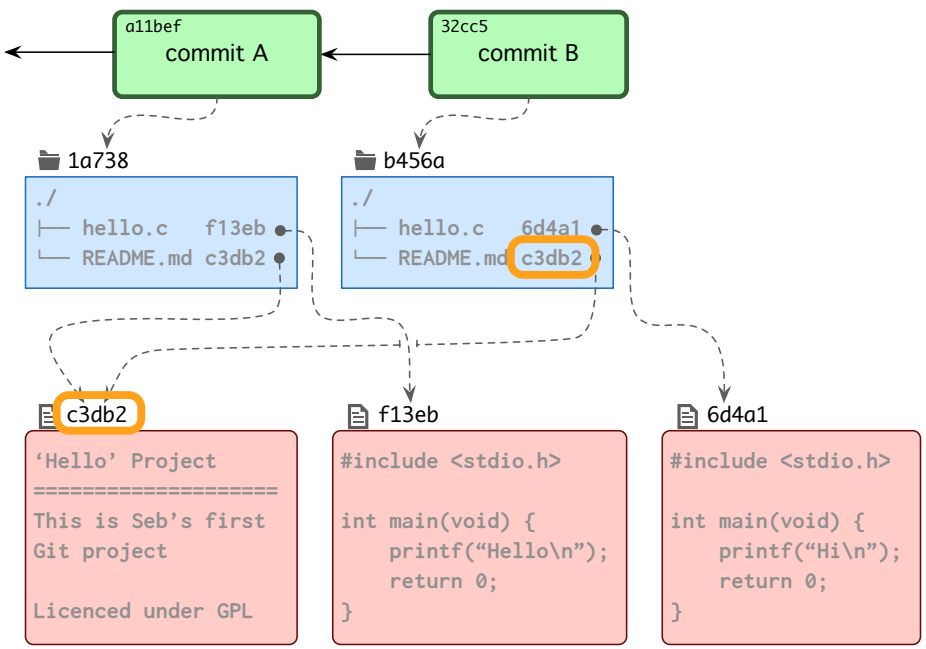
Data Model Example

README.md

```
'Hello' Project
===================
This is Seb's first
Git project

Licenced under GPL
```

hello.c

```
#include <stdio.h>

int main(void) {
    printf("Hi\n");
    return 0;
}
```



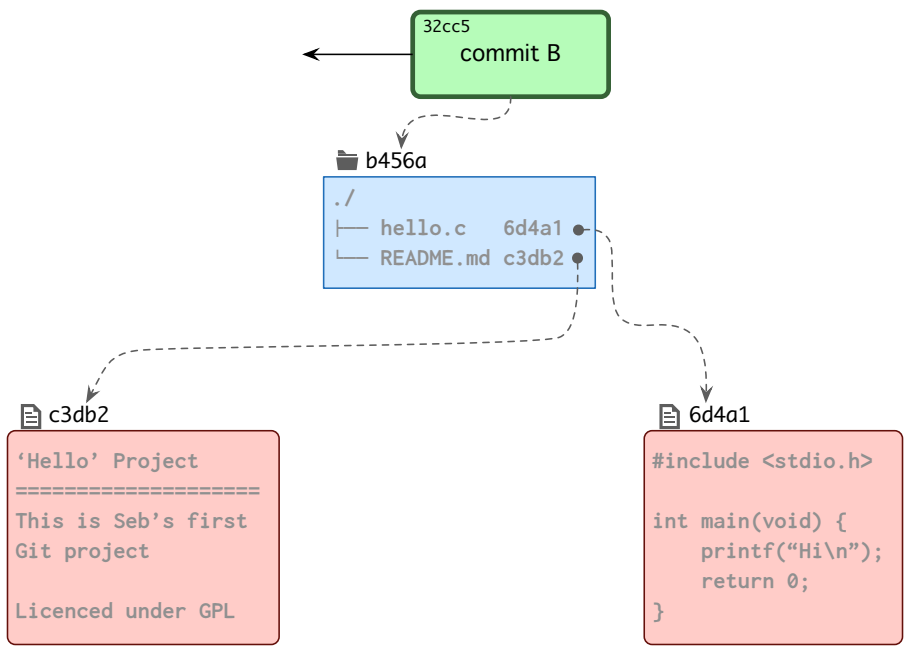
Data Model Example

README.md

```
'Hello' Project
===============
This is Seb's first
Git project

Licenced under GPL
```

hello.c

```
#include <stdio.h>

int main(void) {
    printf("Hi\n");
    return 0;
}
```



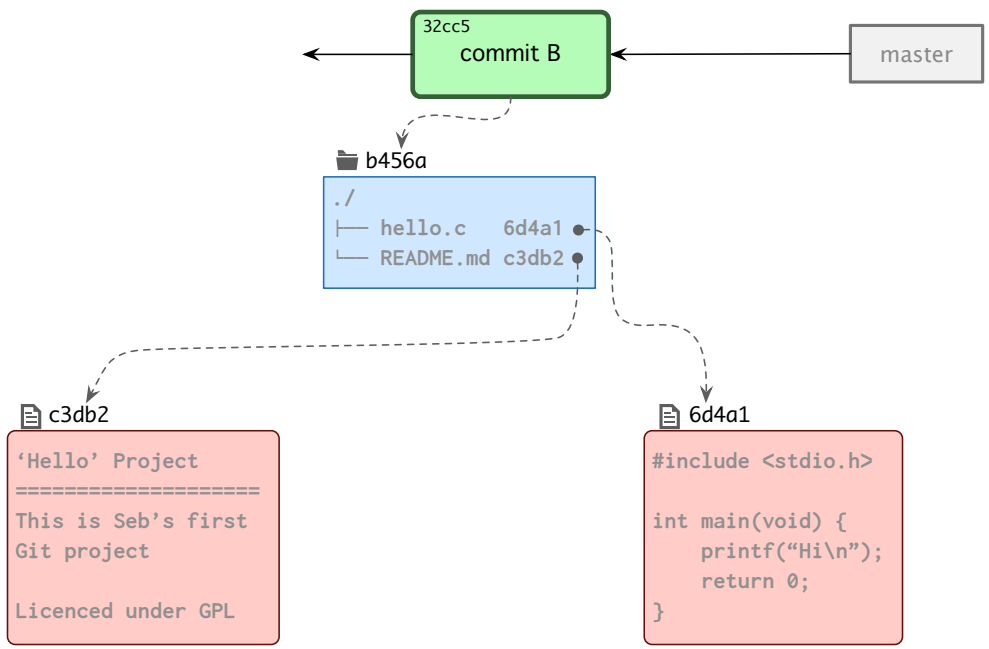
Data Model Example

README.md

```
'Hello' Project
===============
This is Seb's first
Git project

Licenced under GPL
```

hello.c

```
#include <stdio.h>

int main(void) {
    printf("Hi\n");
    return 0;
}
```



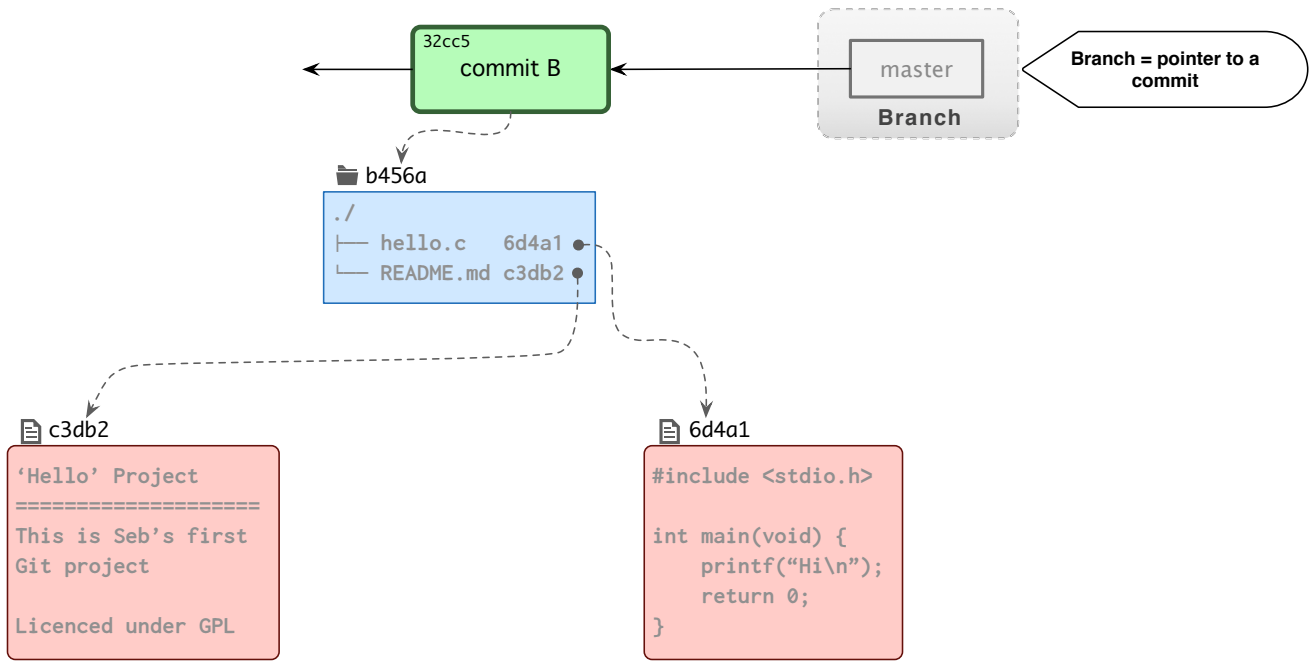
Data Model Example

README.md

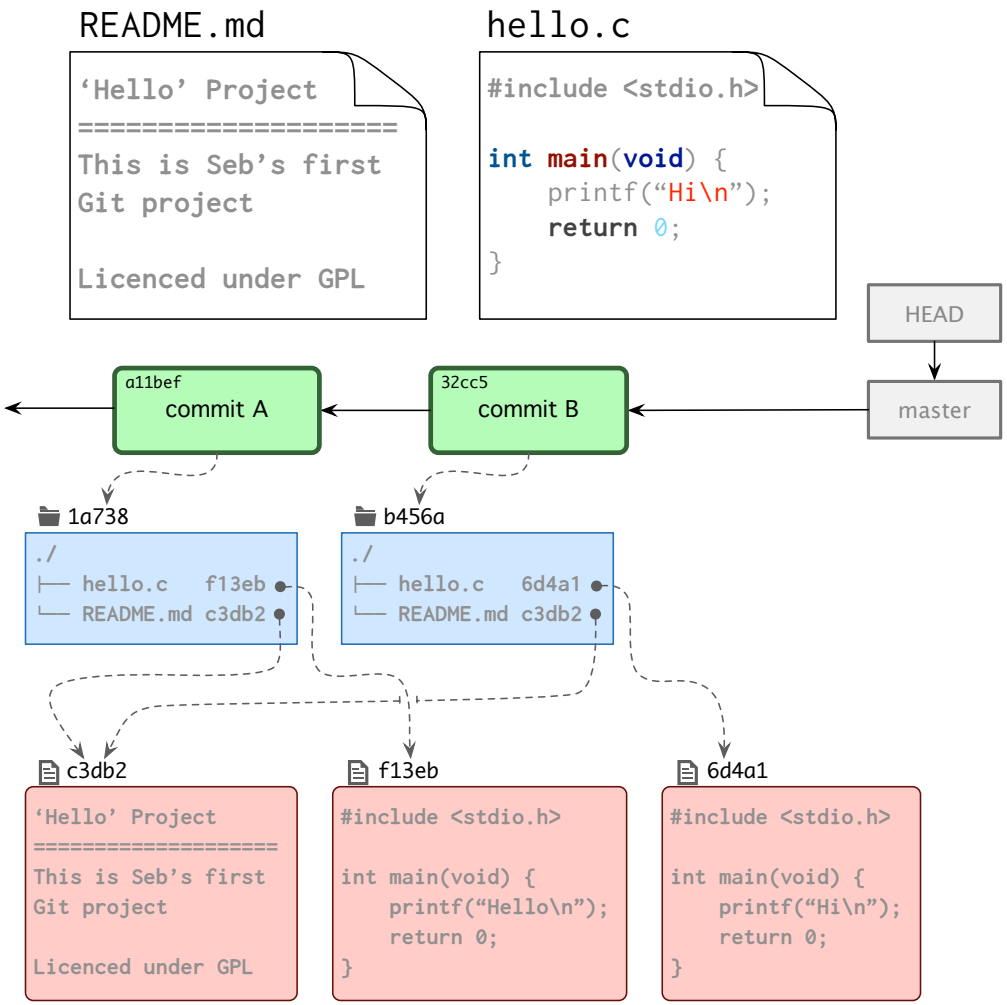
```
'Hello' Project
===============
This is Seb's first
Git project

Licenced under GPL
```

hello.c

```
#include <stdio.h>

int main(void) {
    printf("Hi\n");
    return 0;
}
```



Data Model Example



Branching

```
$> git branch <name>                    # create a branch <name>
```

Branching

```
$> git branch <name>                    # create a branch <name>
```

```
$> git branch -d <name>                  # delete the branch <name>
```

- use `-D` instead of `-d` to force deletion

Branching

```
$> git branch <name>                    # create a branch <name>
```

```
$> git branch -d <name>                 # delete the branch <name>
```

```
$> git branch [-a]                      # List [all] the branches
```

Branching

```
$> git branch <name>                    # create a branch <name>
```

```
$> git branch -d <name>                 # delete the branch <name>
```

```
$> git branch [-a]                      # List [all] the branches
```

```
$> git checkout [-b] <name>     # swicth to the branch <name>
```

- -b: create the branch before switching
- changes commited through `git commit` are committed to `HEAD`

Branching

```
$> git branch <name>                    # create a branch <name>
```

```
$> git branch -d <name>                 # delete the branch <name>
```

```
$> git branch [-a]                      # List [all] the branches
```

```
$> git checkout [-b] <name>             # swicth to the branch <name>
```

- **Switching** branches **changes** the files in your Working directory
  - ↪ since you change the `HEAD` snapshot...

Tags

```
$> git tag [-s] <name> [-m 'msg']
```

- `-s`: GPG-signed tag, assuming you have configured your signing key

```
$> git config --global user.signingkey 0xDD01D5C1
```

Tags

```
$> git tag [-s] <name> [-m 'msg']
```

- `-s`: GPG-signed tag, assuming you have configured your signing key

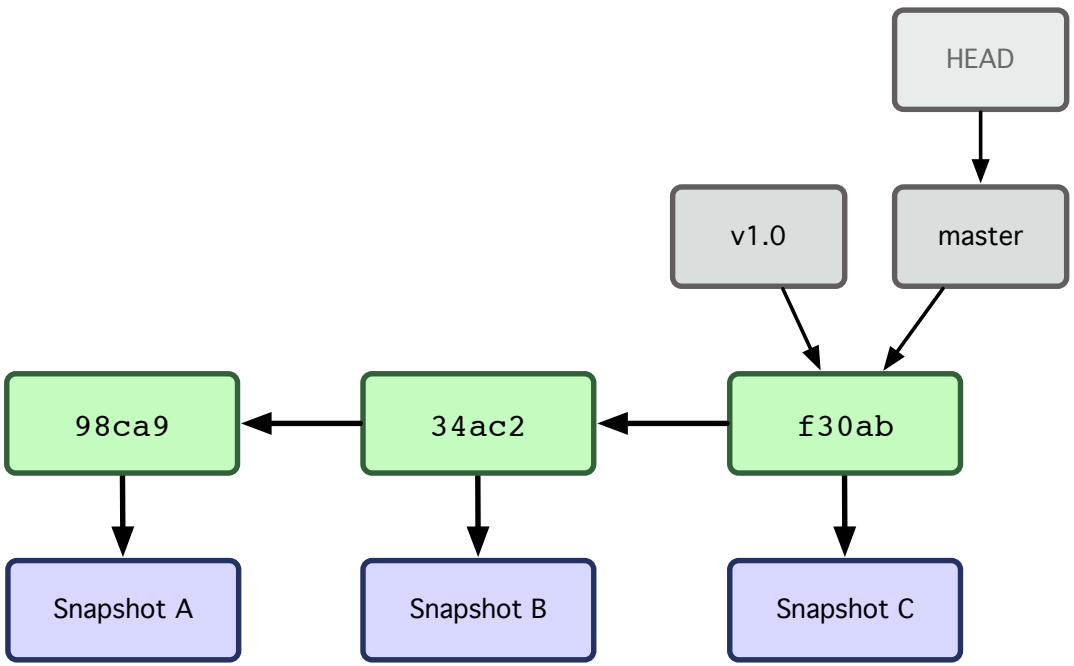
```
$> git config --global user.signingkey 0xDD01D5C1
```

Your Turn!

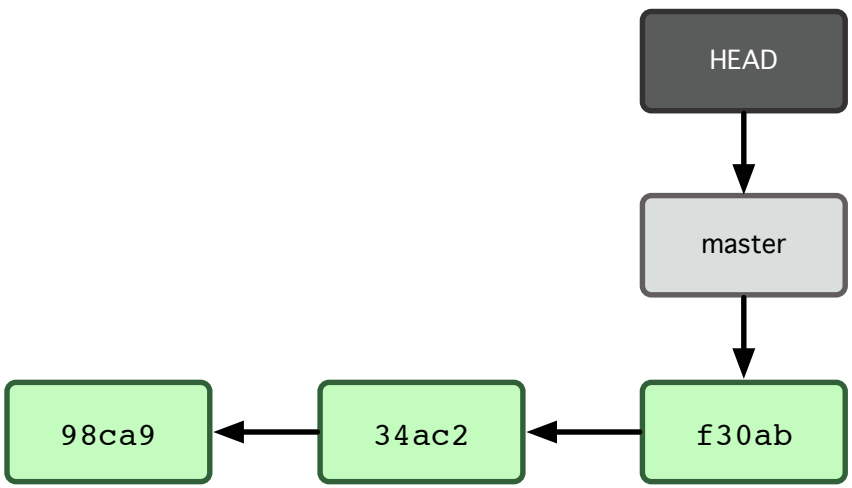
Branch and Tags Hands-on

```
$> cd /tmp/firstproject
$> git branch
* master
$> git tag v1.0 -m 'first tag'
$> git gr
* f31c173 (HEAD -> master, tag: v1.0) a first move with amend
* ee60f53 add README
$> git branch testing
$> git checkout testing   # Move to the 'testing' branch
$> echo 'testing' >> README.md && git commit -a -m "testing 1"
[testing 7afa96d] testing 1
 1 file changed, 1 insertion(+)
$> git checkout master   # return to 'master'
$> echo 'master' >> README.md && git commit -a -m "master"
[master 72d4d5f] master
 1 file changed, 1 insertion(+)
$> git gr
$> gitx   # or gitk
```

Daily Branching Example

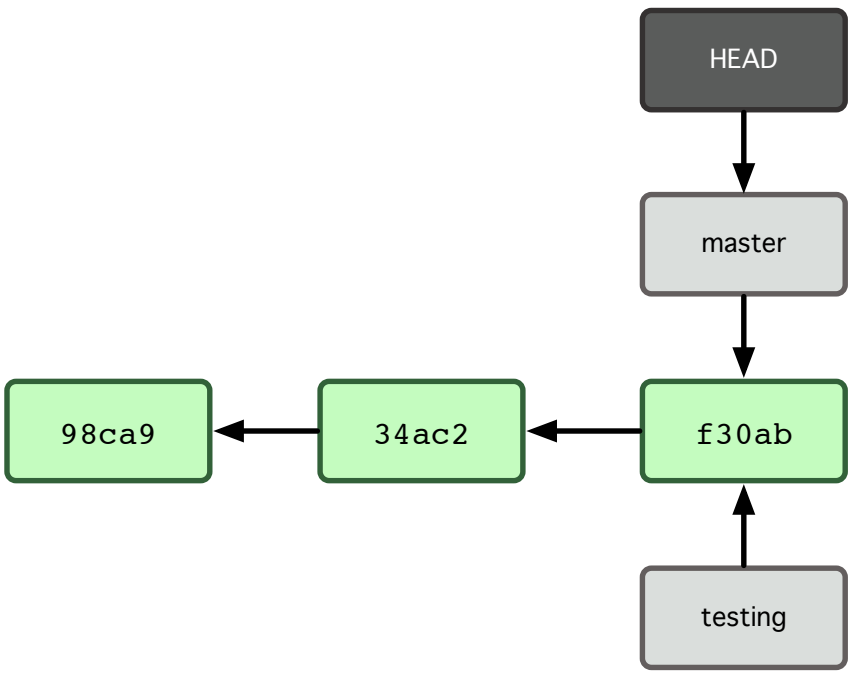


Daily Branching Example



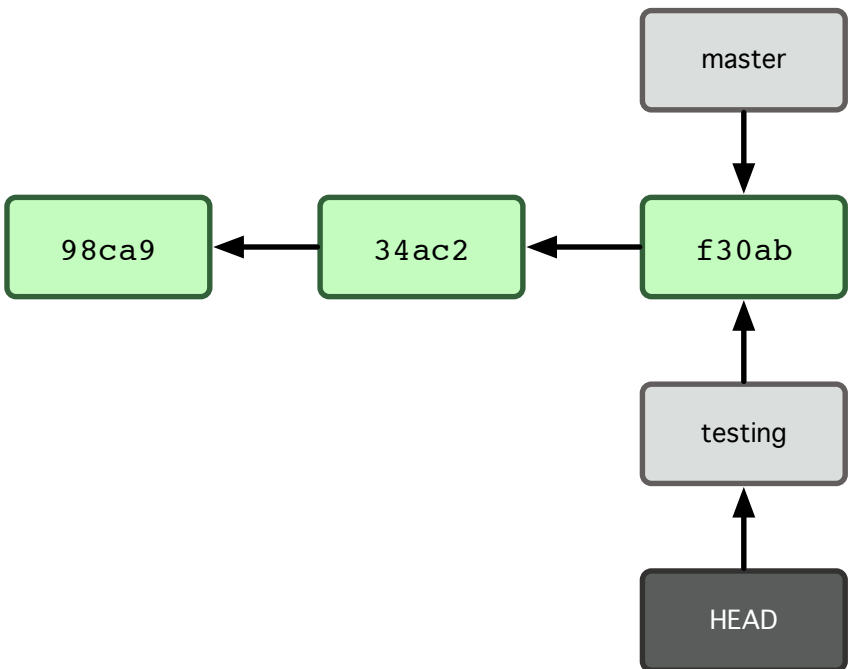
Daily Branching Example

```
(master)$> git branch testing          # create a branch named 'testing'
```



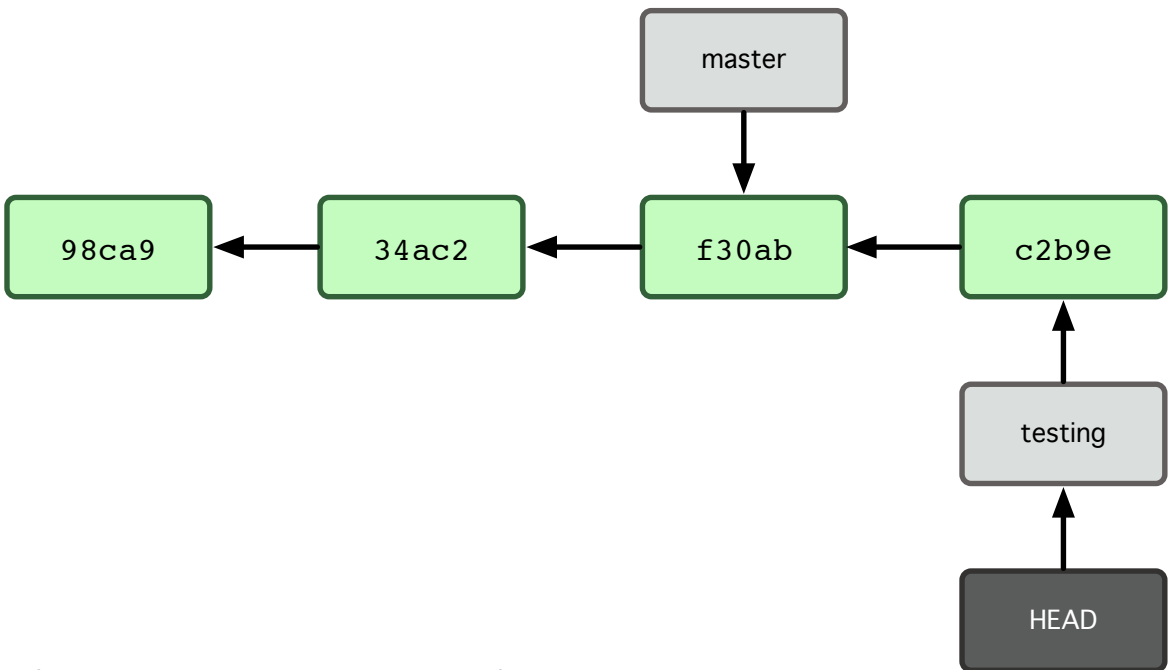
Daily Branching Example

```
(master)$> git checkout testing          # switch to the 'testing' branch
```



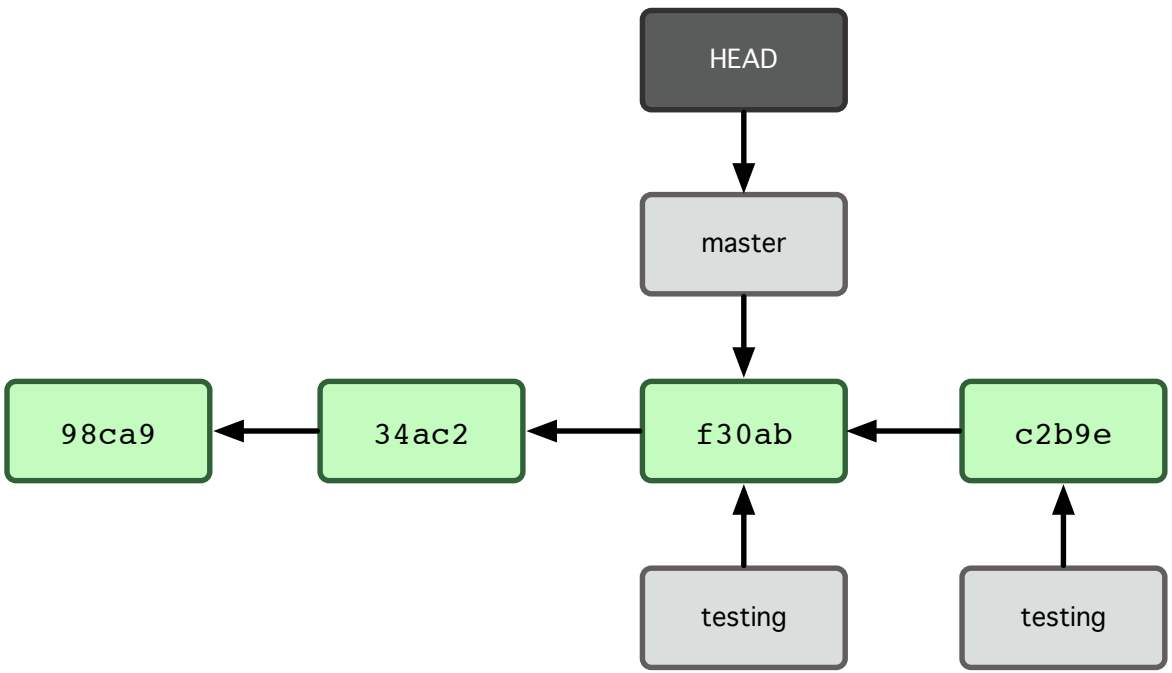
Daily Branching Example

```
(testing)$> vim README.md                          # make some edits...
(testing)$> git commit -a -m "made a change"       # and commit them
```



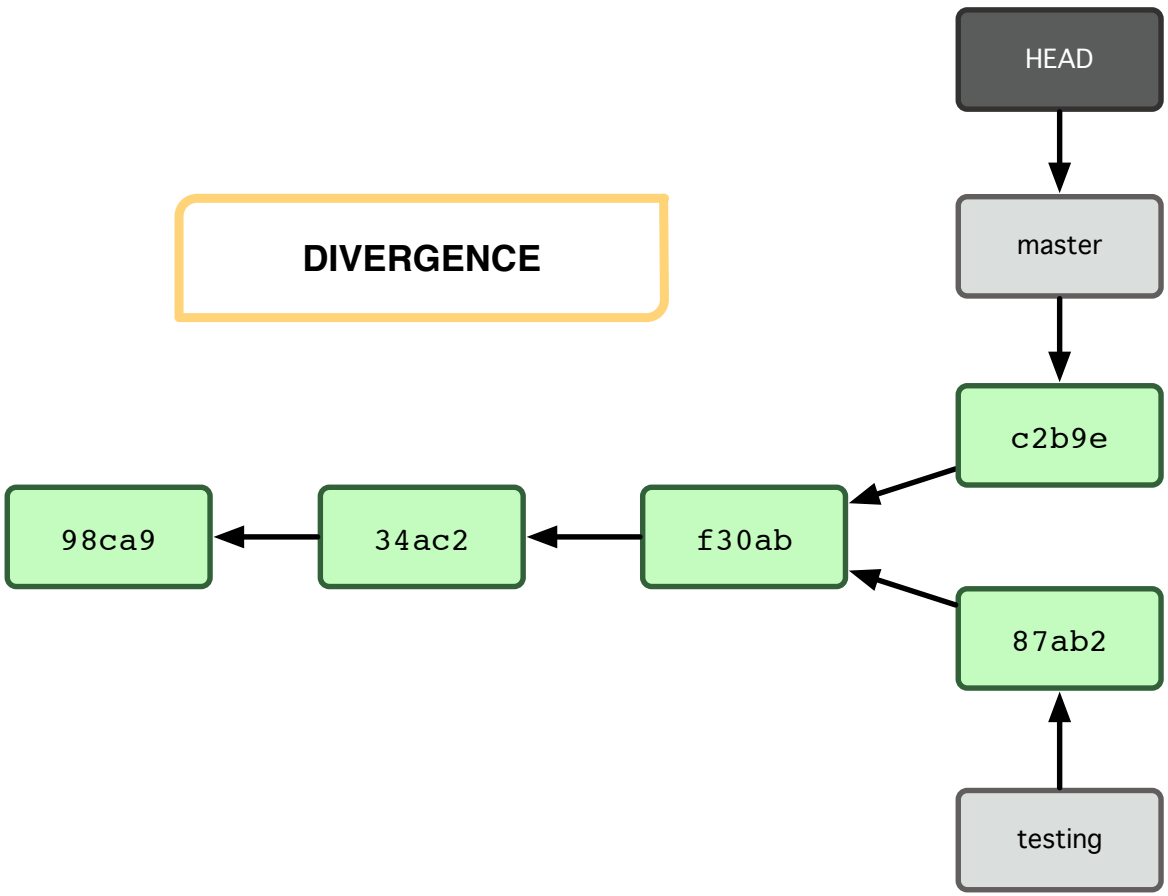
Daily Branching Example

```
(testing)$> git checkout master          # switch back to 'master' branch
```



Daily Branching Example

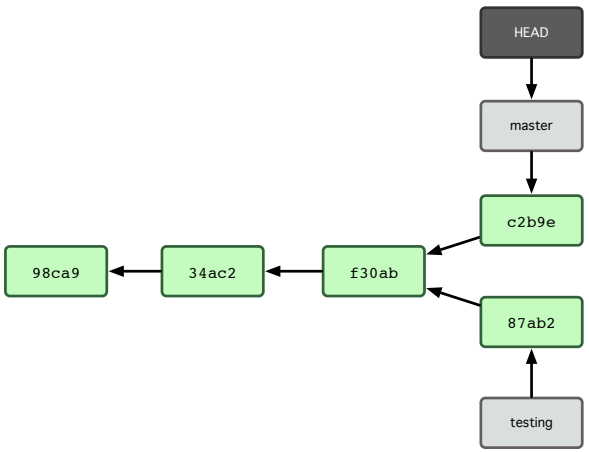
```
(testing)$> vim README.md                    # make some edits
(testing)$> git commit -a -m "intro"         # introduce divergence!
```



Diverging / Converging (Fork-Join)

Diverging

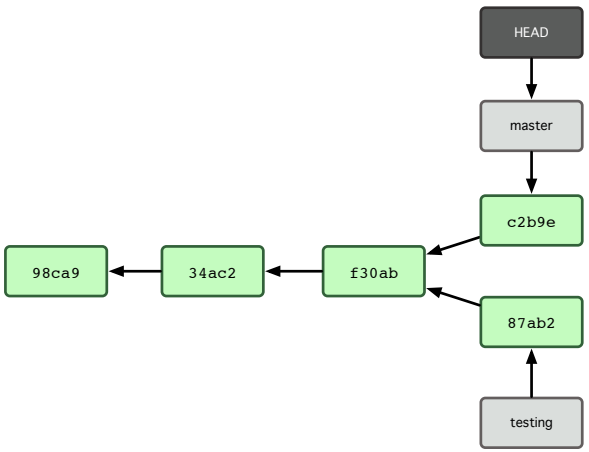
- Changes are committed into two branches independently
  - ↪ Then the branches **diverge**



Diverging / Converging (Fork-Join)

Diverging

- Changes are committed into two branches independently
  - ↪ Then the branches **diverge**



Converging to join branches

1. `merge` (if possible in **fast-forward** mode)
2. `rebase`

Merging

```
$> git merge [--no-ff] <branch>
```

- Different auto-merge strategies
  - ↪ **fast-forward** (if possible)
  - ↪ **3-ways** (regular)
- Usually painless ;)

Merging

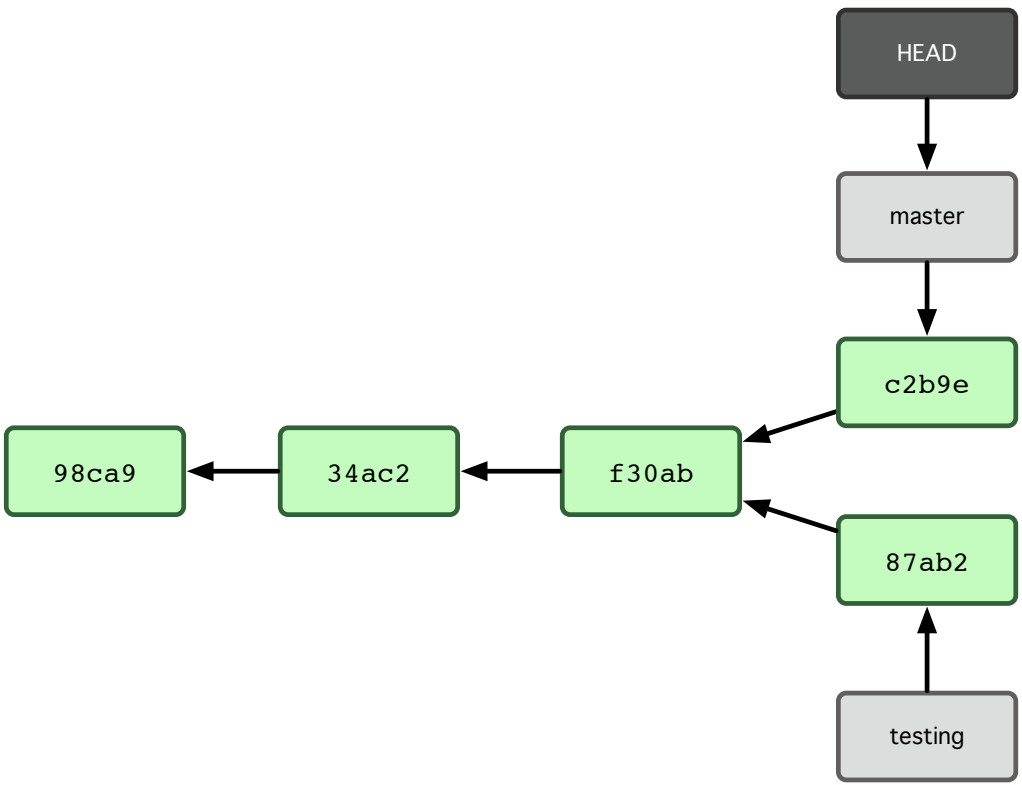
```
$> git merge [--no-ff] <branch>
```

- Different auto-merge strategies
  - ↪ **fast-forward** (if possible)
  - ↪ **3-ways** (regular)
- Usually painless ;)

```
<<<<<<< HEAD
master
=======
testing
>>>>>>> testing
```

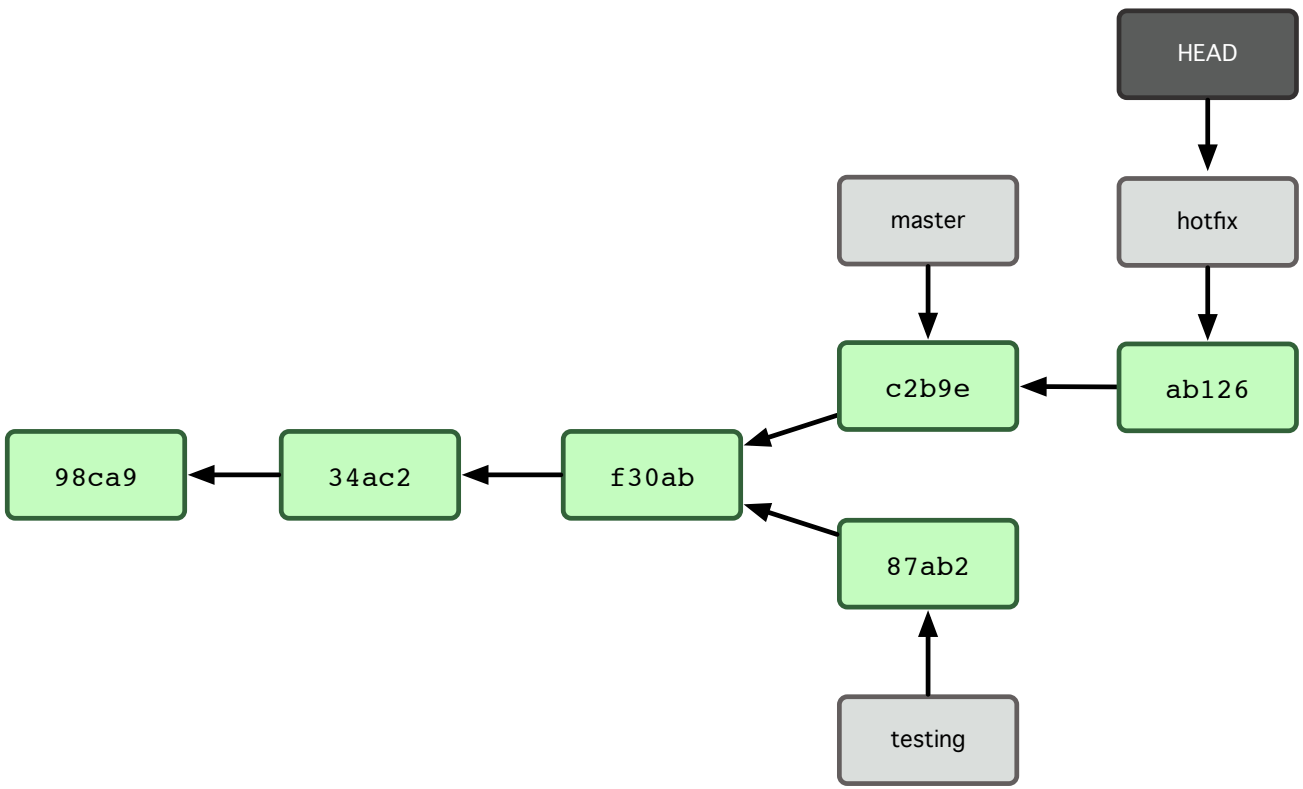
- **In case of conflicts**:
  - ↪ Resolve the conflicts manually — `vim` / `emacs` / `subl` ...
    - ✓ check for the sequence <<< in the text
  - ↪ then mark as resolved — `git add <file>`
  - ↪ and trigger the merge commit — `git commit`

Daily Branching Example



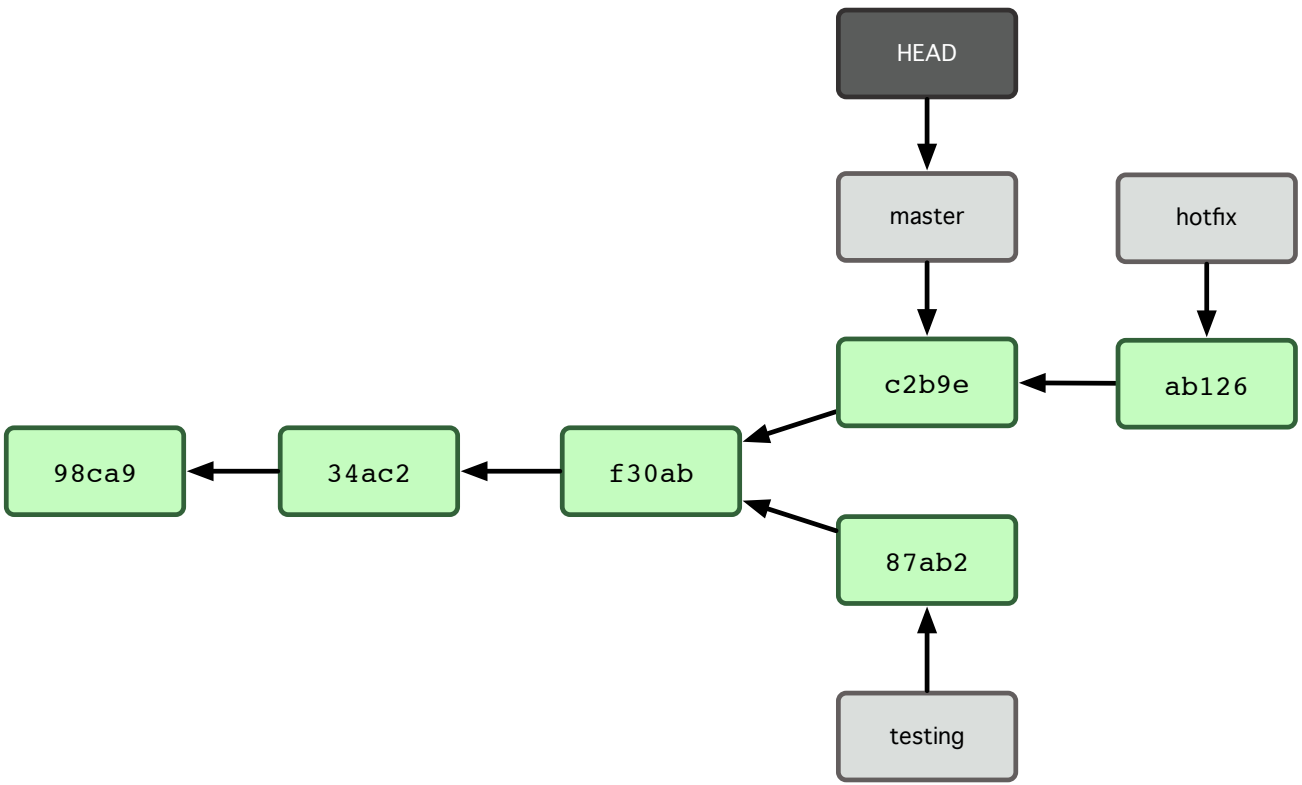
Daily Branching Example

```
(master)$> git checkout -b hotfix        # create and switch to 'hotfix'
(hotfix)$> vim test.rb                      # make some edits...
(hotfix)$> git commit -a -m "hotfix"           # and commit them
```



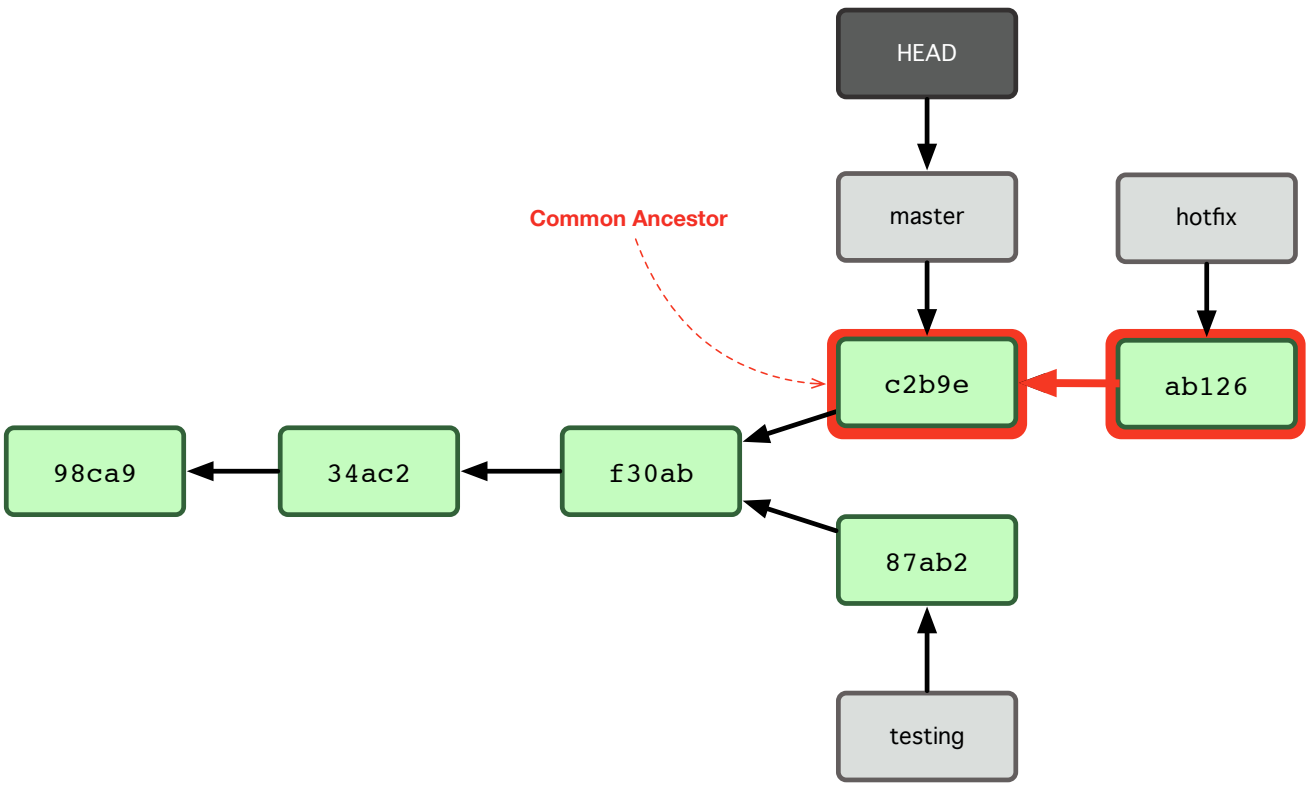
Daily Branching Example

```
(hotfix)$> git checkout master          # swicth back to 'master' branch
```



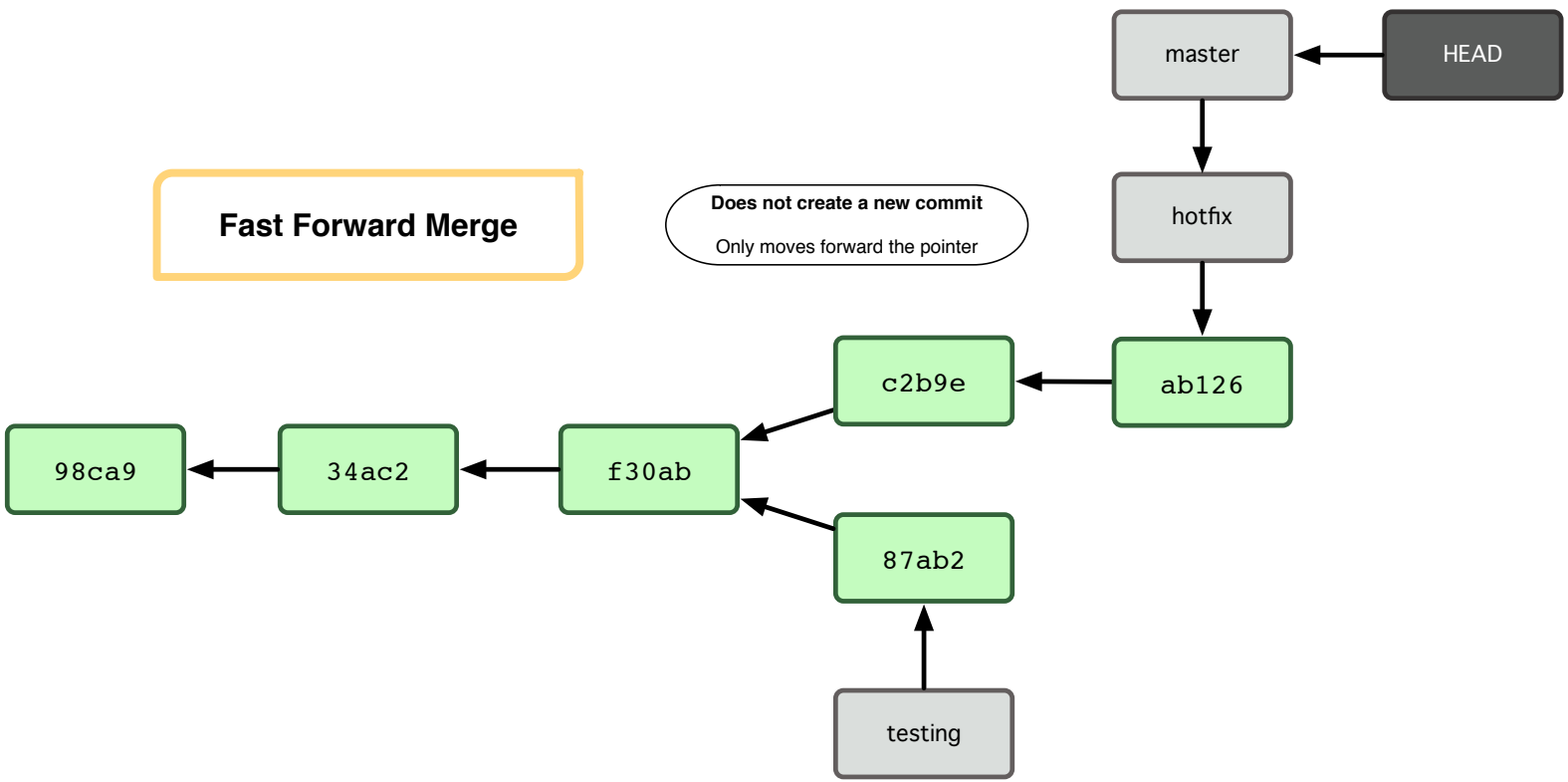
Daily Branching Example

```
(master)$> git merge hotfix   # merge the 'hotfix' branch (fast-forward)
```

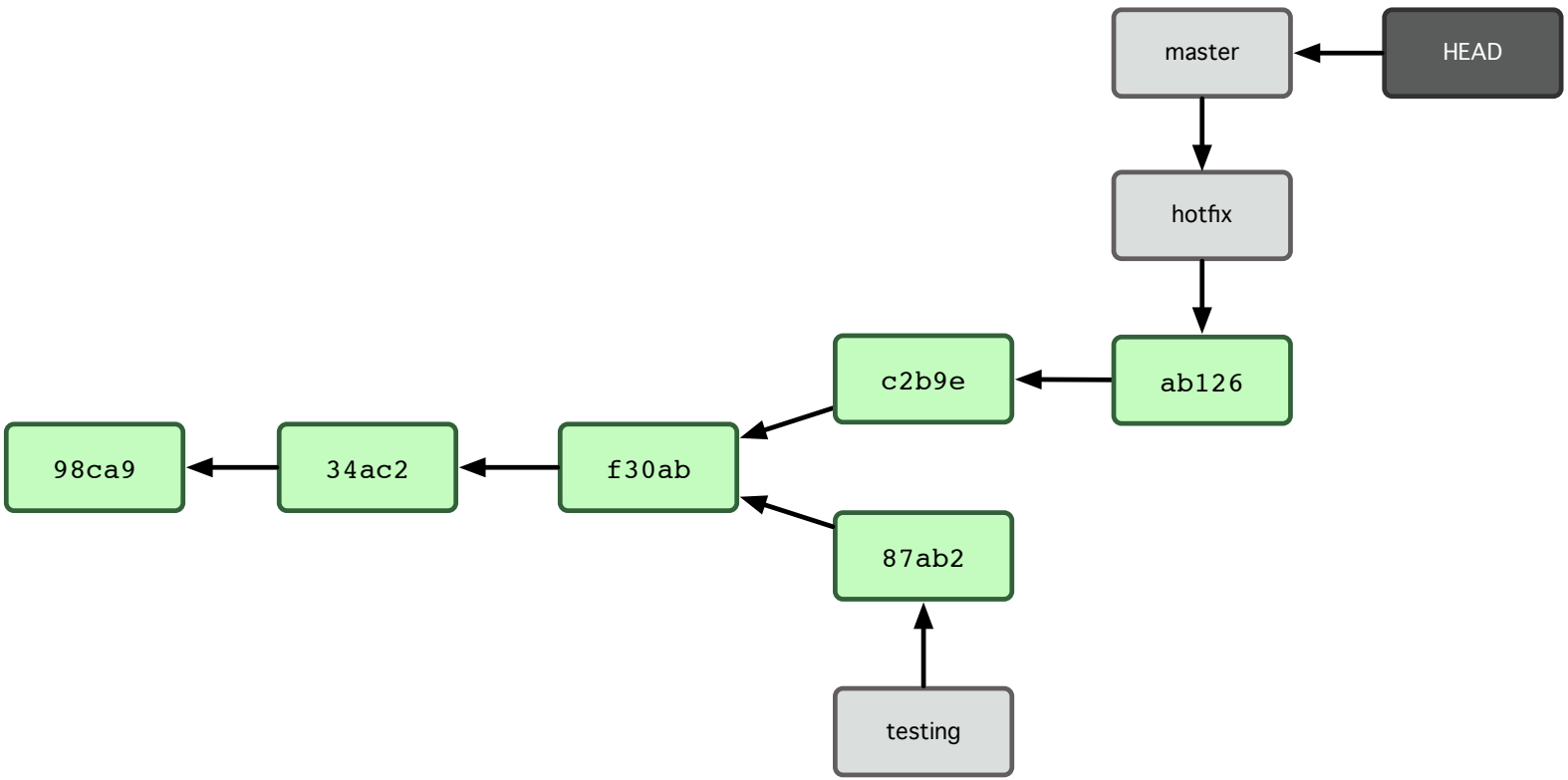


Daily Branching Example

```
(master)$> git merge hotfix    # merge the 'hotfix' branch (fast-forward)
```

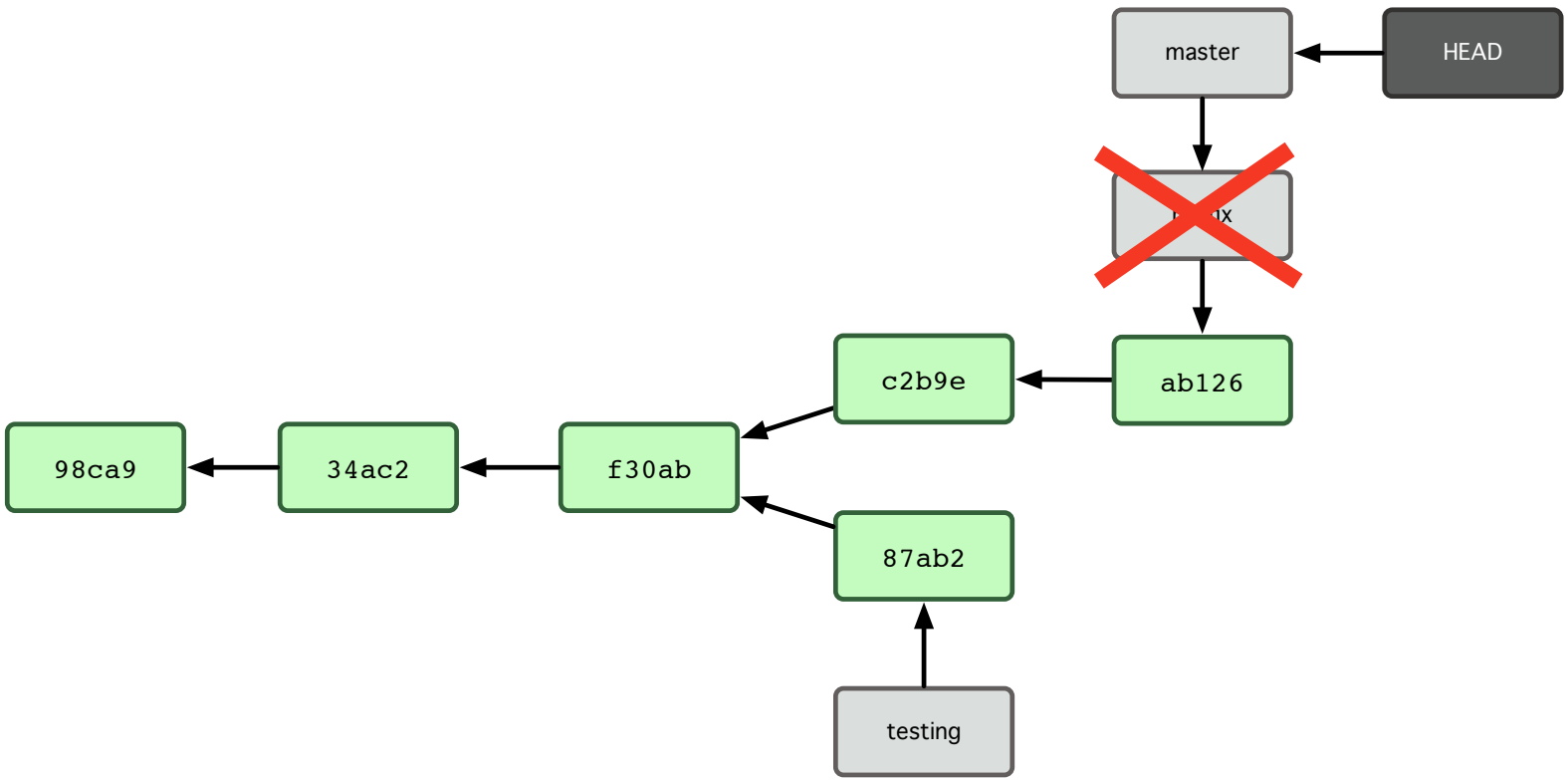


Daily Branching Example



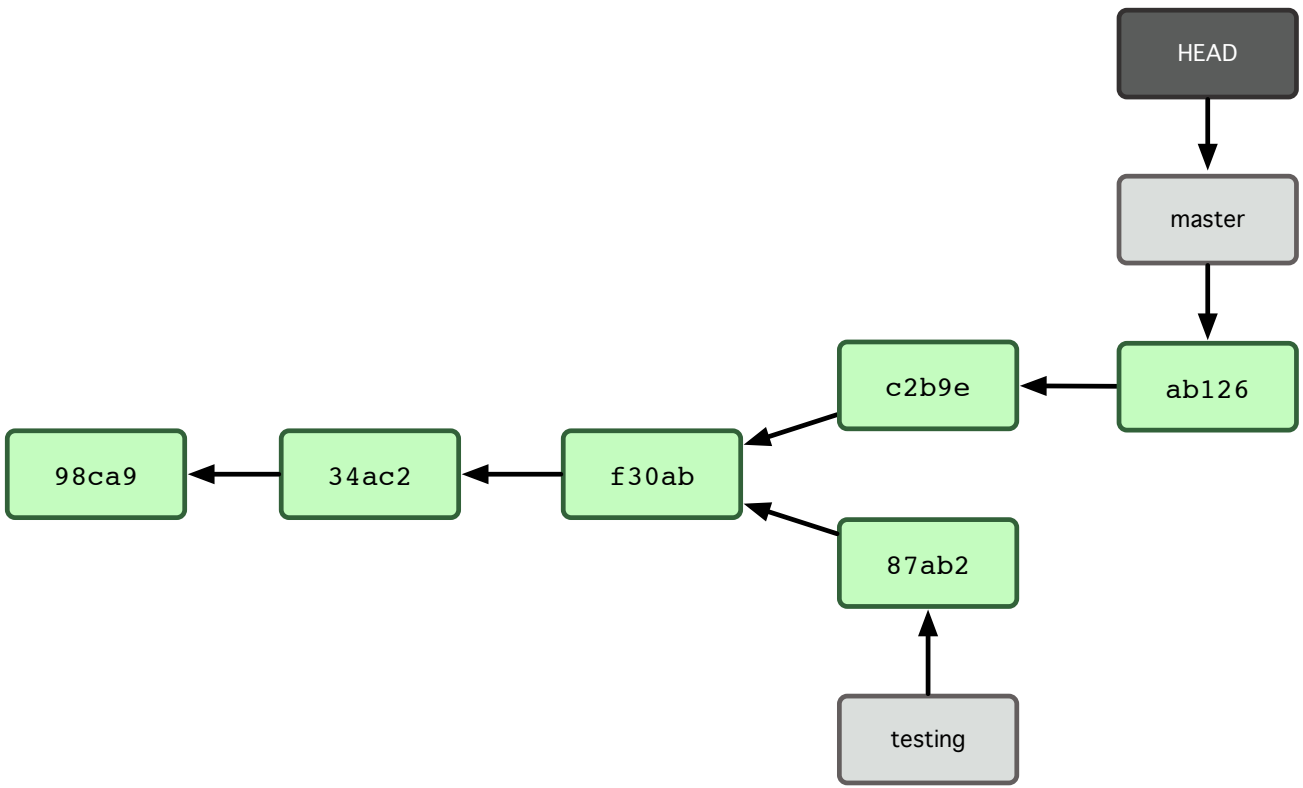
Daily Branching Example

```
(master)$> git branch -d hotfix    # delete the (useless) 'hotfix' branch
```

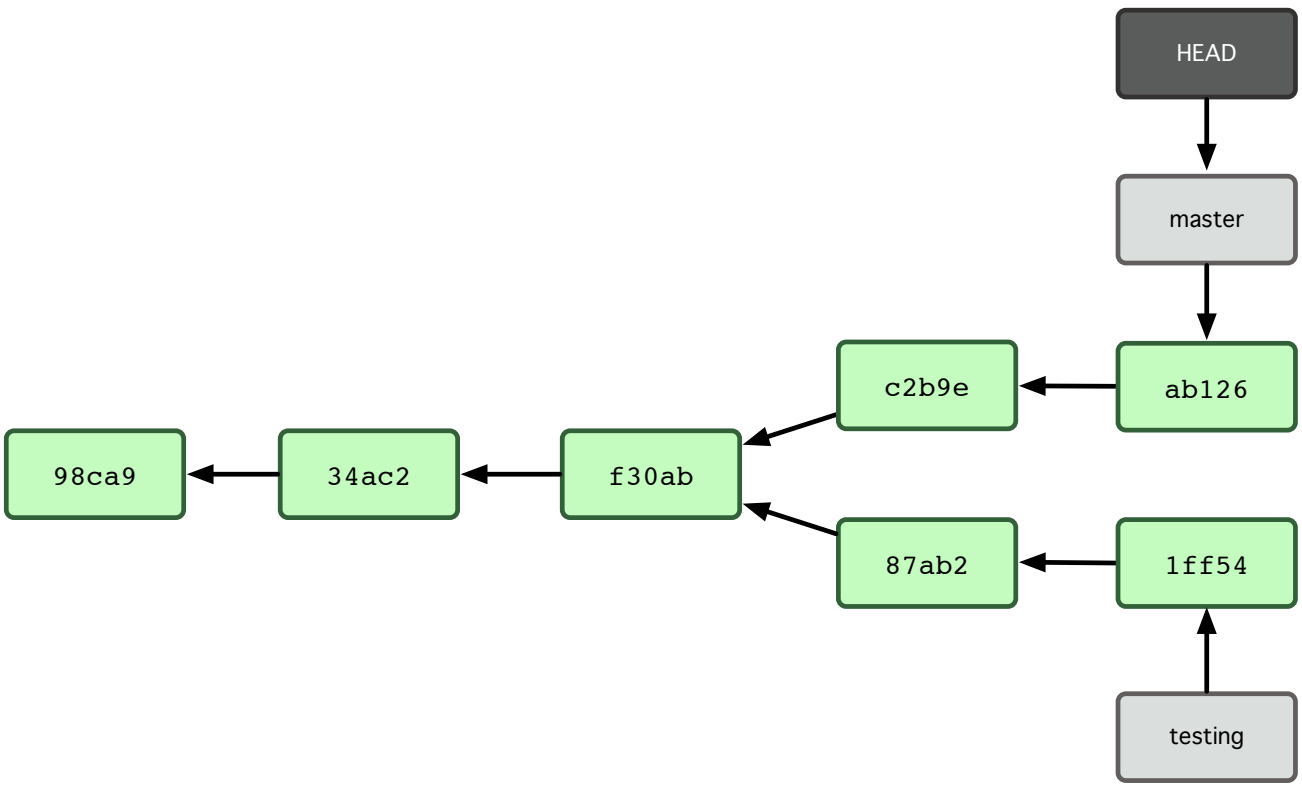


Daily Branching Example

```
(master)$> git branch -d hotfix    # delete the (useless) 'hotfix' branch
```

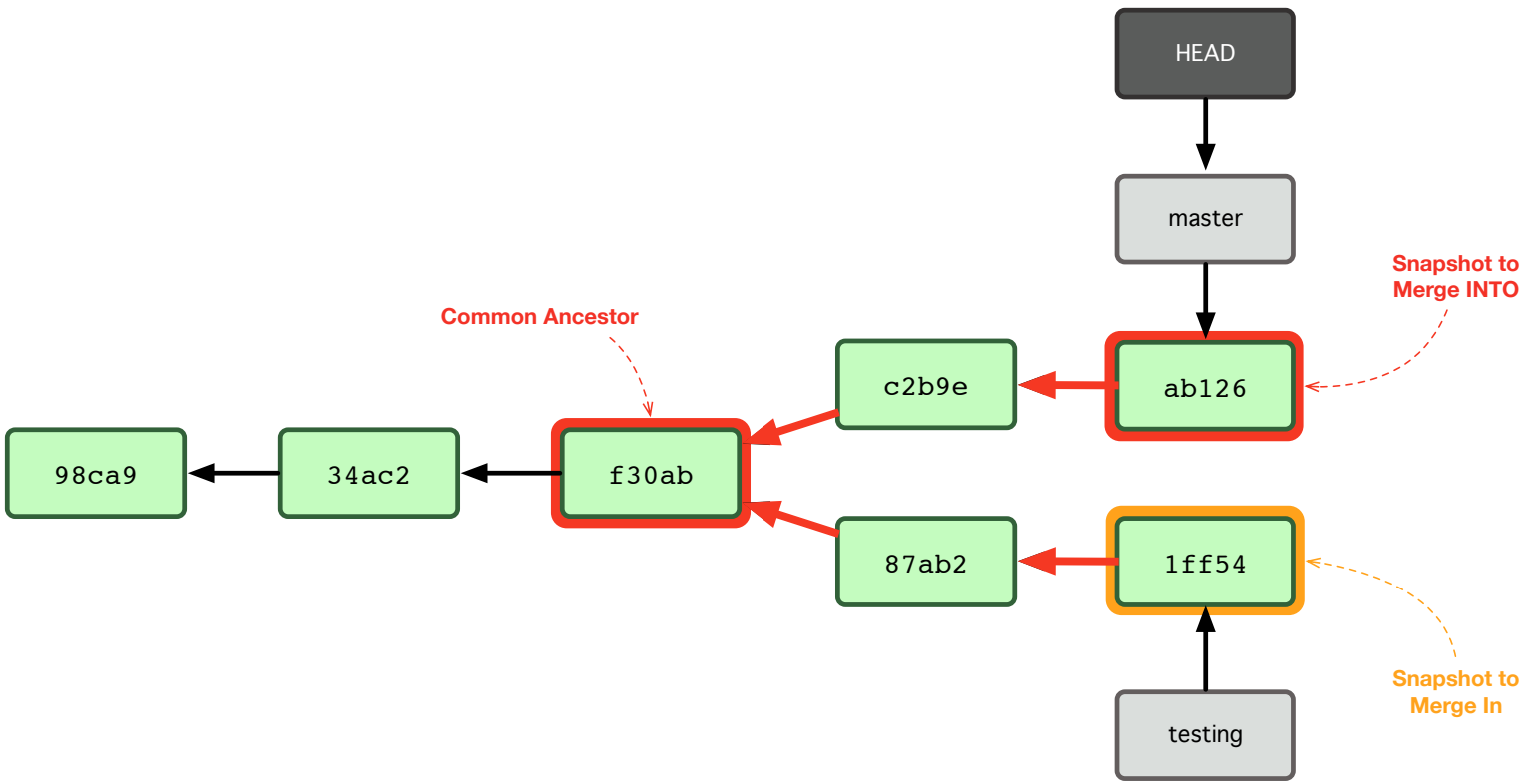


Daily Branching Example



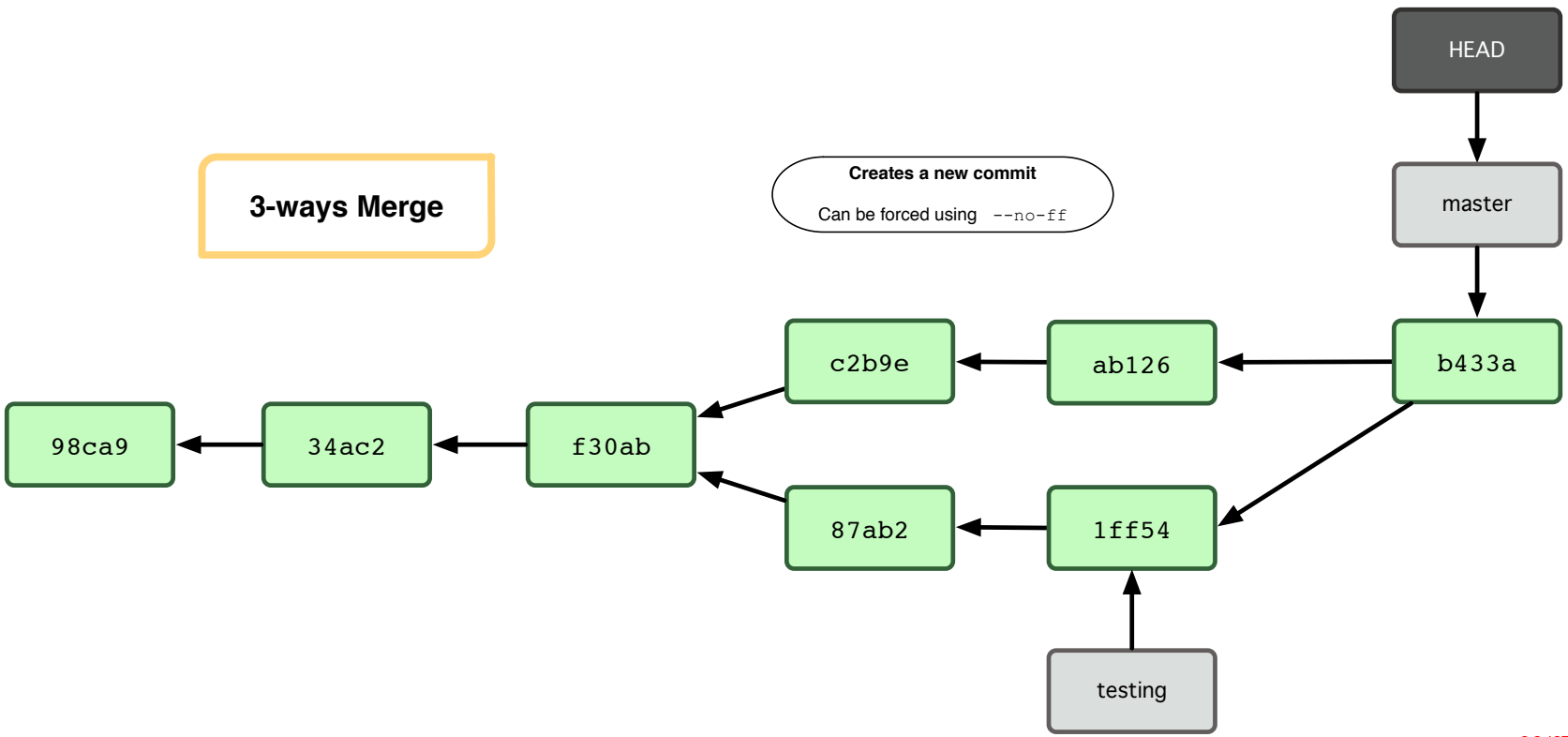
Daily Branching Example

```
(master)$> git merge testing        # merge the 'testing' branch (3-ways)
```



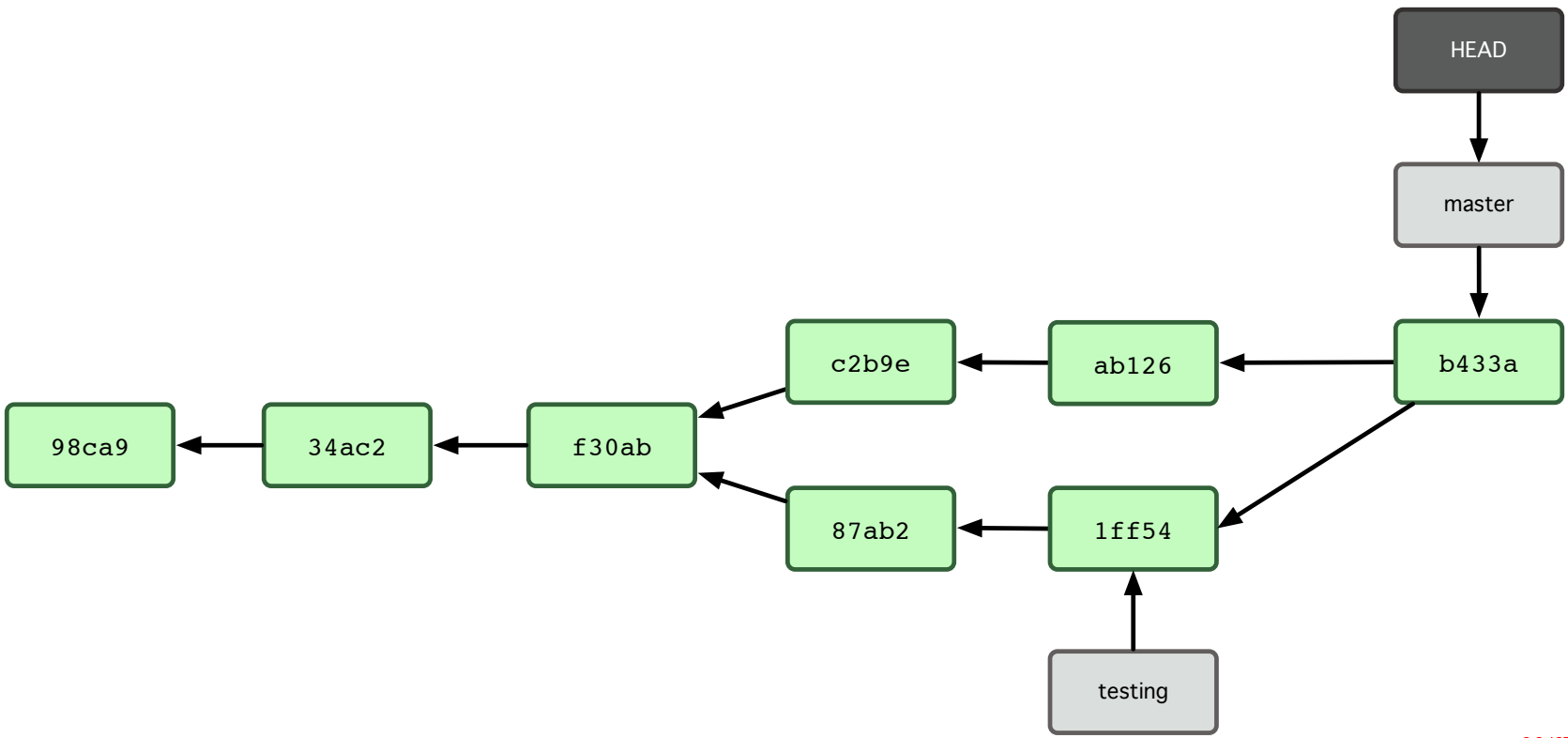
Daily Branching Example

```
(master)$> git merge testing        # merge the 'testing' branch (3-ways)
```



Daily Branching Example

```
(master)$> git merge testing        # merge the 'testing' branch (3-ways)
```



Merging and solving conflicts Hands-on

Your Turn!

Merging and solving conflicts Hands-on

```
$> cd /tmp/firstproject
$> git checkout master
Switched to branch 'master'
$> git checkout -b hotfix
Switched to a new branch 'hotfix'
$> touch test.rb && git add test.rb && git commit -m "hotfix"
[hotfix ac188bd] hotfix
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 test.rb
$> git checkout master
Switched to branch 'master'
$> git gr
* 72d4d5f (HEAD -> master) master
* f31c173 (tag: v1.0) a first move with amend
* ee60f53 add README
```

Merging and solving conflicts Hands-on

- Fast-Forward Merge

```
$> git merge hotfix
Updating 72d4d5f..ac188bd
Fast-forward
 test.rb | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 test.rb
$> git gr
* ac188bd (HEAD -> master, hotfix) hotfix
* 72d4d5f master
* f31c173 (tag: v1.0) a first move with amend
* ee60f53 add README

$> git branch -d hotfix
Deleted branch hotfix (was ac188bd)
```

Merging and solving conflicts Hands-on

- Solving conflicts

```
$> git merge testing
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
$> cat README.md
<<<<<<< HEAD
master
=======
testing
>>>>>>> testing
$> vim README.md    # Edit to solve the conflicts
$> cat README.md
master corrected
```

Merging and solving conflicts Hands-on

- Solving conflicts

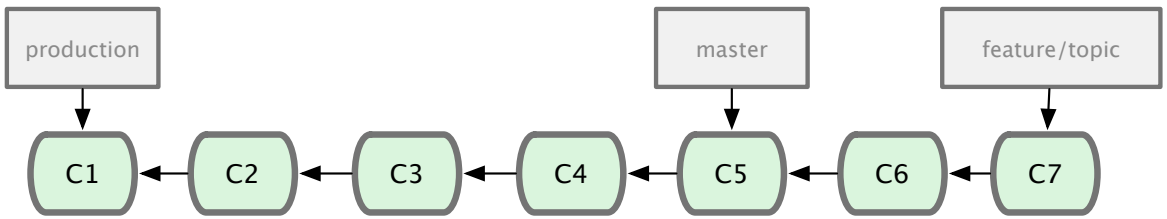
```
$> git status     # OR git st
On branch master
You have unmerged paths.

Unmerged paths:
    both modified:   README.md

no changes added to commit
```

Merging and solving conflicts Hands-on

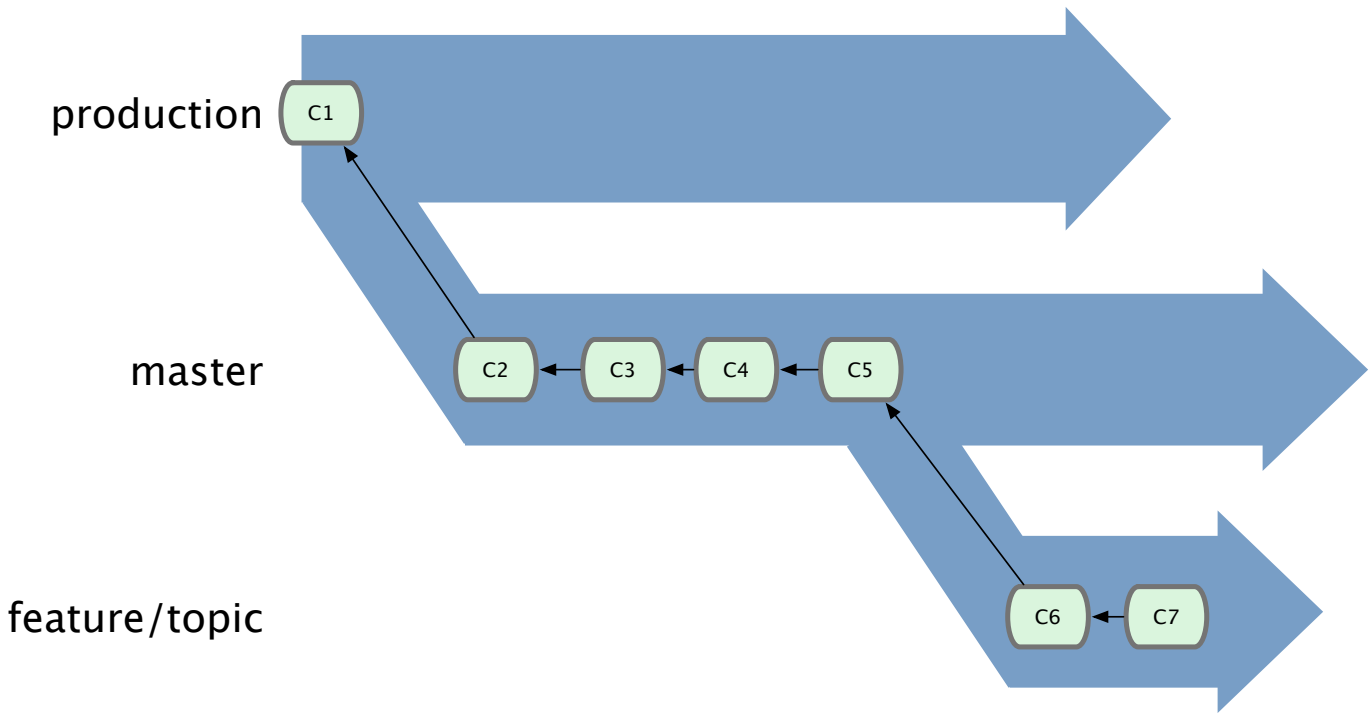
- Solving conflicts and 3-way merge

```
$> git add README.md     # Mark as corrected / conflict solved
$> git commit
Recorded resolution for 'README.md'.
[master ef299b7] Merge branch 'testing'
$> git gr
*   ef299b7 (HEAD -> master) Merge branch 'testing'
|\
| * 7afa96d (testing) testing 1
* | ac188bd hotfix
* | 72d4d5f master
|/
* f31c173 (tag: v1.0) a first move with amend
* ee60f53 add README
```

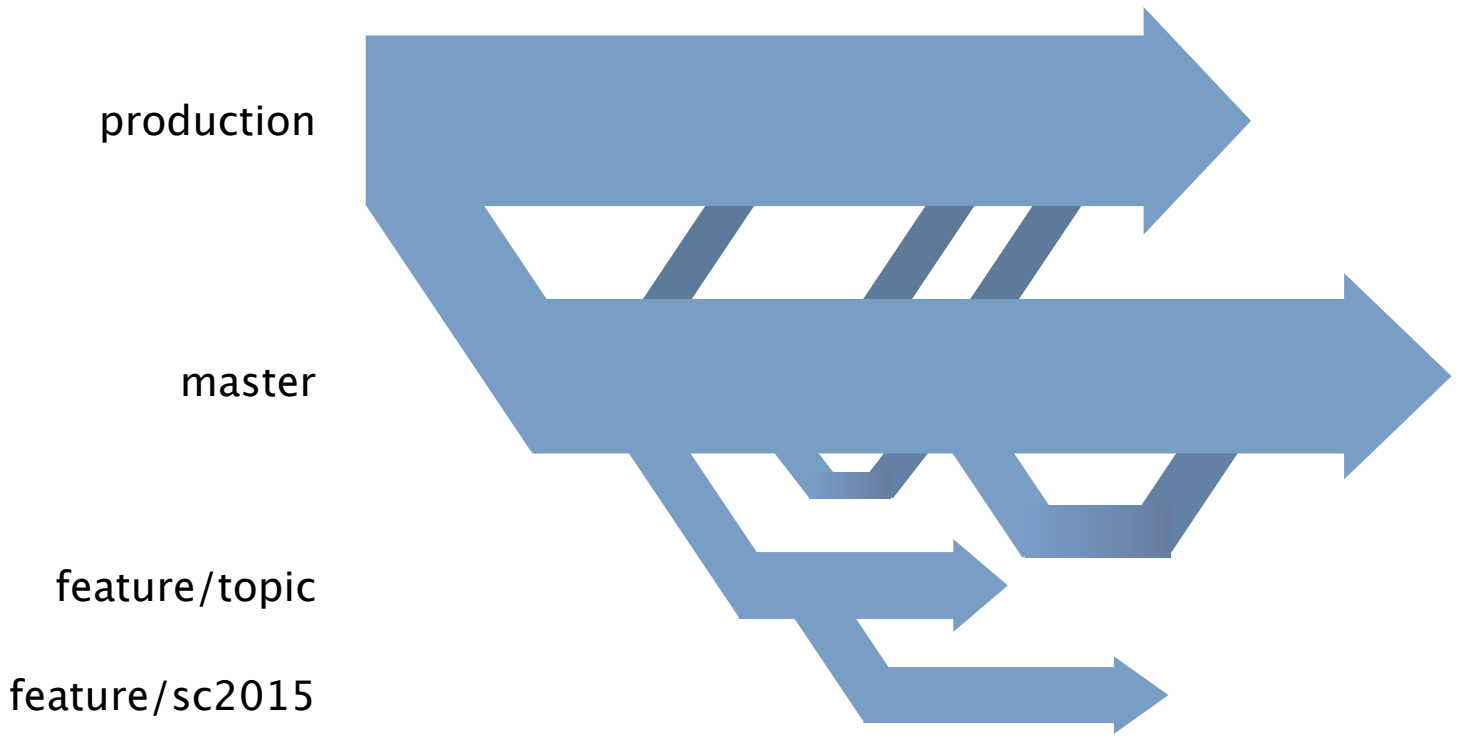
Branching Workflow



Branching Workflow



Branching Workflow



Git-flow to the rescue

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

```
$> git flow init
```

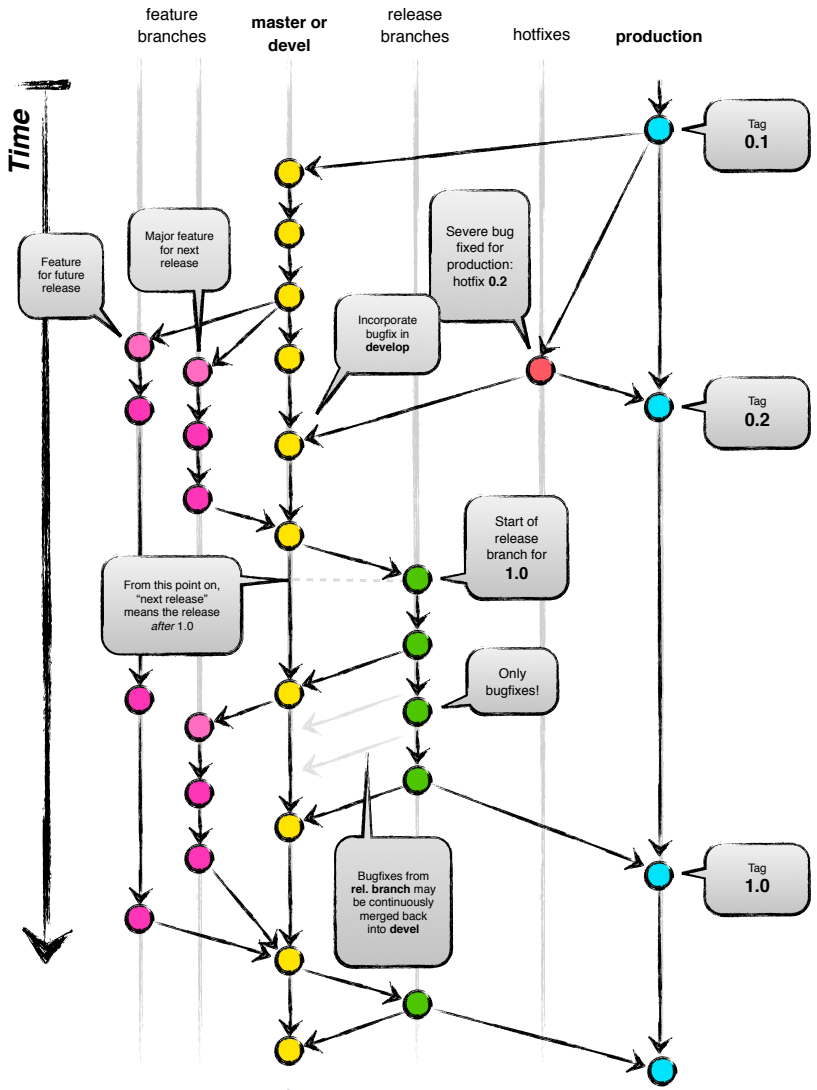
```
$> git flow feature { start, publish, finish } <name>
```

```
$> git flow release { start, publish, finish } <version>
```

- Ensure two long running branches
  - ↪ `production` : the stable branch
    - ✓ ideally holding **only** tags of the successive releases
  - ↪ `master` / `devel`: the **main** branch where the developments occurs
- On demand: make a new feature branch `feature/<name>`
- From time to time, release your code into `production` and tag

Git-flow Illustrated

[Source : Nvie]



Git-flow Setup using FalkorLib

- Initiate a Git-flow-ready repository using FalkorLib
  - ↪ [Personnal] Ruby Library offering the `falkor` binary

```
$> falkor new repo [--rake]     # setup the current directory
```

- The repository is fed with a root `Makefile` (or `Rakefile`)
  - ↪ facilitate repository setup upon cloning

```
$> git clone <url> && cd <cloned_dir>
$> make setup
```

  - ↪ project releasing using Git-flow made easy

```
$> make start_bump_{major,minor,patch}     # bump version with git-flow
$> make release
```

Summary

Working Together

- Sign-up on Github
  - ↪ `http://github.com`
  - ↪ Best place to share **public** repository

Working Together

- Sign-up on Github
  - ↪ `http://github.com`
  - ↪ Best place to share **public** repository

- Alternative for **private** projects:
  - ↪ Gforge @ UL `https://gforge.uni.lu/`
  - ↪ (incoming) Gitlab @ UL `https://gitlab.uni.lu/`

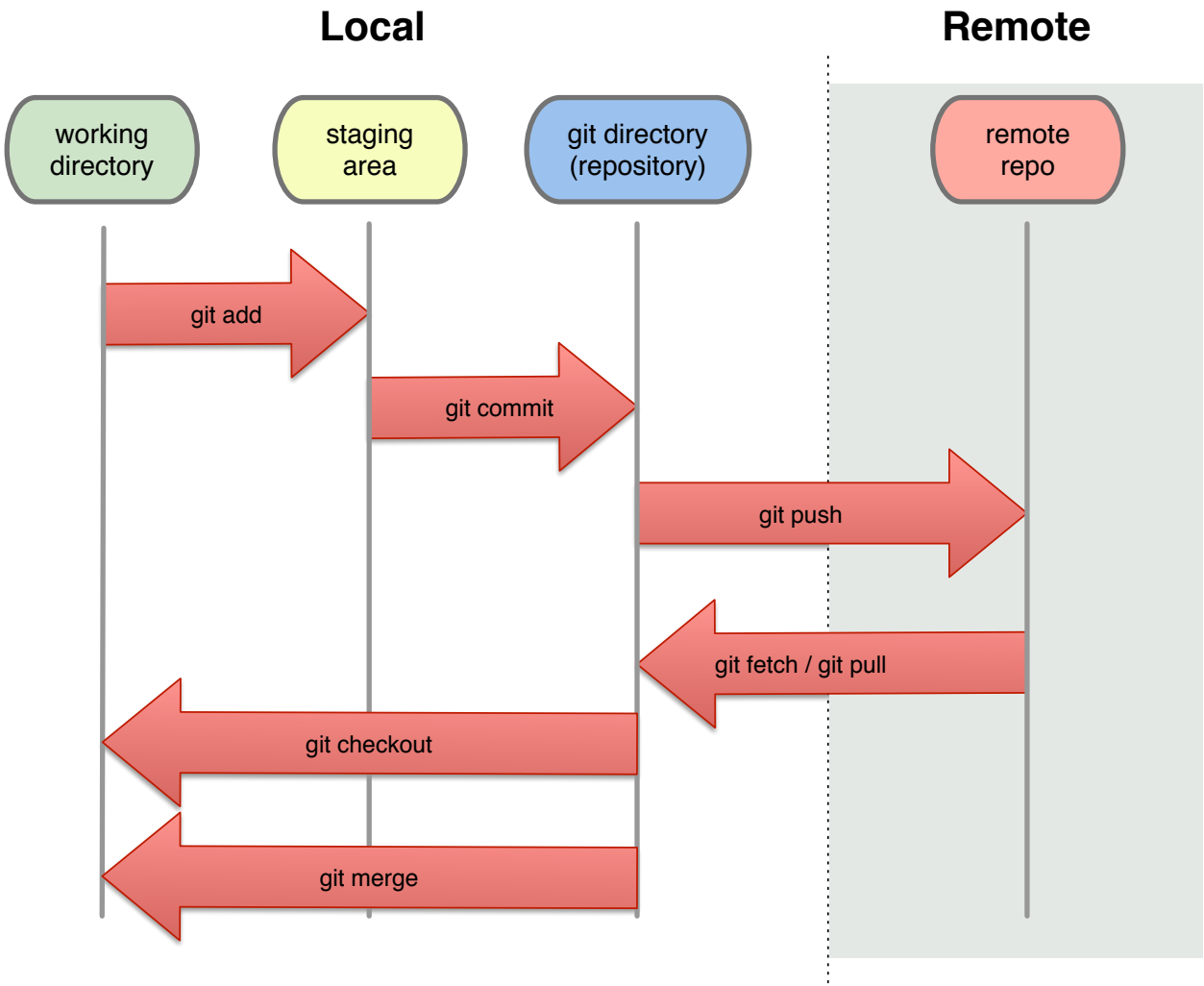
Working Together

- Sign-up on Github
  - ↪ `http://github.com`
  - ↪ Best place to share **public** repository

- Alternative for **private** projects:
  - ↪ Gforge @ UL `https://gforge.uni.lu/`
  - ↪ (incoming) Gitlab @ UL `https://gitlab.uni.lu/`

- Setup your own Git server: gitolite `https://github.com/sitaramc/gitolite`
  - ↪ Management through the `gitolite-admins` Git repository (!)
  - ↪ A single user (`git`) to interact with all repositories
    - ✓ map users though their (multiple) SSH keys
    - ✓ fine-grained access control

Working with remotes



Remotes

```
$> git remote [-v]
```

- Other clones of the same repository
  - ↪ Can be local (another checkout) or remote (coworker, central server)
  - ↪ default remotes for push and pull actions: `origin`
    - ✓ `origin` is set upon clone

Remotes

```
$> git remote [-v]
```

- Other clones of the same repository
  - ↪ Can be local (another checkout) or remote (coworker, central server)
  - ↪ default remotes for push and pull actions: `origin`
    - ✓ `origin` is set upon clone

Your Turn!

```
$> cd /tmp/tutorials
$> git remote
origin
$> git remote -v
origin  https://github.com/ULHPC/tutorials.git (fetch)
origin  https://github.com/ULHPC/tutorials.git (push)
```

Adding Remotes

```
$> git remote add <name> <url>
```

Adding Remotes

```
$> git remote add <name> <url>
```

Your Turn!

- Fork the ULHPC/tutorials (as <yourlogin>/tutorials)
- Clone and add the upstream remote to the original repository

```
$> git clone https://github.com/<yourlogin>/tutorials.git /tmp/fork
$> cd /tmp/fork
$> git remote add upstream  https://github.com/ULHPC/tutorials.git
$> git remote -v
origin      https://github.com/<yourlogin>/tutorials.git (fetch)
origin      https://github.com/<yourlogin>/tutorials.git (push)
upstream    https://github.com/ULHPC/tutorials.git (fetch)
upstream    https://github.com/ULHPC/tutorials.git (push)
```

Removing Remotes

```
$> git remote rm <name>
```

Removing Remotes

```
$> git remote rm <name>
```

Your Turn!

```
$> cd /tmp/fork
$> git remote
origin
upstream
$> git remove rm upstream
```

Remote Branches

- Branches on remotes are represented locally as: `<remote>/<branch>`
  - ↪ **Ex**: `origin/master`

Remote Branches

- Branches on remotes are represented locally as: `<remote>/<branch>`
  - ↪ **Ex**: `origin/master`

Tracking Remote Branches

- You can track a remote branch `<remote>/<branch>`
  - ↪ assuming you have previously `fetch` the remote `origin`
  - ↪ creates the local branch `<branch>`

```
$> git branch --track <branch> origin/<branch>
```

Tracking Remote Branches

```
$> git branch --track <branch> origin/<branch>
```

Your Turn!

```
$> cd /tmp/tutorials
$> git branch -a
* devel
  remotes/origin/HEAD -> origin/devel
  remotes/origin/devel
  remotes/origin/production
$> git branch --track production origin/production
Branch production set up to track remote branch production from
origin.
$> git branch
* devel
  production
```

Pushing to your remote

```
$> git push [<remote>]
```

- Transfer local commits of the **current** branch to a remote.
  - ↪ push to `origin` by default, assuming the current branch is tracked

Pushing to your remote

```
$> git push [<remote>]
```

- Transfer local commits of the **current** branch to a remote.
  - ↪ push to `origin` by default, assuming the current branch is tracked

Your Turn!

```
$> cd /tmp/fork
$> git remote
origin
upstream
$> touch new-file
$> git add new-file
$> git commit -m "add"
```

```
$> git push
Counting objects: 10, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (10/10), done.
Total 10 (delta 4), reused 0 (delta 0)
To git@github.com:<yourlogin>/documents.git
   671eb88..c798919  devel -> devel
```

Pulling from remotes

```
$> git pull [--rebase] [<remote>]        # --rebase = DANGER!
```

- fetch all commits from the remote **and** merge (or rebase)
  - ↪ allows for easy to use, equivalent to the **advanced** alternative:

```
$> git fetch [<remote>]
$> git merge <remote>/<branch>     # 'git rebase' if --rebase
```

  - ↪ `fetch`: allows for inspection and manual merging of remote changes

Pulling from remotes

```
$> git pull [--rebase] [<remote>]        # --rebase = DANGER!
```

- fetch all commits from the remote **and** merge (or rebase)
  - ↪ allows for easy to use, equivalent to the **advanced** alternative:

```
$> git fetch [<remote>]
$> git merge <remote>/<branch>     # 'git rebase' if --rebase
```

  - ↪ `fetch`: allows for inspection and manual merging of remote changes

Your Turn!

```
$> cd /tmp/tutorials
$> git pull          # OR git up
Updating ae97dae..06576e0
[...]
2 files changed, 4 insertions(+), 5 deletions(-)
```

Publish a (local) branch on a remote

```
$> git push -u origin <branch>
```

```
$> git flow feature publish <name>
```

Publish a (local) branch on a remote

```
$> git push -u origin <branch>
```

```
$> git flow feature publish <name>
```

- If you want to **delete** a **remote** branch

```
$> git push origin --delete <branch>                    # DANGER!
```

Publish a (local) branch on a remote

```
$> git push -u origin <branch>
```

```
$> git flow feature publish <name>
```

- If you want to **delete** a **remote** branch

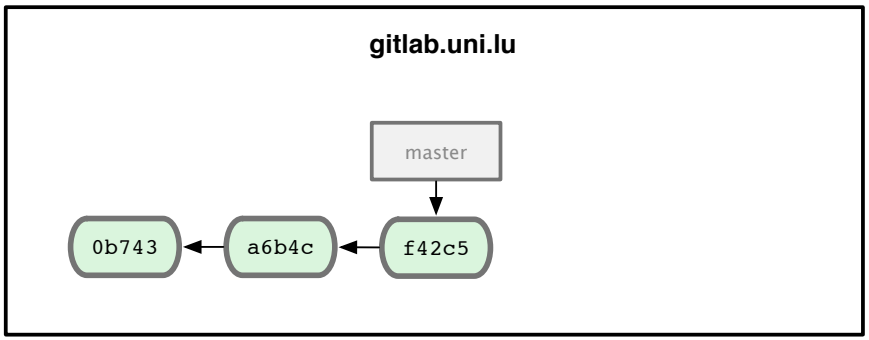
```
$> git push origin --delete <branch>                    # DANGER!
```

Your Turn!

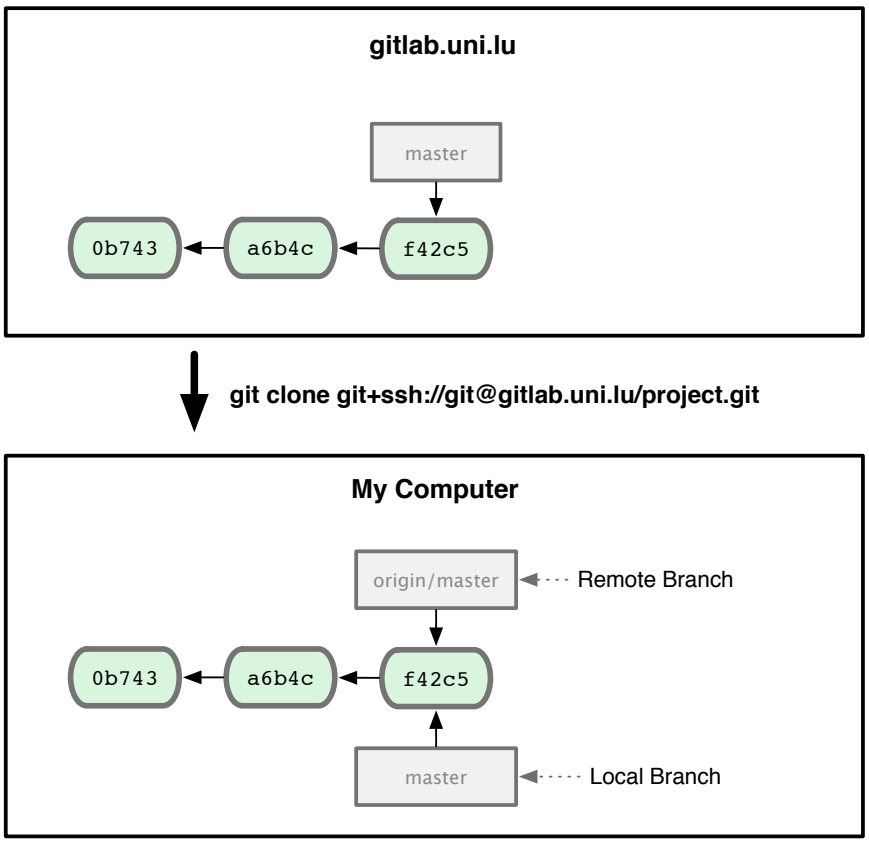
```
$> cd /tmp/fork
$> git flow feature start toto
$> git flow feature publish toto
```

```
$> git branch -a
$> git push origin --delete \
      feature/toto
```

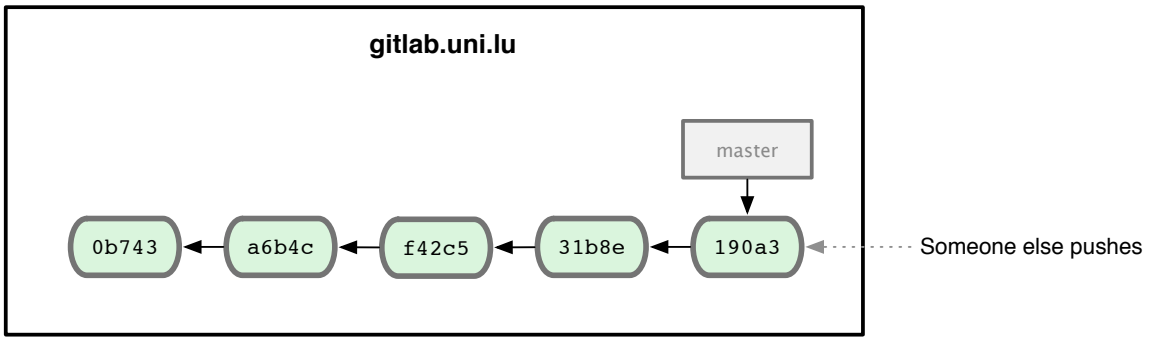
Putting it all together

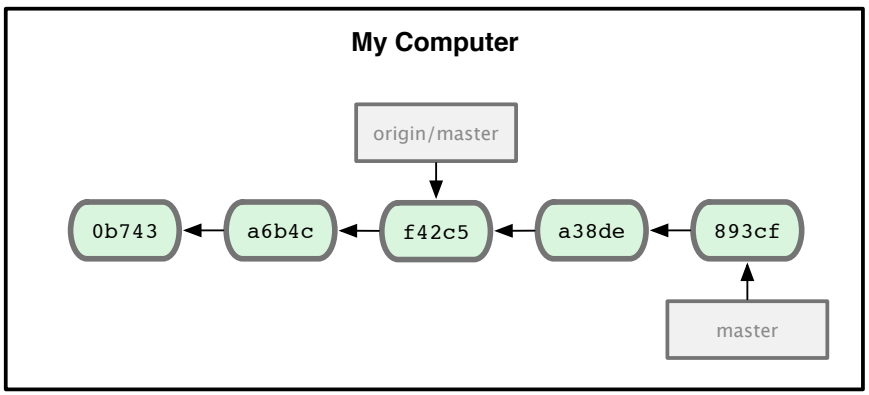


Putting it all together

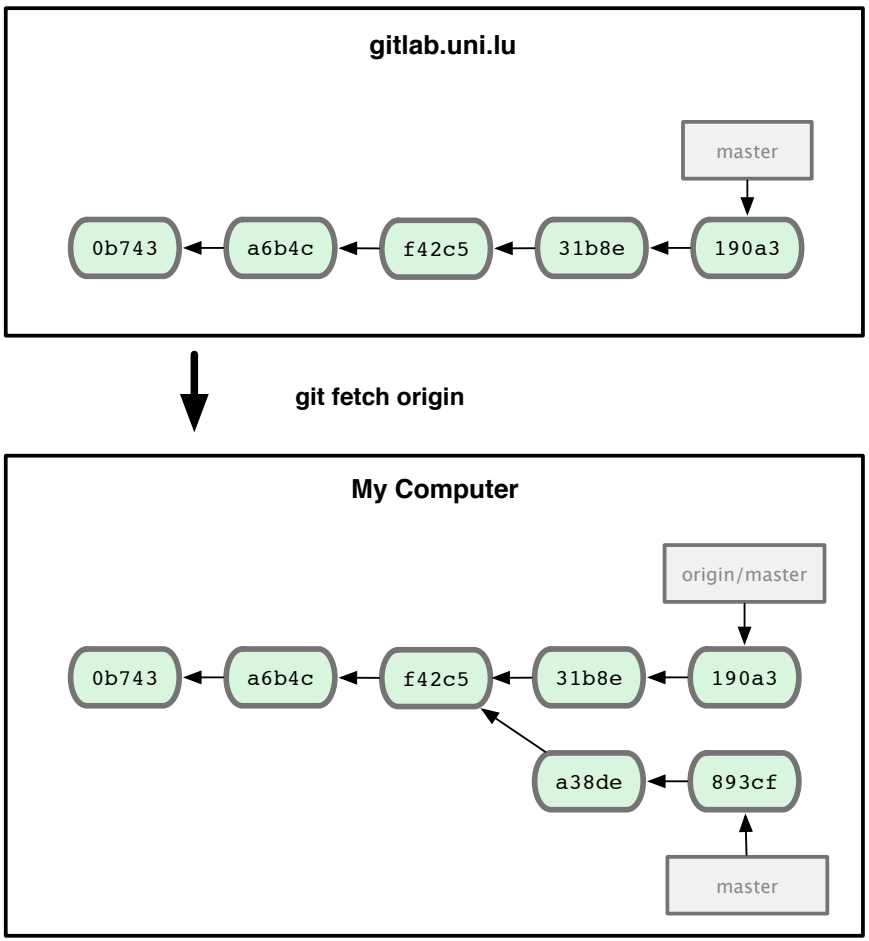


Putting it all together

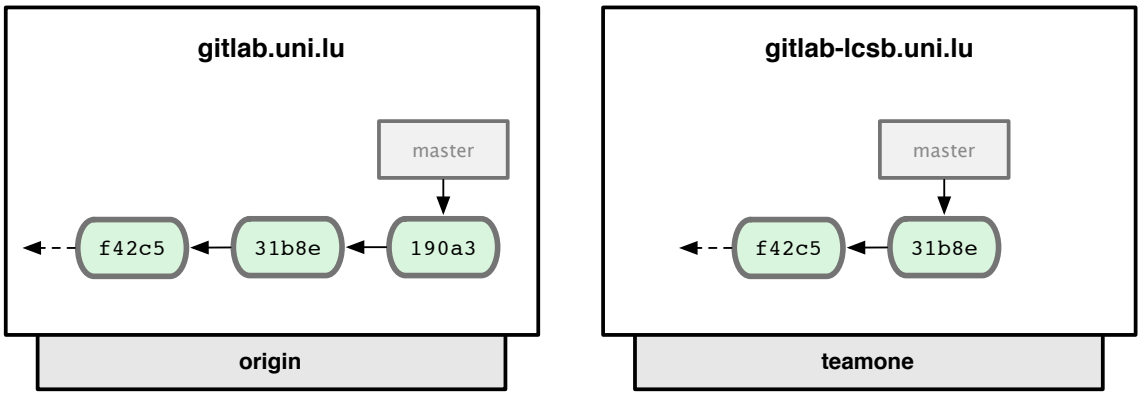




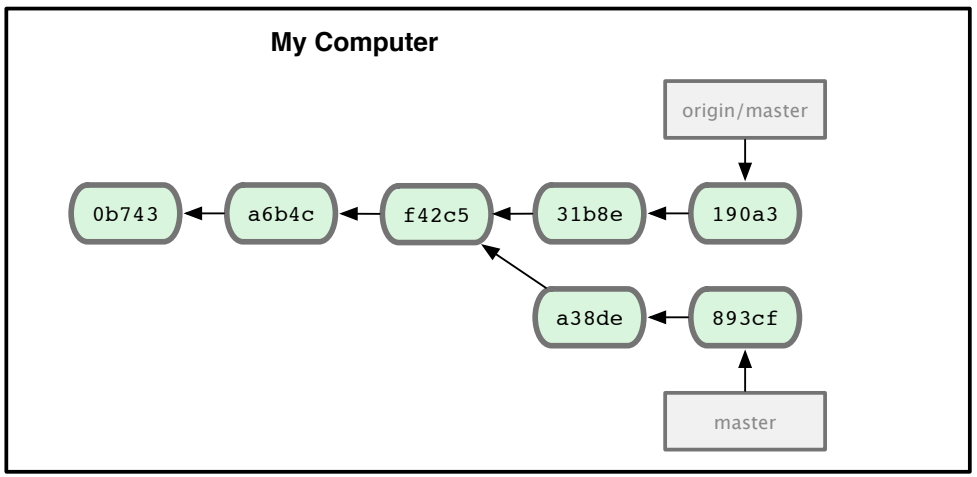
Putting it all together



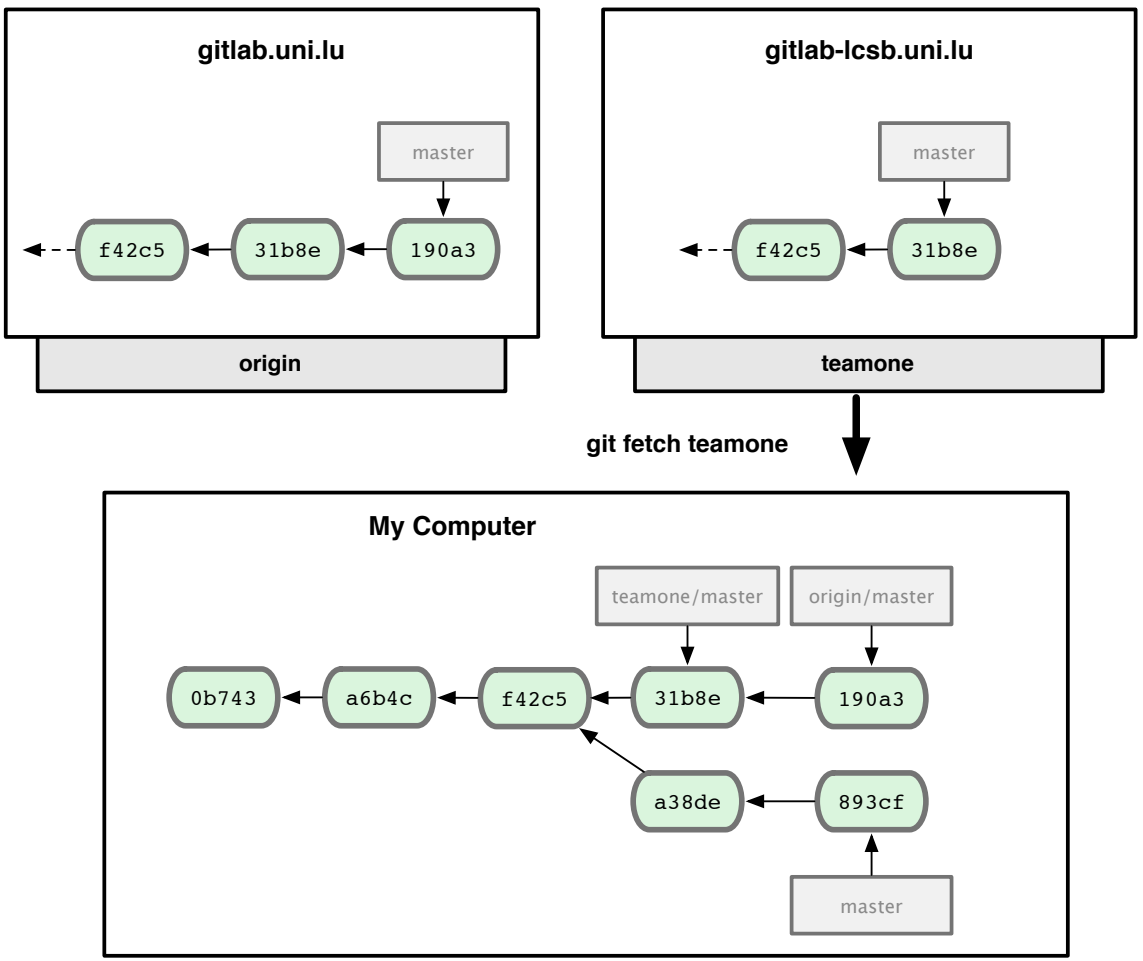
Putting it all together



git remote add teamone git://gitlab-lcsb.uni.lu



Putting it all together



Summary

Summary

Git Submodules

```
$> git submodule add [-b <branch>] <url> <subdir>
```

- **Git submodule**: repository nested within another repository.
  - ↪ see it as a read-only snapshot
  - ↪ make symbolic links to the submodules files
- State saved in `.gitmodules` (git root directory)

```
[submodule ".submodules/Makefiles"]
    path = .submodules/Makefiles
    url = https://github.com/Falkor/Makefiles
```

- Explicit initialization is **mandatory**
  - ↪ **before** cloning `git clone --recursive`
  - ↪ **after** cloning `git submodule init && git submodule updat`

Git Submodules - Update

```
$> git submodule add \
       https://github.com/Falkor/Makefiles .submodules/Makefiles
```

- You might need to **update** the submodules after `fetch` / `pull`
- You might wish to **upgrade** the submodules to the latest version

```
$> git submodule init
$> git submodule update
$> git submodule foreach \
     'git fetch origin; \
      git checkout $(git rev-parse --abbrev-ref HEAD); \
      git reset --hard origin/$(git rev-parse --abbrev-ref HEAD); \
      git submodule update --recursive; git clean -dfx'
```

- See `make upgrade` of this Makefile for repositories

Git subtree

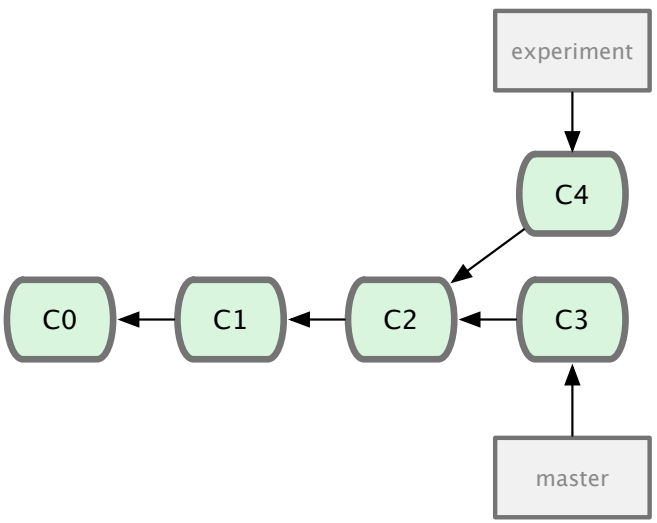
Summary

Rebasing

```
$> git rebase <branch>          # DANGER! Rewrites the tree
```

- Basic (**3-ways**) merging via `git merge` creates a new commit

```
(master)$> git merge experiment
```

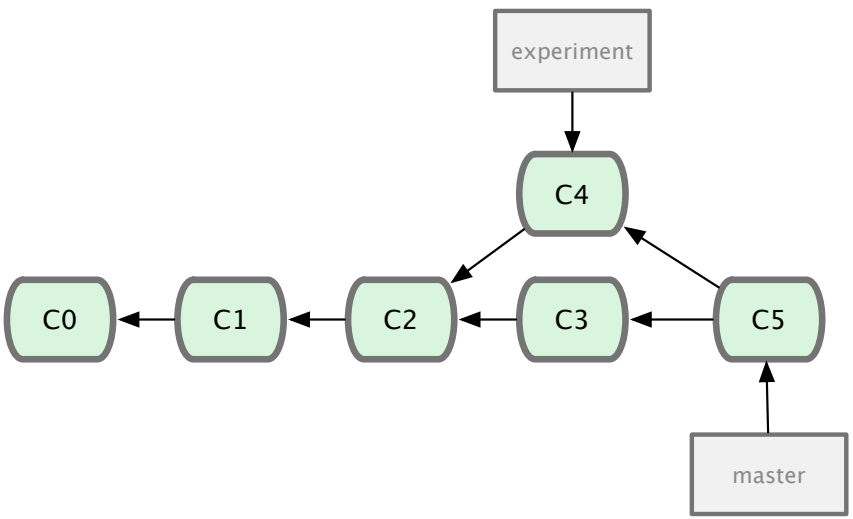


Rebasing

```
$> git rebase <branch>          # DANGER! Rewrites the tree
```

- Basic (**3-ways**) merging via `git merge` creates a new commit

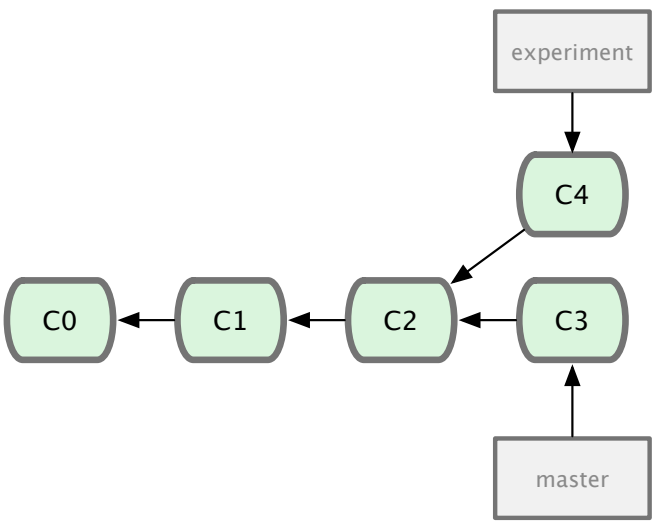
```
(master)$> git merge experiment
```



Rebasing

```
$> git rebase <branch>          # DANGER! Rewrites the tree
```

- **Rebasing**: **Linear** alternative to merging
  - ↪ create a patch of the introduced change (in C4)
  - ↪ reapply it on top (of C3) to create C4'

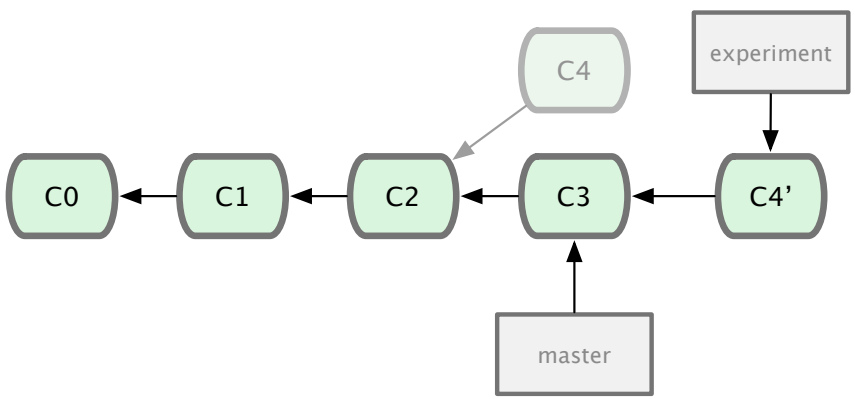


Rebasing

```
$> git rebase <branch>          # DANGER! Rewrites the tree
```

- **Rebasing**: **Linear** alternative to merging
  - ↪ create a patch of the introduced change (in C4)
  - ↪ reapply it on top (of C3) to create C4'

```
(master)$> git checkout experiment
(experiment)$> git rebase master
```

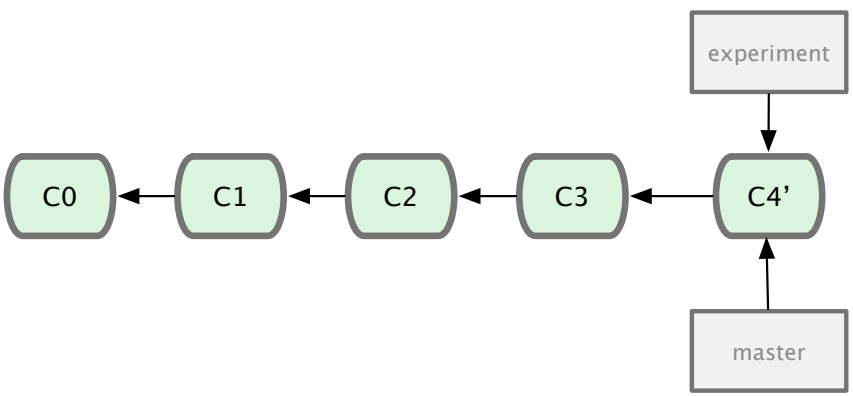


Rebasing

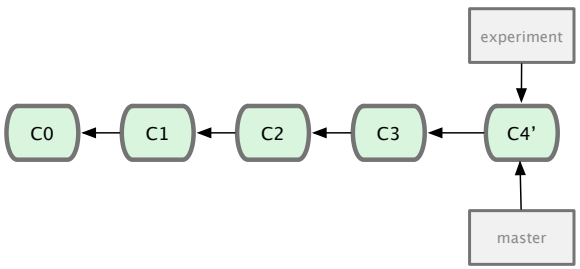
```
$> git rebase <branch>          # DANGER! Rewrites the tree
```

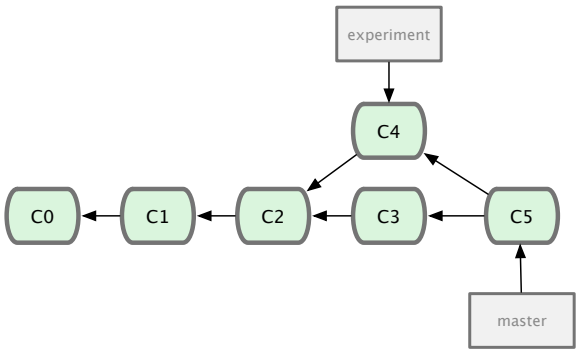
- **Rebasing**: **Linear** alternative to merging
  - ↪ create a patch of the introduced change (in C4)
  - ↪ reapply it on top (of C3) to create C4'

```
(experiment)$> git checkout master
(master)$> git merge experiment
```



Rebasing (left) vs. Merging (right)





- Rebasing ensure your commits apply cleanly on a (remote) branch

Never rebase published code!

Summary

Git-svn: using Git with Subversion

```
$> git svn clone [-s] <svn-url> # checkout an SVN repository
```

- `-s`: standard SVN layout with `trunk/`, `branches/` and `tags/`
- clone into `master` – you **shall** work in another branch **Ex**: `work`

  `$> git checkout -b work`

  ↪ delegate all interactions with SVN repository with `master`

  ↪ thus make all your (local) commits into the `work` branch

Git-svn: using Git with Subversion

```
$> git svn clone [-s] <svn-url> # checkout an SVN repository
```

- `-s`: standard SVN layout with `trunk/`, `branches/` and `tags/`
- clone into `master` – you **shall** work in another branch **Ex**: `work`

  `$> git checkout -b work`

  ↪ delegate all interactions with SVN repository with `master`

  ↪ thus make all your (local) commits into the `work` branch

```
$> git svn rebase       # fetch revisions from SVN and rebase
```

- **Important**: always do that from the `master` branch!

  `(work)$> git checkout master`

  `(master)$> git svn rebase`

Git-svn: commit to Subversion

```
$> git svn dcommit   # create an SVN revision for each commit
```

AFTER you sanitize the 'master' branch!

```
(work)$> git checkout master
(master)$> git svn rebase
```

1. rebase the master branch with the SVN repository

Git-svn: commit to Subversion

```
$> git svn dcommit   # create an SVN revision for each commit
```

AFTER you sanitize the 'master' branch!

```
(master)$> git checkout work
(work)$> git rebase master
```

1. rebase the master branch with the SVN repository
2. go back to the work branch and rebase with master

Git-svn: commit to Subversion

```
$> git svn dcommit   # create an SVN revision for each commit
```

AFTER you sanitize the 'master' branch!

```
(work)$> git log -graph -oneline -decorate                    # OR git gr
```

1. rebase the master branch with the SVN repository
2. go back to the work branch and rebase with master
3. ensure everything is fine

Git-svn: commit to Subversion

```
$> git svn dcommit   # create an SVN revision for each commit
```

AFTER you sanitize the 'master' branch!

```
(work)$> git checkout master
(master)$> git merge -no-ff work
```

1. rebase the master branch with the SVN repository
2. go back to the work branch and rebase with master
3. ensure everything is fine
4. force 3-ways merge your local commit

Git-svn: commit to Subversion

```
$> git svn dcommit   # create an SVN revision for each commit
```

AFTER you sanitize the 'master' branch!

```
(master)$> git commit -amend
```

1. rebase the master branch with the SVN repository
2. go back to the work branch and rebase with master
3. ensure everything is fine
4. force 3-ways merge your local commit
5. edit (amend) the last commit for your SVN dudes

Git-svn: commit to Subversion

```
$> git svn dcommit   # create an SVN revision for each commit
```

AFTER you sanitize the 'master' branch!

```
(master)$> git svn dcommit
```

1. rebase the master branch with the SVN repository
2. go back to the work branch and rebase with master
3. ensure everything is fine
4. force 3-ways merge your local commit
5. edit (amend) the last commit for your SVN dudes
6. **Finally** commit on the SVN server

Git-svn: commit to Subversion

```
$> git svn dcommit   # create an SVN revision for each commit
```

AFTER you sanitize the 'master' branch!

```
(master)$> git checkout work
```

1. rebase the master branch with the SVN repository
2. go back to the work branch and rebase with master
3. ensure everything is fine
4. force 3-ways merge your local commit
5. edit (amend) the last commit for your SVN dudes
6. **Finally** commit on the SVN server
7. Go back to the 'work' branch!

Summary

Shell Integration

- Git Completion – Git flow completion

Colored PS1

- bash: integrate `__git_ps1()` function in your `PS1` variable
  - ↪ normally part of the `bash-completion` package
  - ↪ See integration in the `ULHPC/dotfiles` repository

```
$> export GIT_PS1_SHOWDIRTYSTATE=1    # you probably want that
```

- zsh: agnoster theme / powerline
  - ↪ Mac OS instructions

- On **CentOS/Redhat**, you have to source the correct file

```
$> ln -s /usr/share/git-core/contrib/completion/git-prompt.sh /etc/profile.d/
```

External Merge and Diff Tools

- Git offers visual diff/merge tools, assuming you configured it:

```
$> git config --global merge.tool sourcetree
```

```
$> git difftool [<commit>]                                    diff GUI
```

External Merge and Diff Tools

- Git offers visual diff/merge tools, assuming you configured it:

```
$> git config --global merge.tool sourcetree
```

```
$> git difftool [<commit>]                          diff GUI
```

```
$> git mergetool [<path>...]      # resolving merge conflicts
```

External Merge and Diff Tools

- Git offers visual diff/merge tools, assuming you configured it:

```
$> git config --global merge.tool sourcetree
```

```
$> git difftool [<commit>]                                    diff GUI
```

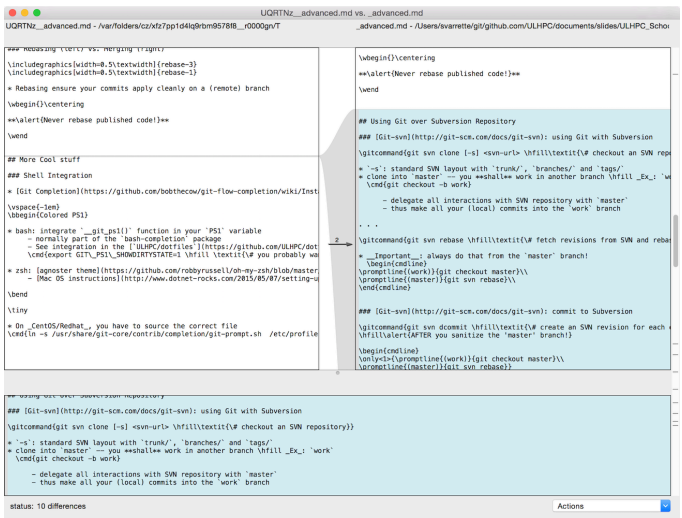
```
$> git mergetool [<path>...]      # resolving merge conflicts
```

- You can set up **another** graphical merge-conflict-resolution tool
  - ↪ List the available tools: `git mergetool --tool-help`
  - ↪ **Mac OS**: `git config --global merge.tool opendiff`
  - ↪ **Linux**: `git config --global merge.tool kdiff3`
  - ↪ Cross-platform: P4Merge (download)

```
$> brew cask install p4merge # on Mac OS, using Homebrew and Cask
```

Using P4Merge as diff/merge tool

```
git config --global merge.tool p4mergetool
git config --global mergetool.p4mergetool.trustexitcode   false
git config --global mergetool.p4mergetool.keeptemporaries false
git config --global mergetool.p4mergetool.keepbackup      false
git config --global mergetool.p4mergetool.cmd \
    $BASE $LOCAL $REMOTE $MERGED
```



- **Alternatives** (mostly Mac OS)
  - ↪ Kaleidoscope
  - ↪ Araxis Merge
  - ↪ DeltaWalker
  - ↪ DiffMerge (free)
  - ↪ SourceTree (free)

Other Cool Stuff

Stashing

- Move changes to a separate "stash".

Interactive Rebase

```
$> git rebase -i <branch>
```

```
$> git stash
$> git stash pop
$> git stash list
$> git stash apply
$> git stash drop
$> git stash clear
```

Other Cool Stuff

Stashing

- Move changes to a separate "stash".

```
$> git stash
$> git stash pop
$> git stash list
$> git stash apply
$> git stash drop
$> git stash clear
```

Interactive Rebase

```
$> git rebase -i <branch>
```

- **Git hooks**:
  - ↪ Located in `.git/hooks/`
  - ↪ scripts run at various stages of Git operation
  - ↪ useful to perform lint actions for instance before pushing

Thank you for your attention...

Questions?

Sébastien Varrette, PhD
mail: sebastien.varrette@uni.lu
Office E-007
Campus Kirchberg
6, rue Coudenhove-Kalergi
L-1359 Luxembourg

UL HPC Management Team
mail: hpc-sysadmins@uni.lu