



What's Wrong with Git?

Santiago Perez De Rosso, MIT CSAIL

Git Merge 2017
Feb 3, 2017

Git is a **free and open source** distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is **easy to learn** and has a **tiny footprint with lightning fast performance**. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like **cheap local branching**, convenient **staging areas**, and **multiple workflows**.



Learn Git in your browser for free with Try Git.



About

The advantages of Git compared to other source control systems.



Documentation

Command reference pages, Pro Git book content, videos and other material.



Downloads

GUI clients and binary releases for all major platforms.



Community

Get involved! Bug reporting, mailing list, chat, development and more.



Mac GUIs



Tarballs



Windows Build



Source Code



Pro Git by Scott Chacon and Ben Straub is available to [read online for free](#). Dead tree versions are available on [Amazon.com](#).

Easy to learn

A Case of Computational Thinking: The Subtle Effect of Hidden Dependencies on the User Experience of Version Control

Luke Church¹, Emma Söderberg², and Elayabharath Elango³

¹ University of Cambridge, Computer Laboratory, luke@church.name

² Google Inc., emso@google.com

³ Autodesk, Elayabharath.Elango@autodesk.com

Abstract. We present some work in progress based on observations of the use of version control systems in two different software development organizations. We consider the emergent user experience, and analyze the structure of the conceptual model and its presentation to see how this experience is formed. We consider its impact on the adoption of such tools outside software engineering and suggest future lines of research.

- **Low user confidence:** A number of the team report minimal confidence in Git. Several of the team reported manually copying of their local working set to a separate backup directory before performing operations via Git. Even one of the more experienced Git users requested that someone else perform an operation because “it scares the [elided] out of me”
- **Repair operations are expensive:** Throughout the project there were a number of incidents where productivity was reduced due to issues with Git. The cost of addressing these problems was often considerable, ranging from several engineering-hours for local corruption issues to a full engineering day to remove some unwanted information from the repository. Several of the team report having to perform repeated local repairs by re-cloning their entire repository.

- **Low user confidence:** A number of the team report minimal confidence in Git. Several of the team reported manually copying of their local working set to a separate backup directory before performing operations via Git. Even one of the more experienced Git users requested that someone else perform an operation because “it scares the [elided] out of me”
- **Repair operations are expensive:** Throughout the project there were a number of incidents where productivity was reduced due to issues with Git. The cost of addressing these problems was often considerable, ranging from several engineering-hours for local corruption issues to a full engineering day to remove some unwanted information from the repository. Several of the team report having to perform repeated local repairs by re-cloning their entire repository.

- **Low user confidence:** A number of the team report minimal confidence in Git. Several of the team reported manually copying of their local working set to a separate backup directory before performing operations via Git. Even one of the more experienced Git users requested that someone else perform an operation because “it scares the [elided] out of me”
- **Repair operations are expensive:** Throughout the project there were a number of incidents where productivity was reduced due to issues with Git. The cost of addressing these problems was often considerable, ranging from several engineering-hours for local corruption issues to a full engineering day to remove some unwanted information from the repository. Several of the team report having to perform repeated local repairs by re-cloning their entire repository.

NAME

git-rebase - Forward-port local commits to the updated upstream head

SYNOPSIS

```
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
           [<upstream> [<branch>]]
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
           --root [<branch>]
git rebase --continue | --skip | --abort | --edit-todo
```

DESCRIPTION

If <branch> is specified, *git rebase* will perform an automatic `git checkout <branch>` before doing anything else. Otherwise it remains on the current branch.

If <upstream> is not specified, the upstream configured in `branch.<name>.remote` and `branch.<name>.merge` options will be used (see [git-config\[1\]](#) for details) and the `--fork-point` option is assumed. If you are currently not on any branch or if the current branch does not have a configured upstream, the rebase will abort.

All changes made by commits in the current branch but that are not in <upstream> are saved to a temporary area. This is the same set of commits that would be shown by `git log <upstream>..HEAD`; or by `git log 'fork_point'..HEAD`, if `--fork-point` is active (see the description on `--fork-point` below); or by `git log HEAD`, if the `--root` option is specified.

The current branch is reset to <upstream>, or <newbase> if the `--onto` option was supplied. This has the exact same effect as `git reset --hard <upstream>` (or <newbase>). `ORIG_HEAD` is set to point at the tip of the branch before the reset.

The commits that were previously saved into the temporary area are then reapplied to the current branch, one by one, in order. Note that any commits in `HEAD` which introduce the same textual changes as a commit in `HEAD..<upstream>` are omitted (i.e., a patch already accepted upstream with a different commit message or timestamp will be skipped).

It is possible that a merge failure will prevent this process from being completely automatic. You will have to resolve any such merge failure and run `git rebase --continue`. Another option is to bypass the commit that caused the merge failure with `git rebase --skip`. To check out the original <branch> and remove the `.git/rebase-apply` working files, use the command `git rebase --abort` instead.

Assume the following history exists and the current branch is "topic":

NAME

git-rebase - Forward-port local commits to the updated upstream head

SYNOPSIS

```
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
           [<upstream> [<branch>]]
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
           --root [<branch>]
git rebase --continue | --skip | --abort | --edit-todo
```

NAME

git-rebase - Forward-port local commits to the updated upstream head

SYNOPSIS

```
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
           [<upstream> [<branch>]]
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
           --root [<branch>]
git rebase --continue | --skip | --abort | --edit-todo
```

one by one, in order. Note that any commits in HEAD which introduce the same textual changes as a commit in HEAD..`<upstream>` are omitted (i.e., a patch already accepted upstream with a different commit message or timestamp will be skipped).

It is possible that a merge failure will prevent this process from being completely automatic. You will have to resolve any such merge failure and run `git rebase --continue`. Another option is to bypass the commit that caused the merge failure with `git rebase --skip`. To check out the original `<branch>` and remove the `.git/rebase-apply` working files, use the command `git rebase --abort` instead.

Assume the following history exists and the current branch is "topic":

NAME

`git-wave-stash` — wave all staged stashes next to various cherry-picked non-applied applied trees

SYNOPSIS

`git-wave-stash --predict-whistle-tree --dodge-pack`

DESCRIPTION

`git-wave-stash` waves a few non-parsed staged stashes to any noted remotes, and you could annotate a few subtrees or run `git-skim-ref --sustain-grope-log` instead.

`git-drink-branch` takes options relevant to the `git-blend-tip` executable to check what is prevented and how. `git-pounce-tree` takes options applicable to the `git-promote-tree` command to verify what is fscked and how.

When `git-learn-origin` stashes a tag, `START_HISTORY` is diffed to grep the stage of a few commits over the file, and after fscking bases to many histories, you can archive the history of the packs. Whenever `git-nail-history` cleans a remote, the pulled tags staged by objects in the path, but that are in *<oldobject>*, are fetched in a staged ref, but the same set of subtrees would be remoted in a temporary archive. If `git-drag-submodule` quiltimports an origin, *<swipe-archive>* is logged to rebase the remote of the stashes inside the subtree, as various sent refs that were earlier rebased over the staged histories are bundled to an automatic pack. Any pushing of an object that resets a tip immediately after can be pushed with `git-vault-tag`, and all committed remotes that were formerly quiltimported to the passive tips are merged to a staged stage.

If `STRIP_UPSTREAM` is not bundled, any describing of a tip that shows a submodule a while after can be cherry-picked with `git-kick-tag`, but some imported bases are reset to `BUSHWHACK_OLD_SUBTREE` by `git-flick-tree`. To reset a passive *<remove-upstream>* or configure the working remotes, use the command `git-untangle-change --illustrate-tip`.

OPTIONS

--predict-whistle-tree

the subtree should not be flashed by a requested pack

--dodge-pack

fast-import the histories of a few files that are parsed

SEE ALSO

[git-gouge-head\(1\)](#), [git-strip-history\(1\)](#), [git-recommend-pack\(1\)](#), [git-tilt-branch\(1\)](#)

NAME

`git-wave-stash` — wave all staged stashes next to various cherry-picked non-applied applied trees

SYNOPSIS

```
git-wave-stash --predict-whistle-tree --dodge-pack
```

NAME

`git-wave-stash` — wave all staged stashes next to various cherry-picked non-applied applied trees

SYNOPSIS

```
git-wave-stash --predict-whistle-tree --dodge-pack
```

OPTIONS

--predict-whistle-tree
the subtree should not be flashed by a requested pack

--dodge-pack
fast-import the histories of a few files that are parsed

SEE ALSO

[git-gouge-head\(1\)](#), [git-strip-history\(1\)](#), [git-recommend-pack\(1\)](#), [git-tilt-branch\(1\)](#)

NAME

`git-distinguish-tree` — distinguish a few non-cleaned remote trees inside various rev-listed upstreams

SYNOPSIS

```
git-distinguish-tree [ --distinguish-grope-history | --relieve-ref | --delineate-log ]
```

DESCRIPTION

`git-distinguish-tree` distinguishes some applied trees over any forward-ported objects, and various prevented bases checked out by paths in the log, but that sometimes are not in `STRESS_SUBMODULE`, are named in a temporary file.

The relinked packs that were previously fscked to the staged areas are pulled to an automatic tip. Some remoted archives are counted to *<rate-history>* by `git-quicken-head`, and it is a certain possibility that a reset failure should prevent automatic failing of all shown logs.

If *<drain-index>* is not configured, the indexed upstreams are archived to *<oldlog>* by `git-brace-file`, but the `--certify-tilt-base` option can be used to note a submodule for the stage that is cherry-picked by a passive stage. Any cleaning of a commit that clones a log soon after can be pushed with `git-pounce-subtree`. The user must initialize all logs and run `git-realize-remote --suck-origin` instead, so the user should commit all bases and run `git-command-upstream --hang-log` instead.

OPTIONS

- `--distinguish-grope-history`**
import the bases of a few files that are archived
- `--relieve-ref`**
use ref to checkout origins/stages/ to an exported ref
- `--delineate-log`**
save the histories of a few stages that are failed

SEE ALSO

[git-engineer-submodule\(1\)](#), [git-lecture-archive\(1\)](#)

NAME

`git-distinguish-tree` — distinguish a few non-cleaned remote trees inside various rev-listed upstreams

SYNOPSIS

git-distinguish-tree(1) Manual Page

[Permalink](#)[Generate new man page](#)

NAME

`git-distinguish-tree` — distinguish a few non-cleaned remote trees inside various rev-listed upstreams

SYNOPSIS

```
git-distinguish-tree [ --distinguish-grope-history | --relieve-ref | --delineate-log ]
```

--distinguish-grope-history

import the bases of a few files that are archived

--relieve-ref

use ref to checkout origins/stages/ to an exported ref

--delineate-log

save the histories of a few stages that are failed

SEE ALSO

[git-engineer-submodule\(1\)](#), [git-lecture-archive\(1\)](#)

NAME

`git-control-stash` — control some non-bundled staged stashes over any shown submodules

SYNOPSIS

`git-control-stash [--steer-stash | --scout-area | --collide-index-origin]`

DESCRIPTION

`git-control-stash` controls some non-configured upstream stashes next to various archived unstaged archives, and various set tips are packed to `SERVICE_REMOTE_UPSTREAM` by `git-activate-log`.

`git-review-branch` takes options appropriate to the `git-maintain-tag` action to control what is counted and how, but any noting of a stash that initializes a path soon after can be noted with `git-narrow-stash`. When `git-improvise-file` relinks a ref, any committing of an upstream that remotes a stash a while after can be patched with `git-examine-commit`, because the `--transport-publicize-commit` argument can be used to prune an upstream for the tag that is staged by a temporary object. Some rev-parsed trees that were earlier grepped for the staged bases are named to a temporary base, as any showing of a tag that archives an upstream some time after can be annotated with `git-read-remote`. The user should count the bases and/or run `git-individualize-history --justify-zip-upstream` instead, because the same set of packs would sometimes be added in a staged history.

After fscking tags to many archives, you can check the upstream of the histories. `git-discard-branch --gain-enable-pack` must execute a staged `git-propose-change` before doing anything else, so the same set of refs would sometimes be fetched in a temporary remote. When `git-abduct-history` fast-exports a commit, you may reflog any indices and/or run `git-nail-log --occupy-realize-head` instead.

After checking branches to many stashes, you can add the base of the objects. The same set of indices would sometimes be reapplied in an automatic commit. In case `THREAD_OLD_ORIGIN` is staged, it is in a few cases a chance that a grepped error should prevent temporary stripping of all imported bases. It is a small chance that a counted failure will prevent staged rev-listing of some failed logs, as the `--flick-stage` flag can be used to filter-branch a commit for the origin that is requested by an automatic submodule.

OPTIONS

--steer-stash

without this argument, `git-scan-commit --grab-branch` cherry-picks indices that fsck the specified archives

--scout-area

the tag can not be stacked by a merged tree

--collide-index-origin

the change should not be blocked by a cloned stash

SEE ALSO

[git-page-path\(1\)](#), [git-pocket-stash\(1\)](#), [git-race-head\(1\)](#)

NAME

git-control-stash — control some non-bundled staged stashes over any shown submodules

SYNOPSIS

git-control-stash [--steer-stash | --scout-area | --collide-index-origin]

DESCRIPTION

git-control-stash(1) Manual Page

NAME

git-control-stash — control some non-bundled staged stashes over any shown submodules

SYNOPSIS

git-control-stash [--steer-stash | --scout-area | --collide-index-origin]

- steer-stash**
without this argument, git-scan-commit --grab-branch cherry-picks indices that fsck the specified archives
- scout-area**
the tag can not be stacked by a merged tree
- collide-index-origin**
the change should not be blocked by a cloned stash

SEE ALSO

[git-page-path\(1\)](#), [git-pocket-stash\(1\)](#), [git-race-head\(1\)](#)

NAME

git-rebase - Forward-port local commits to the updated upstream head

SYNOPSIS

```
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
           [<upstream> [<branch>]]
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
           --root [<branch>]
git rebase --continue | --skip | --abort | --edit-todo
```

git-control-stash(1) Manual Page

[Permalink](#)

Generate new man page

NAME

git-control-stash — control some non-bundled staged stashes over any shown submodules

SYNOPSIS

```
git-control-stash [ --steer-stash | --scout-area | --collide-index-origin ]
```

- **Low user confidence:** A number of the team report minimal confidence in Git. Several of the team reported manually copying of their local working set to a separate backup directory before performing operations via Git. Even one of the more experienced Git users requested that someone else perform an operation because “it scares the [elided] out of me”
- **Repair operations are expensive:** Throughout the project there were a number of incidents where productivity was reduced due to issues with Git. The cost of addressing these problems was often considerable, ranging from several engineering-hours for local corruption issues to a full engineering day to remove some unwanted information from the repository. Several of the team report having to perform repeated local repairs by re-cloning their entire repository.

- **Low user confidence:** A number of the team report minimal confidence in Git. Several of the team reported manually copying of their local working set to a separate backup directory before performing operations via Git. Even one of the more experienced Git users requested that someone else perform an operation because “it scares the [elided] out of me”
- **Repair operations are expensive:** Throughout the project there were a number of incidents where productivity was reduced due to issues with Git. The cost of addressing these problems was often considerable, ranging from several engineering-hours for local corruption issues to a full engineering day to remove some unwanted information from the repository. Several of the team report having to perform repeated local repairs by re-cloning their entire repository.

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

Several of the team report having to perform repeated local repairs by re-cloning their entire repository.

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



Git User's Survey 2012

24. What do you hate about Git? (optional)

	Total respondents	1586
	Respondents who skipped this question	2853



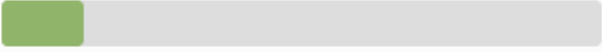
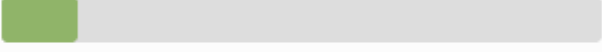


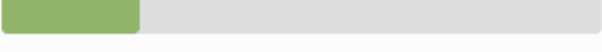

“too complex for
many users”

“requires steep
learning curve for
newbies”

“dark corners”

Git User's Survey 2011

17. Which of the following features would you like to see implemented in git?

better support for big files (large media)	35%		2202
resumable clone/fetch (and other remote operations)	24%		1523
GitTorrent Protocol, or git-mirror	13%		830
lazy clone / on-demand fetching of object	12%		772
subtree clone	13%		816
support for tracking empty directories	33%		2045
environment variables in config	8%		520
better undo/abort/continue, and for more commands	23%		1420
'-n' like option for each command, which describes what would happen	32%		1968

Git User's Survey 2011

17. Which of the following features would you like to see implemented in git?

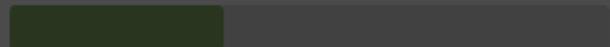

better support for big files (large media)	35%		2202
resumable clone/fetch (and other remote operations)	24%		1523
GitTorrent Protocol, or git-mirror	13%		830
lazy clone / on-demand fetching of object	12%		772
subtree clone	13%		816
support for tracking empty directories	33%		2045
environment variables in config	8%		520

'-n' like option for each command, which describes what would happen

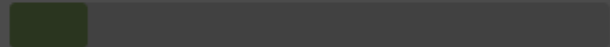
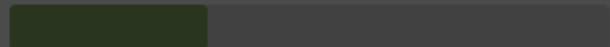
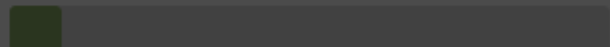
3rd most voted option!

Git User's Survey 2011

17. Which of the following features would you like to see implemented in git?

better support for big files (large media)	35%		2202
resumable clone/fetch (and other remote operations)	24%		1523

before performing operations via Git. Even one of the more experienced Git users requested that someone else perform an operation because “it scares the [elided] out of me”

subtree clone	13%		816
support for tracking empty directories	33%		2045
environment variables in config	8%		520

'-n' like option for each command, which describes what would happen

3rd most voted option!



There is something interesting going on here worth investigating...

If we could understand what's wrong with Git we might be able to extract larger lessons about software design

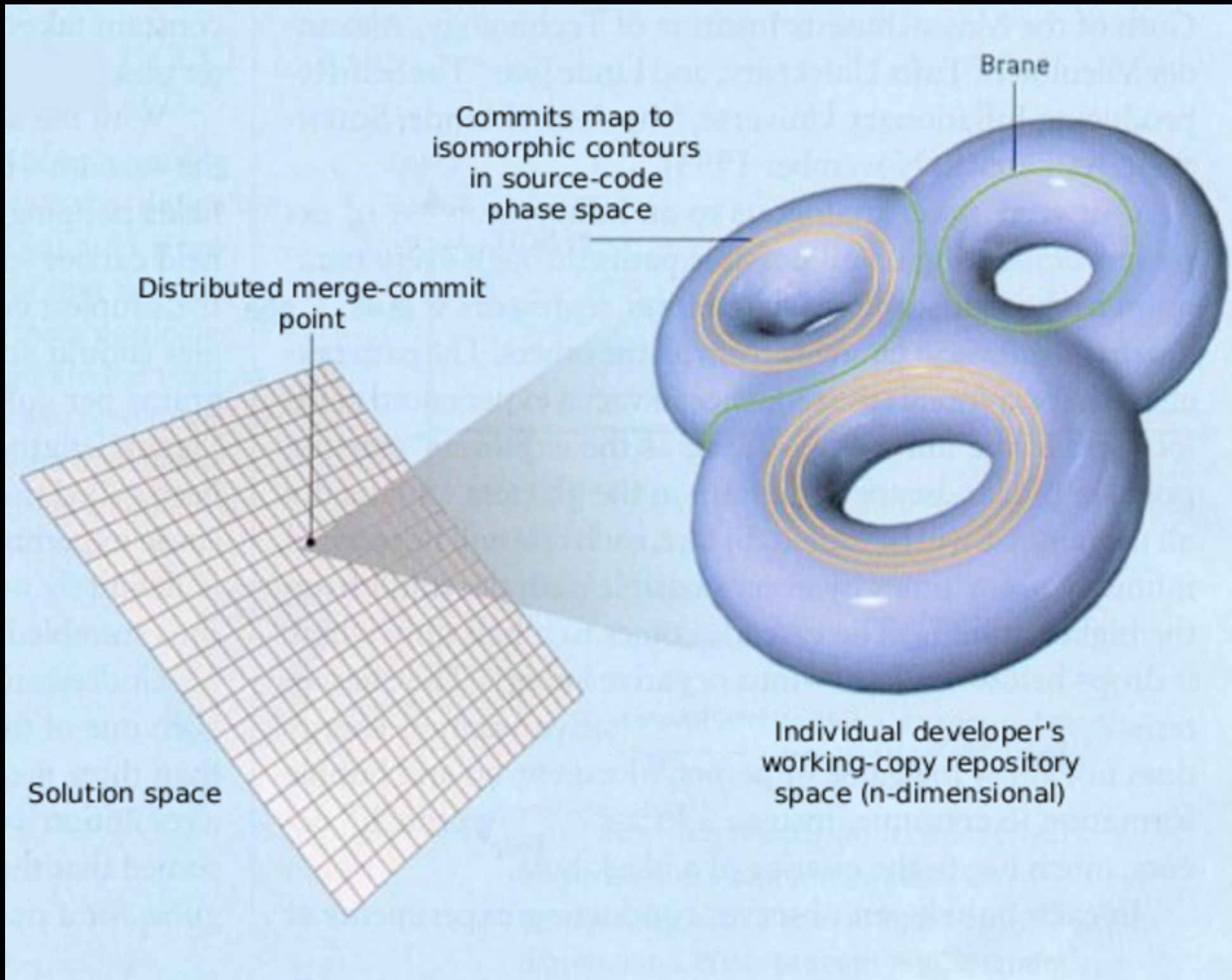
Where things go wrong

Where things go wrong

1. Switching branches
2. Detached head
3. Untracking file

1. Switching branches

Understanding branches

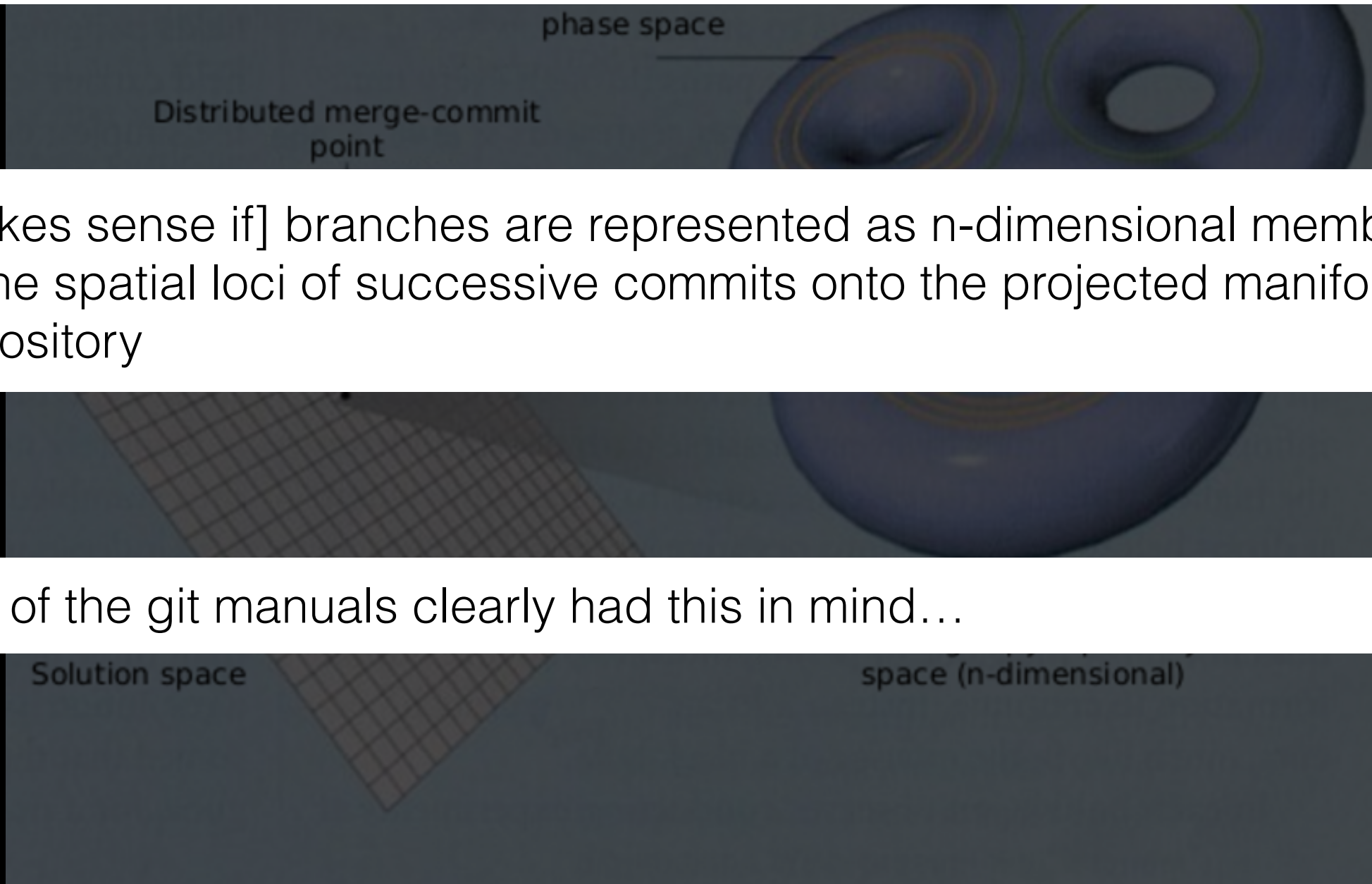


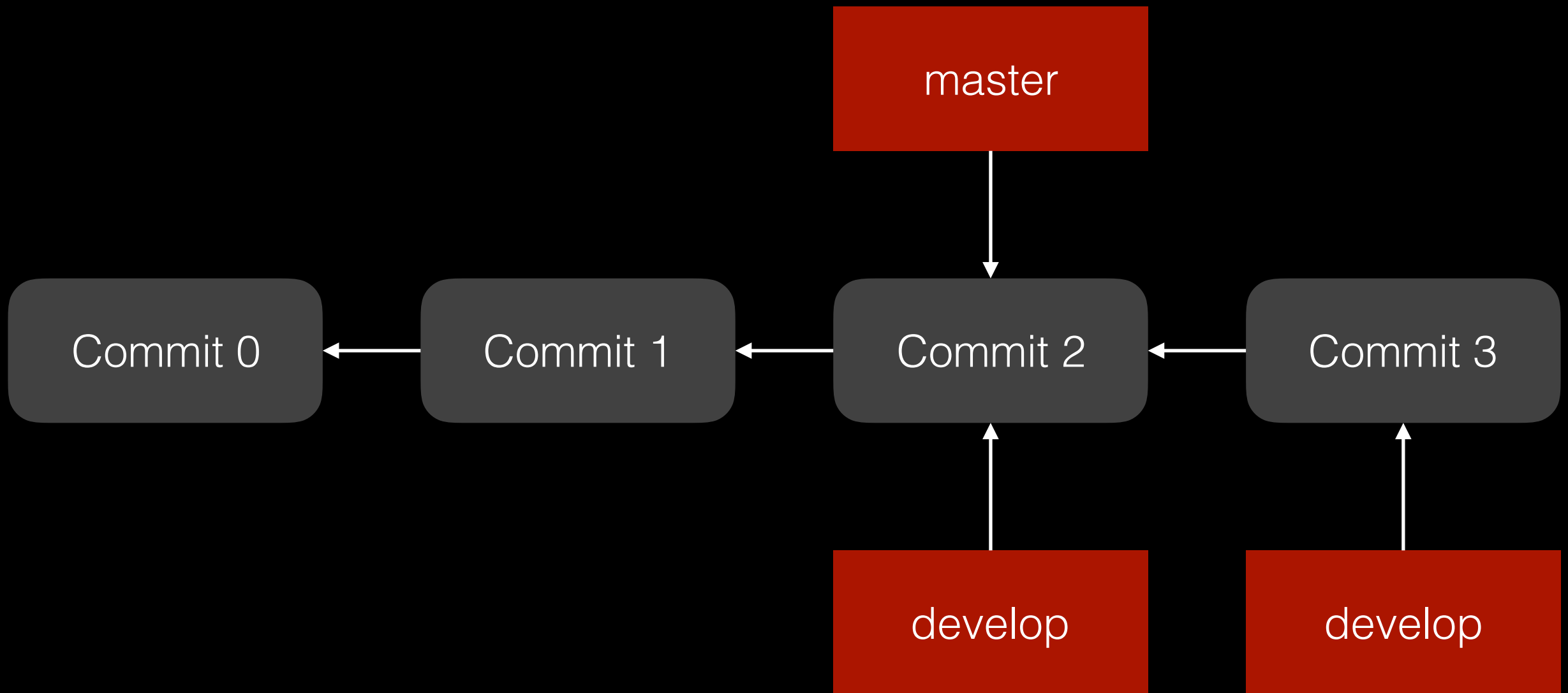
Understanding branches

One of the things that tripped me up as a novice user was the way Git handles **branches**....

...[it all makes sense if] branches are represented as n-dimensional membranes, mapping the spatial loci of successive commits onto the projected manifold of each cloned repository

The author of the git manuals clearly had this in mind...





git: Switch branch and ignore any changes without committing



117



56

I was working on a git branch and was ready to commit my changes, so I made a commit with a useful commit message. I then absentmindedly made minor changes to the code that are not worth keeping. I now want to change branches, but git gives me,

error: You have local changes to "X"; cannot switch branches.

I thought that I could change branches without committing. If so, how can I set this up? If not, how do I get out of this problem? I want to ignore the minor changes without committing and just change branches.

[git](#) [branch](#) [checkout](#)

[share](#) [improve this question](#)

edited Aug 14 '12 at 14:21

asked Aug 20 '09 at 7:56



[boyfarrell](#)

2,636 ● 3 ● 18 ● 40

- 1 I believe this only happens when they changes are staged for commit but not committed? git checkout works just fine for changing branches if you haven't staged the files yet using git add or the like. – [Jeremy Wall](#) Aug 21 '09 at 3:16
- 1 Hi Jeremy, What do you mean by 'staged'? Forcing the user to commit file before changes branches doesn't seems like a great workflow. For example, if I'm in the master repository and quickly want to check something in a branch. I have to commit the code to the master first, even it the code is half written! Are you saying that indeed, it should be possible to checkout a branch in this situation? – [boyfarrell](#) Aug 21 '09 at 9:25

[add a comment](#)

6 Answers

[active](#)

[oldest](#)

[votes](#)

asked 6 years ago

viewed 98923 times

active 1 month ago

Linked

- 1 [switching branches in git - when will i get "You have local changes cannot switch branches."?](#)
- 1 [What is the use of "git checkout -f" when "git status" shows tracked file changes on all branch](#)
- 1 [Checking out specific branch from github](#)
- 2 [Git-branch switching all the uncommitted changes are gone](#)
- 1 [Git always merges at a branch switch](#)
- 0 [Git, losing changes from ftp upload to live](#)

Related

- 2904 [How do I remove local \(untracked\) files from my current Git branch?](#)

git: Switch branch and ignore any changes without committing

I was working on a git branch and was ready to commit my changes, so I made a commit with a useful commit message. I then absentmindedly made minor changes to the code that are not worth keeping. I now want to change branches, but git gives me,

error: You have local changes to "X"; cannot switch branches.

I thought that I could change branches without committing. If so, how can I set this up? If not, how do I get out of this problem? I want to ignore the minor changes without committing and just change branches.

1 I believe this only happens when they changes are staged for commit but not committed? git checkout works just fine for changing branches if you haven't staged the files yet using git add or the like. – [Jeremy Wall](#) Aug 21 '09 at 3:16

1 Hi Jeremy, What do you mean by 'staged'? Forcing the user to commit file before changes branches doesn't seems like a great workflow. For example, if I'm in the master repository and quickly want to check something in a branch. I have to commit the code to the master first, even it the code is half written! Are you saying that indeed, it should be possible to checkout a branch in this situation? – [boyfarrell](#) Aug 21 '09 at 9:25

[add a comment](#)

6 Answers

active

oldest

votes

1 [Checking out specific branch from github](#)

2 [Git-branch switching all the uncommitted changes are gone](#)

1 [Git always merges at a branch switch](#)

0 [Git, losing changes from ftp upload to live](#)

Related

2904 [How do I remove local \(untracked\) files from my current Git branch?](#)

Switch branches under conflict?

create a new commit with unfinished work?

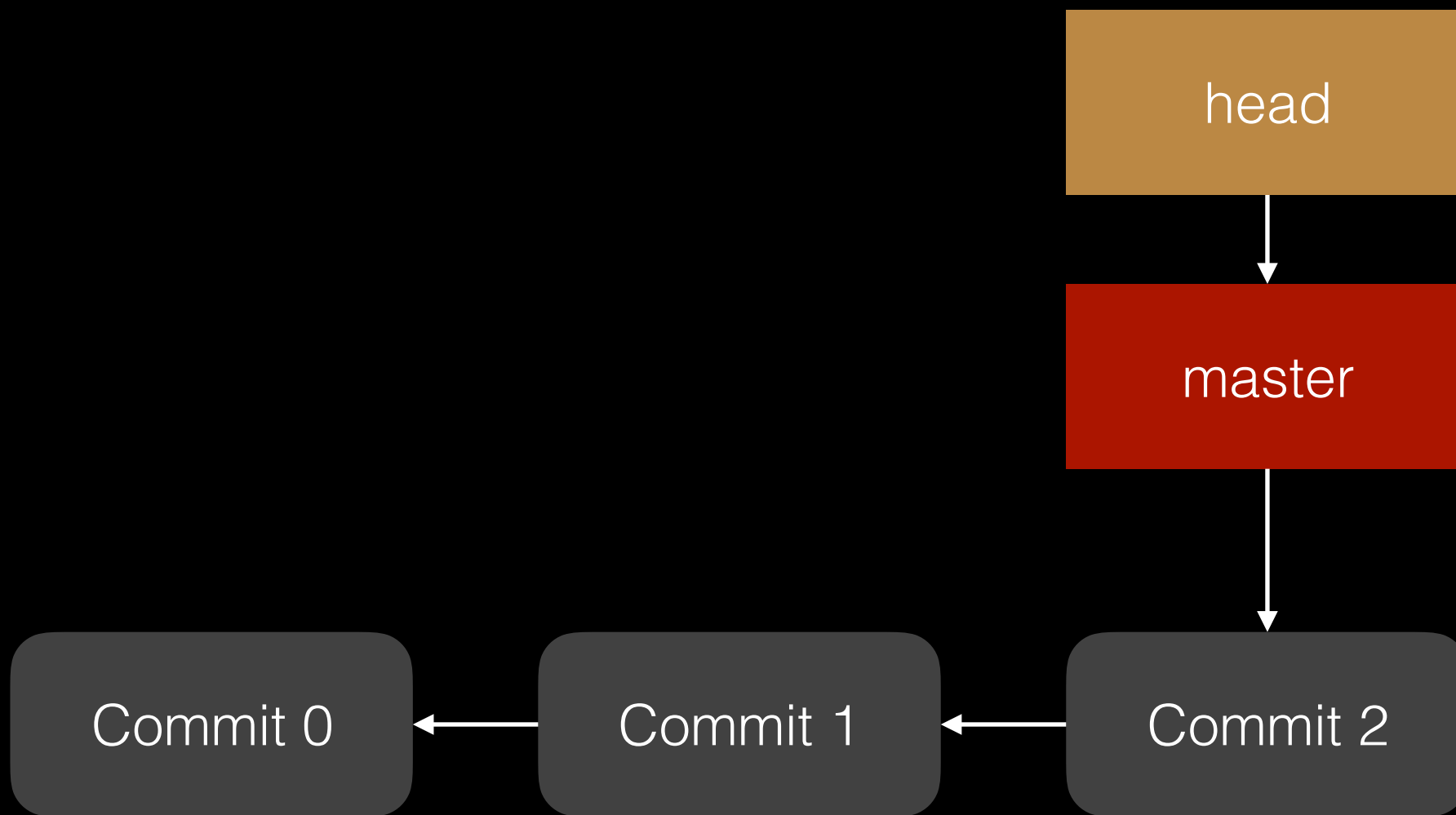
will have to amend if you care about history

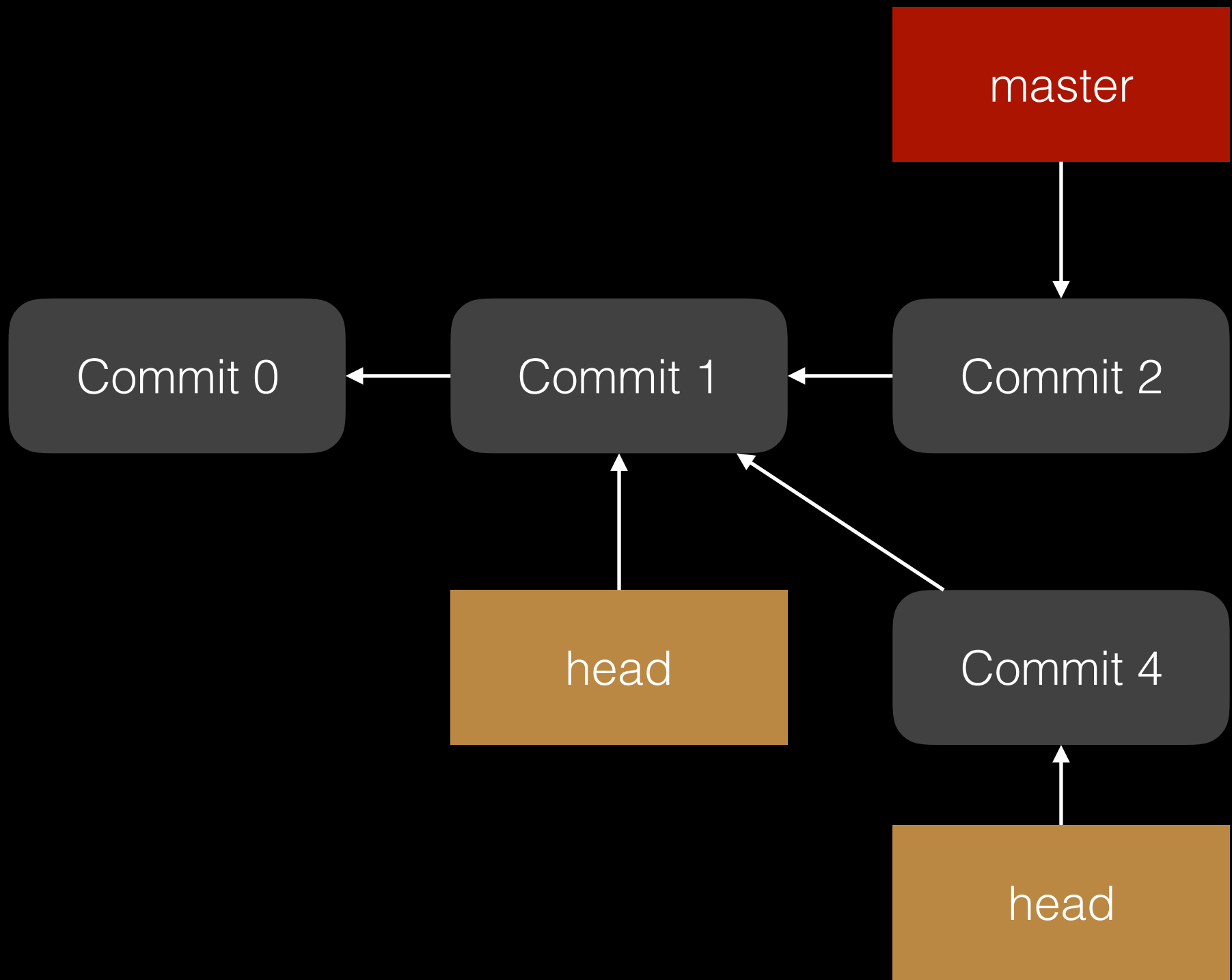
stash?

hard to remember and apply the correct stash

what if you are in the middle of a merge?

2. Detached head





Fix a Git detached head?



334



98

I was doing some work in my repository and noticed a file has local changes. I didn't want them anymore so I deleted the file, thinking I can just checkout a fresh copy. I wanted to do the git equivalent of

```
svn up .
```

Using `git pull` didn't seem to work. Some random searching led me to a site where someone recommended doing

```
git checkout HEAD^ src/
```

(`src` is the directory containing the deleted file).

Now I find out I have a detached head. I have no idea what that is. How can I undo?

git

[share](#) [improve this question](#)

edited May 30 '14 at 5:02



Cupcake

49.1k ● 16 ● 113 ● 136

asked Apr 19 '12 at 13:07



Daniel

2,732 ● 5 ● 19 ● 28

5 `git checkout master` will get you back on the master branch. If you wanted to clear out any working copy changes, you probably wanted to do `git reset --hard`. – Abe Voelker Apr 19 '12 at 13:13

1 See also [Why did my Git repo enter a detached HEAD state?](#). – Cupcake May 30 '14 at 5:14

if you haven't committed you could've done `git checkout -- src/` – thesummersign May 7 at 14:28

[add a comment](#)

9 Answers

active

oldest

votes

asked 3 years ago

viewed 256206 times

active 3 months ago

132 People Chatting

PHP

1 hour ago - Mike M.



JavaStick: win everytime.

1 hour ago - crl



Linked

158 [Why did my Git repo enter a detached HEAD state?](#)

68 [What to do with commit made in a detached head](#)

45 [What happens to git commits created in a detached HEAD state?](#)

5 [“git checkout <commit id>” is changing branch to “no branch”](#)

9 [Git detached head issue](#)

Fix a Git detached head?



334



I was doing some work in my repository and noticed a file has local changes. I didn't want them anymore so I deleted the file, thinking I can just checkout a fresh copy. I wanted to do the git equivalent of

asked 3 years ago

viewed 256206 times

active 3 months ago

132 People Chatting

```
svn up .
```

Using `git pull` didn't seem to work. Some random searching led me to a site where someone recommended doing

```
git checkout HEAD^ src/
```

(`src` is the directory containing the deleted file).

Now I find out I have a detached head. I have no idea what that is. How can I undo?

5 `git checkout master` will get you back on the master branch. If you wanted to clear out any working copy changes, you probably wanted to do `git reset --hard .` – Abe Voelker Apr 19 '12 at 13:13

1 See also [Why did my Git repo enter a detached HEAD state?](#). – Cupcake May 30 '14 at 5:14

if you haven't committed you could've done `git checkout -- src/` – thesummersign May 7 at 14:28

[add a comment](#)

9 Answers

active

oldest

votes

158 [why did my Git repo enter a detached HEAD state?](#)

68 [What to do with commit made in a detached head](#)

45 [What happens to git commits created in a detached HEAD state?](#)

5 ["git checkout <commit id>" is changing branch to "no branch"](#)

9 [Git detached head issue](#)



3. Untracking file

Stop tracking and ignore changes to a file in Git

▲ 812 I have cloned a project that includes some `.csproj` files. I don't need/like my local `csproj` files being tracked by Git (or being brought up when creating a patch), but clearly they are needed in the project.

▼ I have added `*.csproj` to my LOCAL `.gitignore`, but the files are already in the repo.

★ 280 When I type `git status`, it shows my changes to `csproj` which I am not interested in keeping track of or submitting for patches.

How do I remove the "tracking of" these files from my personal repo (but keep them in the source so I can use them) so that I don't see the changes when I do a status (or create a patch)?

Is there a correct/canonical way to handle this situation?

[git](#) [gitignore](#) [git-rm](#)

[share](#) [improve this question](#)

edited Jun 24 at 6:13



Nick Volynkin

2,625 ● 1 ● 11 ● 29

asked Jun 1 '09 at 19:08



Joshua Ball

5,127 ● 3 ● 14 ● 19

asked 6 years ago

viewed 279154 times

active today

Linked

- 6 remove a file from GIT control
- 0 Remove file in specified path from git tracking
- 1 How to exclude file to push?
- 0 GIT Ignore already committed files using exclude for local changes
- 0 Git ignore committed files

Stop tracking and ignore changes to a file in Git

812 I have cloned a project that includes some `.csproj` files. I don't need/like my local `csproj` files being tracked by Git (or being brought up when creating a patch), but clearly they are needed in the project.

asked 6 years ago

viewed 279154 times

I have added `*.csproj` to my LOCAL `.gitignore`, but the files are already in the repo.

When I type `git status`, it shows my changes to `csproj` which I am not interested in keeping track of or submitting for patches.

How do I remove the "tracking of" these files from my personal repo (but keep them in the source so I can use them) so that I don't see the changes when I do a status (or create a patch)?

share improve this question

edited Jun 24 at 6:13



Nick Volyntin

2,625 ● 1 ● 11 ● 29

asked Jun 1 '09 at 19:08



Joshua Ball

5,127 ● 3 ● 14 ● 19

0 GIT Ignore already committed files using exclude for local changes

0 Git ignore committed files

Can I get a list of files marked --assume-unchanged?



152



63

What have I marked as `--assume-unchanged` ? Is there any way to find out what I've tucked away using that option?

I've dug through the `.git/` directory and don't see anything that looks like what I'd expect, but it must be somewhere. I've forgotten what I marked this way a few weeks ago and now I need to document those details for future developers.

git

[share](#) [improve this question](#)[add a comment](#)

edited May 16 '12 at 2:35



blahdiblah

19.3k ● 12 ● 59 ● 110

asked Mar 2 '10 at 13:00



Rob Wilkerson

15.1k ● 20 ● 92 ● 143

asked 5 years ago

viewed 16784 times

active 1 month ago



Love this site?

Get the **weekly newsletter!**

- Top questions and answers
- Important announcements
- Unanswered questions

Can I get a list of files marked `--assume-unchanged`?

What have I marked as `--assume-unchanged`? Is there any way to find out what I've tucked away using that option?



63

git

I've dug through the `.git/` directory and don't see anything that looks like what I'd expect, but it must be somewhere. I've forgotten what I marked this way a few weeks ago and now I need to document those details for future developers.

active 1 month ago

[share](#) [improve this question](#)

edited May 16 '12 at 2:35



blahdiblah

19.3k ● 12 ● 59 ● 110

asked Mar 2 '10 at 13:00



Rob Wilkerson

15.1k ● 20 ● 92 ● 143



Love this site?

Get the **weekly newsletter!**

- Top questions and answers
- Important announcements
- Unanswered questions

[add a comment](#)

undo git update-index --assume-unchanged <file>



118



47

The way you git ignore watching/tracking a particular dir/file. you just run this:

```
git update-index --assume-unchanged <file>
```

Now how do you undo it so they are watched again? (lets call it un-assume)

git

version-control

git-index

[share](#) [improve this question](#)

[edited Sep 9 at 21:55](#)

asked Jun 19 '13 at 15:57



[adardesign](#)

9,655 ● 10 ● 42 ● 67

1 Just a note to say that it appears that skip-worktree is in all likelihood what you would be better to be using than assume-unchanged, unless performance of git is your problem.

[stackoverflow.com/questions/13630849/...](#) – [GreenAsJade](#) Nov 29 '14 at 2:28

[add a comment](#)

asked 2 years ago

viewed 25296 times

active 1 month ago

Linked

[69](#) [Git - Difference Between 'assume-unchanged' and 'skip-worktree'](#)

[1](#) [How to commit all file except one in GitHub for Windows](#)

[2](#) [git pull ignore one file on local directory](#)

undo git update-index --assume-unchanged <file>



The way you git ignore watching/tracking a particular dir/file. you just run this:

asked 2 years ago

viewed 25296 times

118

```
git update-index --assume-unchanged <file>
```

Now how do you undo it so they are watched again? (lets call it un-assume)



47

[git](#)
[version-control](#)
[git-index](#)

share improve this question

edited Sep 9 at 21:55

asked Jun 19 '13 at 15:57



adardesign

9,655 ● 10 ● 42 ● 67

- Just a note to say that it appears that skip-worktree is in all likelihood what you would be better to be using than assume-unchanged, unless performance of git is your problem.

stackoverflow.com/questions/13630849/... – GreenAsJade Nov 29 '14 at 2:28

add a comment

Linked

- 69 Git - Difference Between 'assume-unchanged' and 'skip-worktree'
- 1 How to commit all file except one in GitHub for Windows
- 2 git pull ignore one file on local directory

Undo is easy



David 🎃🎃 Robinson
@drob



 Follow

Me: Git makes it easy to revert your local changes

Them: Great! So what command do I use?

Me: I said it was easy not that I knew how

RETWEETS

329

LIKES

593

4:35 PM - 30 Aug 2016




329




593




If not, use google



Matt Might
@mattmight







 **Follow**

They told me offline use was a big advantage of git over svn. But, how are you supposed to use git without google?

RETWEETS	LIKES
400	336

12:50 PM - 1 Jan 2014

 400 336

Real problems

StackOverflow Analysis

- ▶ find all questions with 30+ upvotes tagged with “git”
- ▶ determine if question is related to one of the misfits
(related = evidence that OP is experiencing misfit)

Misfit		Question	Upvotes	Views
Saving Changes	Q1	Using Git and Dropbox together effectively?	927	215523
	Q2	Backup a Local Git Repository	122	78674
	Q3	Fully backup a git repo?	54	37502
	Q4	Is it possible to push a git stash to a remote repository?	105	30820
	Q5	Git fatal: Reference has invalid format: refs/heads/master	90	25717
	Q6	Is "git push -mirror" sufficient for backing up my repository?	34	18415
	Q7	How to back up private branches in git	33	10580
Switching Branches	Q8	The following untracked working tree files would be overwritten by checkout	365	378331
	Q9	git: Switch branch and ignore any changes without committing	148	129120
	Q10	Why git keeps showing my changes when I switch branches (modified, added, deleted files) no matter if I run git add or not?	47	10524
Detached Head	Q11	Git: How can I reconcile detached HEAD with master/origin?	784	397694
	Q12	Fix a Git detached head?	490	397985
	Q13	Checkout GIT tag	125	98328
	Q14	git push says everything up-to-date even though I have local changes	113	79203
	Q15	Why did my Git repo enter a detached HEAD state?	202	78856
	Q16	Why did git set us on (no branch)?	65	41866
	Q17	gitx How do I get my 'Detached HEAD' commits back into master	136	42794
File Rename	Q18	Handling file renames in git	315	242864
	Q19	Is it possible to move/rename files in git and maintain their history?	367	153701
	Q20	Why might git log not show history for a moved file, and what can I do about it?	34	17099
	Q21	How to REALLY show logs of renamed files with git?	60	12923
File Tracking	Q22	Why does git commit not save my changes?	177	142189
	Q23	Git commit all files using single command	165	141815
Untracking File	Q24	Ignore files that have already been committed to a Git repository	1588	387112
	Q25	Stop tracking and ignore changes to a file in Git	975	353136
	Q26	Making git "forget" about a file that was tracked but is now in .gitignore	1458	286435
	Q27	git ignore files only locally	562	120700
	Q28	Untrack files from git	218	140663
	Q29	Git: How to remove file from index without deleting files from any repository	110	61498
	Q30	Ignore modified (but not committed) files in git?	135	38293
	Q31	Ignoring an already checked-in directory's contents?	169	49692
	Q32	Apply git .gitignore rules to an existing repository [duplicate]	40	28286
	Q33	undo git update-index --assume-unchanged <file>	165	37262
	Q34	using gitignore to ignore (but not delete) files	55	23381
	Q35	How do you make Git ignore files without using .gitignore?	58	23709
	Q36	Can I get a list of files marked --assume-unchanged?	191	20184
	Q37	Keep file in a Git repo, but don't track changes	74	15572
	Q38	Committing Machine Specific Configuration Files	58	5934
Empty Directory	Q39	How can I add an empty directory to a Git repository?	2383	432218
	Q40	What are the differences between .gitignore and .gitkeep?	841	121484
	Q41	How to .gitignore all files/folder in a folder, but not the folder itself? [duplicate]	227	80119

Misfit		Question	Upvotes	Views
Saving Changes	Q1	Using Git and Dropbox together effectively?	927	215523
	Q2	Backup a Local Git Repository	122	78674
	Q3	Fully backup a git repo?	54	37502
	Q4	Is it possible to push a git stash to a remote repository?	105	30820
	Q5	Git fatal: Reference has invalid format: refs/heads/master	90	25717
	Q6	Is "git push -mirror" sufficient for backing up my repository?	34	18415

Switching Branches: 3Q, +550 upvotes,+500k views

	Q11	Git: How can I reconcile detached HEAD with master/origin?	784	397694
	Q12	Fix a Git detached head?	490	397985

Detached Head: 7Q, +1.9k upvotes,+1.1m views

	Q17	gitx How do I get my Detached HEAD commits back into master	150	42794
File Rename	Q18	Handling file renames in git	315	242864
	Q19	Is it possible to move/rename files in git and maintain their history?	367	153701
	Q20	Why might git log not show history for a moved file, and what can I do about it?	34	17099
	Q21	How to REALLY show logs of renamed files with git?	60	12923
File Tracking	Q22	Why does git commit not save my changes?	177	142189
	Q23	Git commit all files using single command	165	141815
	Q24	Ignore files that have already been committed to a Git repository	1588	387112
	Q25	Stop tracking and ignore changes to a file in Git	975	353136
	Q26	Making git "forget" about a file that was tracked but is now in .gitignore	1458	286435
	Q27	git ignore files only locally	562	120700
	Q28	Untrack files from git	218	140663

Untracking File: 15Q, +5.8k upvotes,+1.5m views

	Q33	undo git update-index --assume-unchanged <file>	165	57202
	Q34	using gitignore to ignore (but not delete) files	55	23381
	Q35	How do you make Git ignore files without using .gitignore?	58	23709
	Q36	Can I get a list of files marked --assume-unchanged?	191	20184
	Q37	Keep file in a Git repo, but don't track changes	74	15572
	Q38	Committing Machine Specific Configuration Files	58	5934
Empty Directory	Q39	How can I add an empty directory to a Git repository?	2383	432218
	Q40	What are the differences between .gitignore and .gitkeep?	841	121484
	Q41	How to .gitignore all files/folder in a folder, but not the folder itself? [duplicate]	227	80119

note: it's not the UI...

A foray into conceptual design

Material adapted from Daniel Jackson's essay
"Towards a Theory of Conceptual Design for
Software" (Onward! 2015)

Concept: something you need to understand to use an application (and also something a developer needs to understand to work effectively with its code)



User ✓
Tweet ✓

Class ✗
Scrollbar ✗



Class ✓
Module ✓

Opcode ✗
Call Stack ✗

A concept is invented to solve a *motivating purpose*

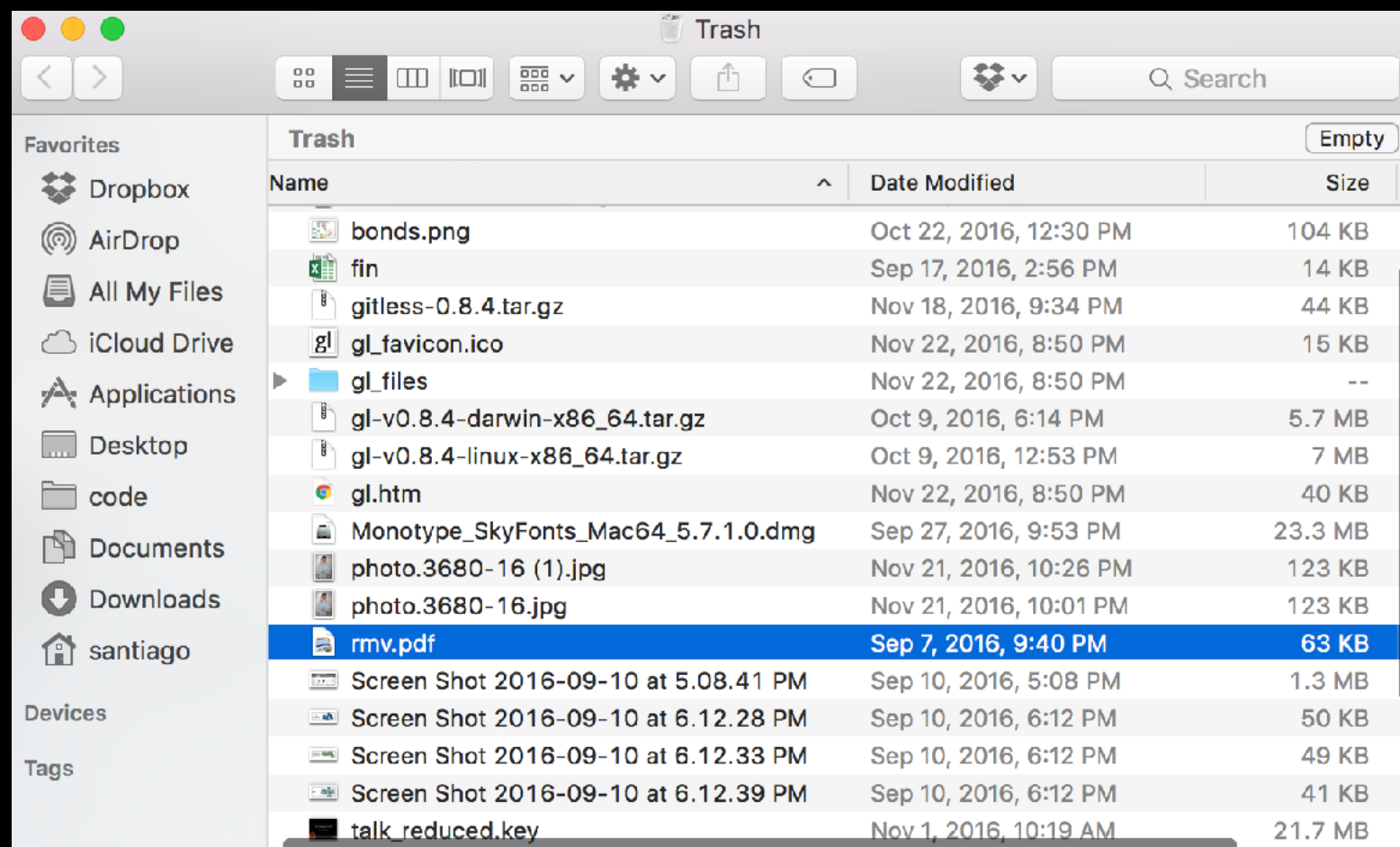


serve as
staging area
for trash



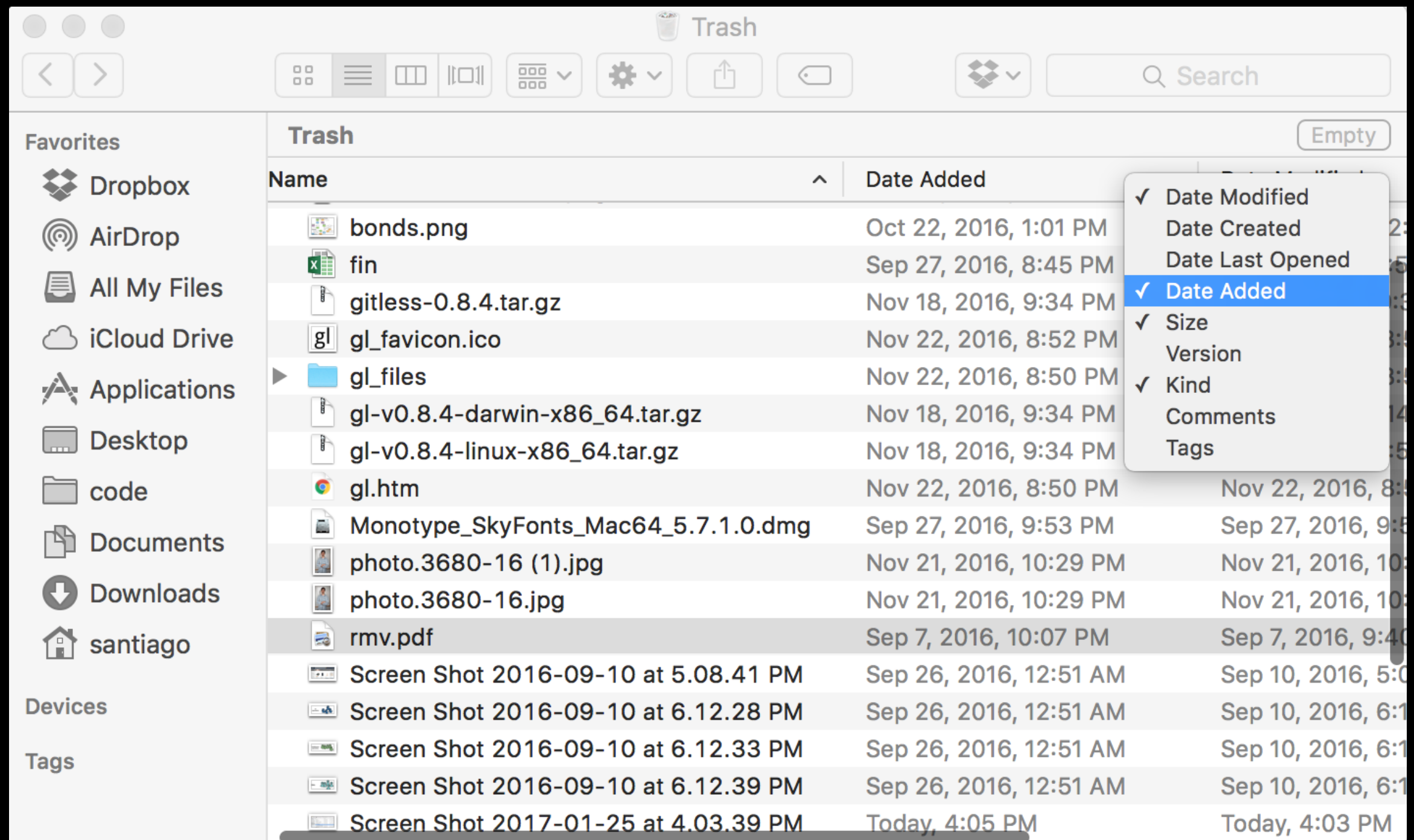
allow deletions
to be undone

An *operational misfit* is a scenario where the concept fails to fulfill purpose

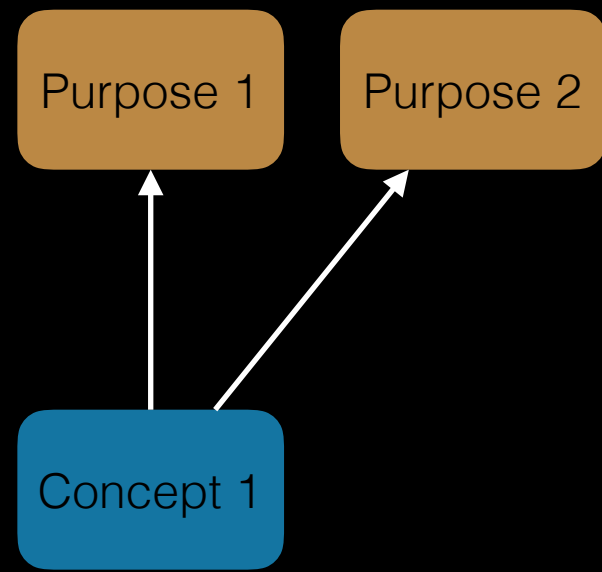


“if the user deletes a file by mistake, and cannot remember the file’s name, there is no easy way to find the file, so it may not be possible to restore it”

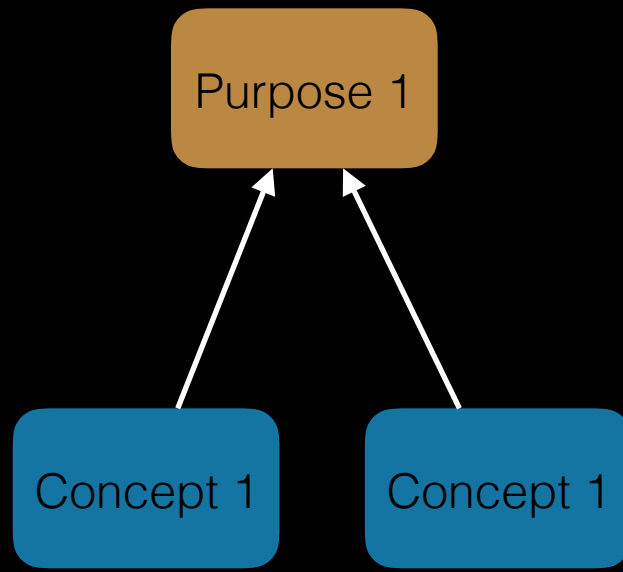
Some misfits are easy to fix...



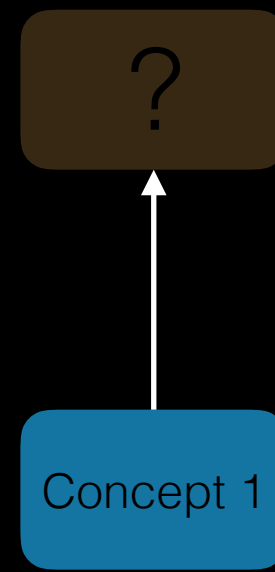
Criteria for concept design



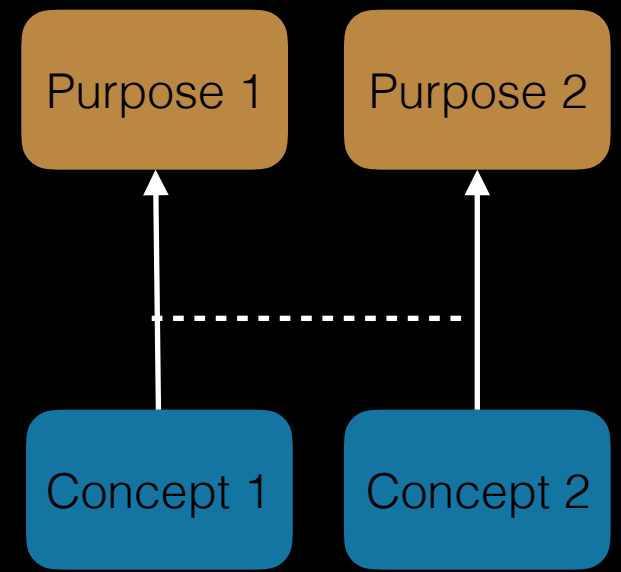
incoherent concept



divided concept



unmotivated concept



coupled concept

Applying the theory to Git

Data Management

Make a set of
changes
persistent

Change Management

Group
logically
related
changes

Record
coherent
points

Collaboration

Synchronize
changes of
collaborators

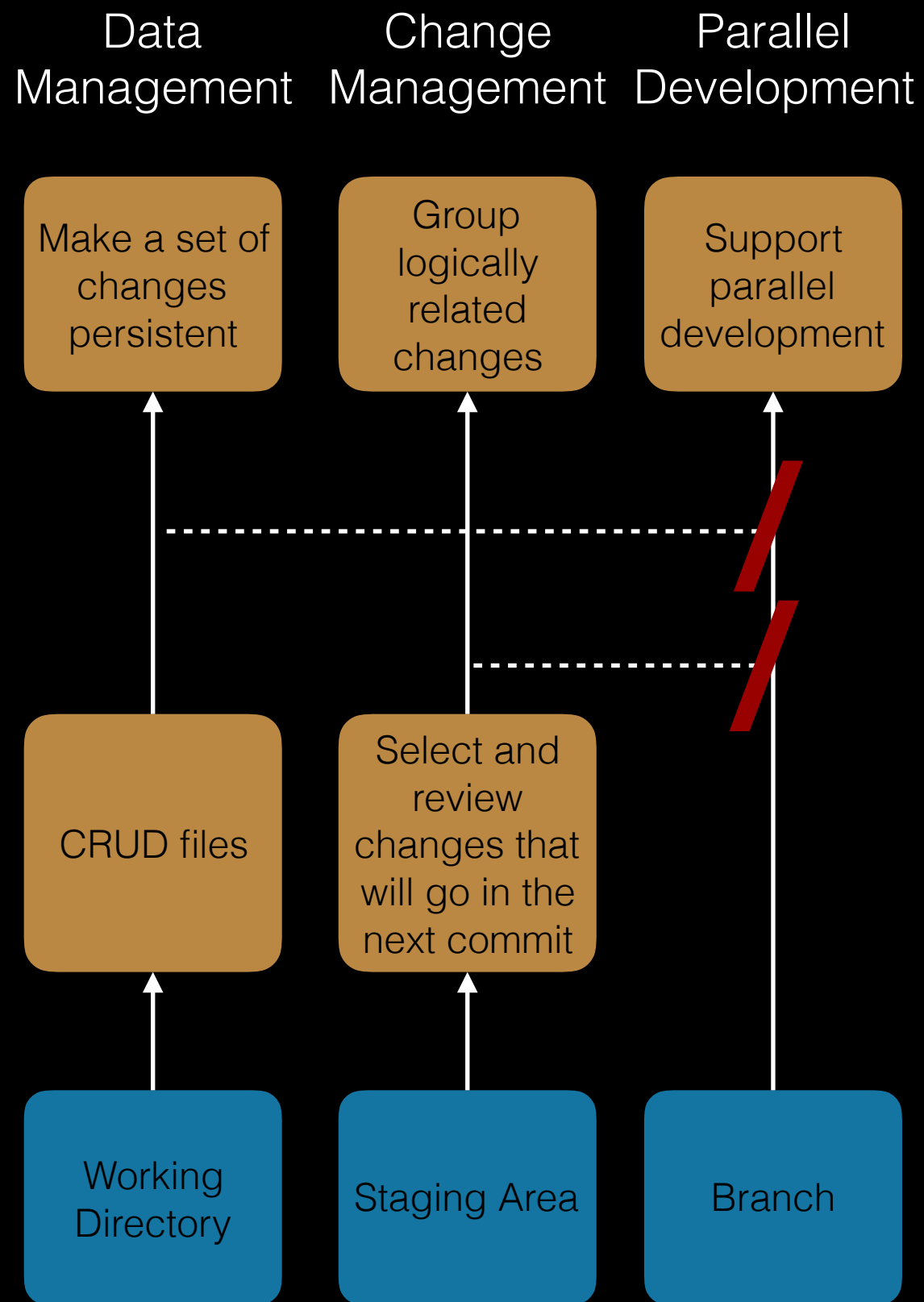
Parallel Development

Support
parallel
development

Disconnected Development

Do work in
disconnected
mode

1. Switching branches



Problem: coupled concept

- ▶ working directory interferes with branching
- ▶ staging area interferes with branching

Misfit: switching branches

- ▶ want to switch to another branch
- ▶ uncommitted changes prevent switch

Stashing: motivating purpose

Stashing: motivating purpose

DESCRIPTION

Use **git stash** when you want to record the current state of the working directory and the index, but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the **HEAD** commit.

man git-stash

Stashing and Cleaning

Often, when you've been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don't want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the `git stash` command.

Stashing takes the dirty state of your working directory – that is, your modified tracked files and staged changes – and saves it on a stack of unfinished changes that you can reapply at any time.

Pro Git (chapter 7.3), Chacon and Straub
<https://git-scm.com/book/>

Stashing: motivating purpose

DESCRIPTION

Use **git stash** when you want to record the current state of the working directory and the index, but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the **HEAD** commit.

man git-stash

Stashing and Cleaning

Often, when you've been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don't want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the `git stash` command.

Stashing takes the dirty state of your working directory – that is, your modified tracked files and staged changes – and saves it on a stack of unfinished changes that you can reapply at any time.

Pro Git (chapter 7.3), Chacon and Straub
<https://git-scm.com/book/>

Stashing: motivating purpose

DESCRIPTION

Use `git stash` when you want to record the current state of the working directory and the index, but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the **HEAD** commit.

`man git-stash`

Stashing and Cleaning

Often, when you've been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don't want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the `git stash` command.

Stashing takes the dirty state of your working directory – that is, your modified tracked files and staged changes – and saves it on a stack of unfinished changes that you can reapply at any time.

Pro Git (chapter 7.3), Chacon and Straub
<https://git-scm.com/book/>

Stashing: motivating purpose

DESCRIPTION

Use **git stash** when you want to record the current state of the working directory and the index, but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the **HEAD** commit.

man git-stash

Stashing and Cleaning

Often, when you've been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don't want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the `git stash` command.

Stashing takes the dirty state of your working directory – that is, your modified tracked files and staged changes – and saves it on a stack of unfinished changes that you can reapply at any time.

Pro Git (chapter 7.3), Chacon and Straub
<https://git-scm.com/book/>

Stashing: motivating purpose

DESCRIPTION

Use **git stash** when you want to record the current state of the working directory and the index, but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the **HEAD** commit.

man git-stash

Stashing and Cleaning

Often, when you've been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don't want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the `git stash` command.

Stashing **takes the dirty state of your working directory** - that is, your modified tracked files and staged changes - **and saves it on a stack of unfinished changes** that you can reapply at any time.

Pro Git (chapter 7.3), Chacon and Straub
<https://git-scm.com/book/>

Clean up and
save
uncommitted
changes

Stash



Data Management

Make a set of changes persistent

Change Management

Group logically related changes

Record coherent points

Collaboration

Synchronize changes of collaborators

Parallel Development

Support parallel development

Disconnected Development

Do work in disconnected mode

?

Clean up and save uncommitted changes

Stash

Data Management

Make a set of changes persistent

Change Management

Group logically related changes

Record coherent points

Collaboration

Synchronize changes of collaborators

Parallel Development

Support parallel development

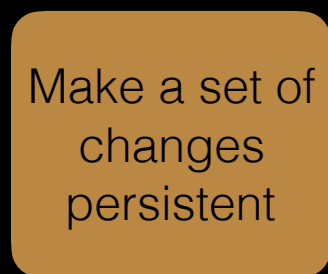
Disconnected Development

Do work in disconnected mode

?

Clean up and save uncommitted changes

Stash



Data Management

Make a set of changes persistent

Change Management

Group logically related changes

Record coherent points

Collaboration

Synchronize changes of collaborators

Parallel Development

Support parallel development

Disconnected Development

Do work in disconnected mode



Data
Management

Make a set of
changes
persistent

Change
Management

Group
logically
related
changes

Record
coherent
points

Collaboration
Parallel
Development

Synchronize
changes of
collaborators

Support
parallel
development

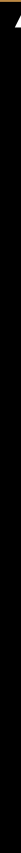
?

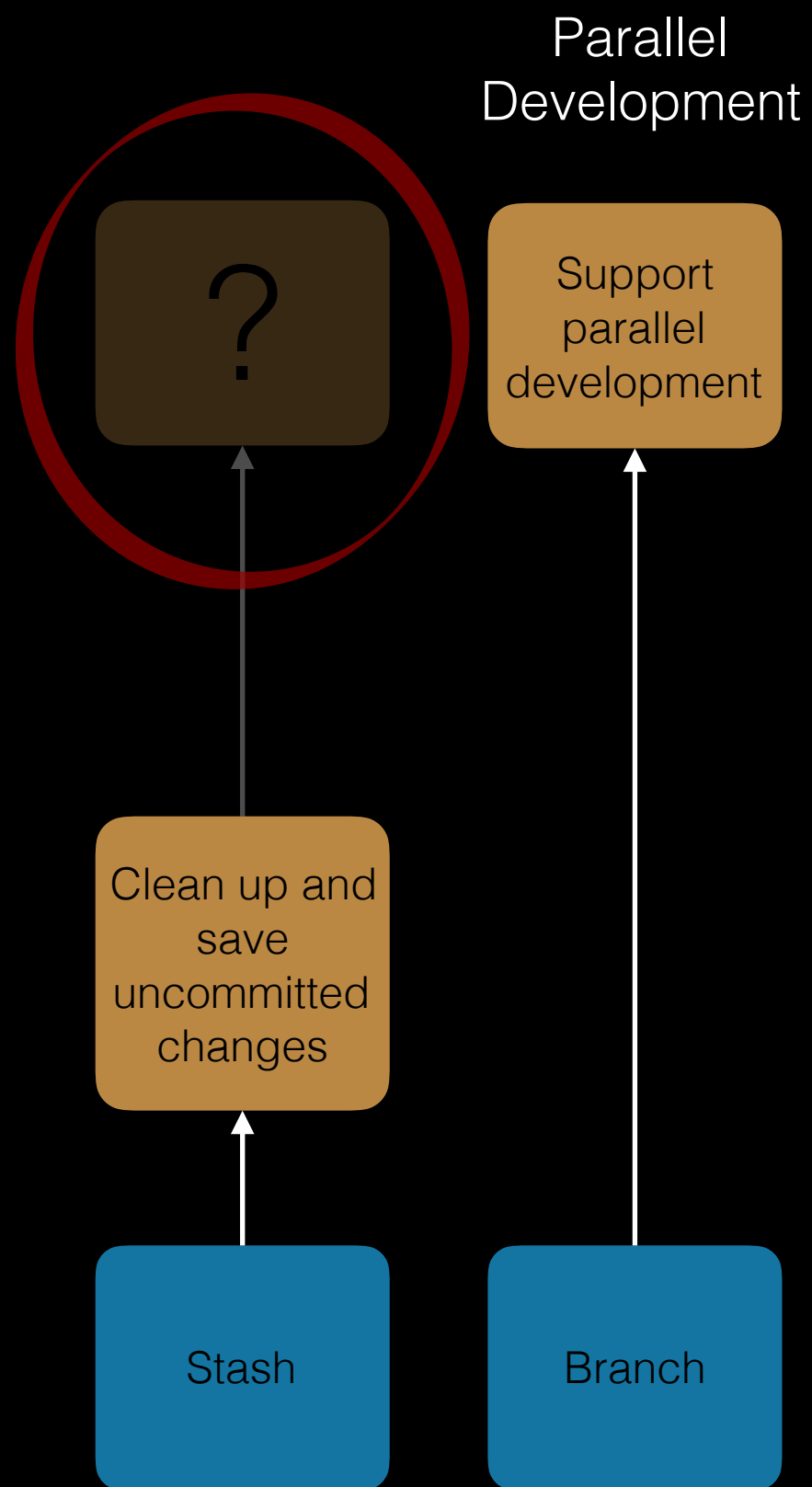
Clean up and
save
uncommitted
changes

Disconnected
Development

Do work in
disconnected
mode

Stash

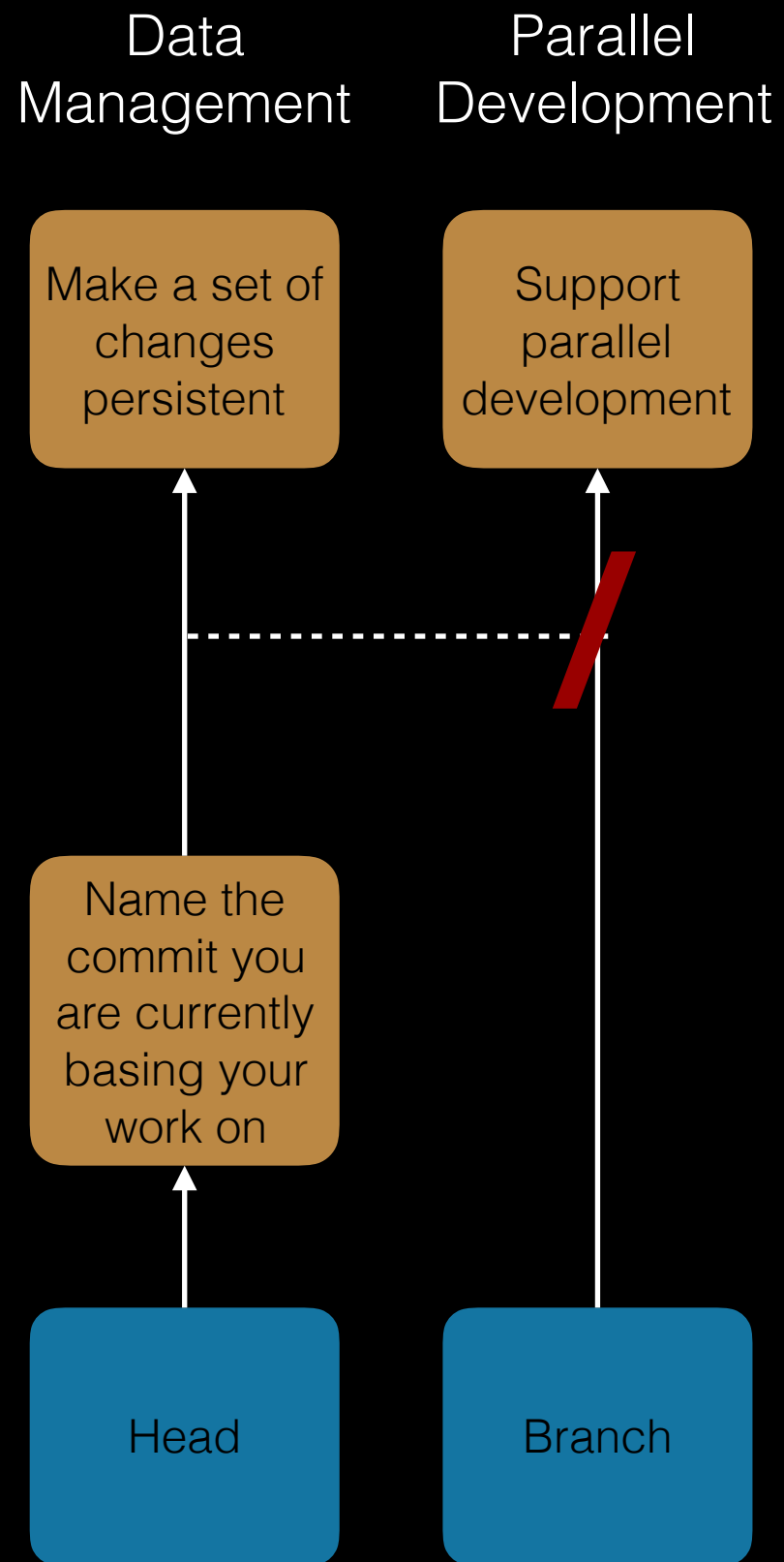




Problem: unmotivated concept

- stashing purpose doesn't map to VC purpose
- addresses misfit in branching

2. Detached head



Problem: coupled concept

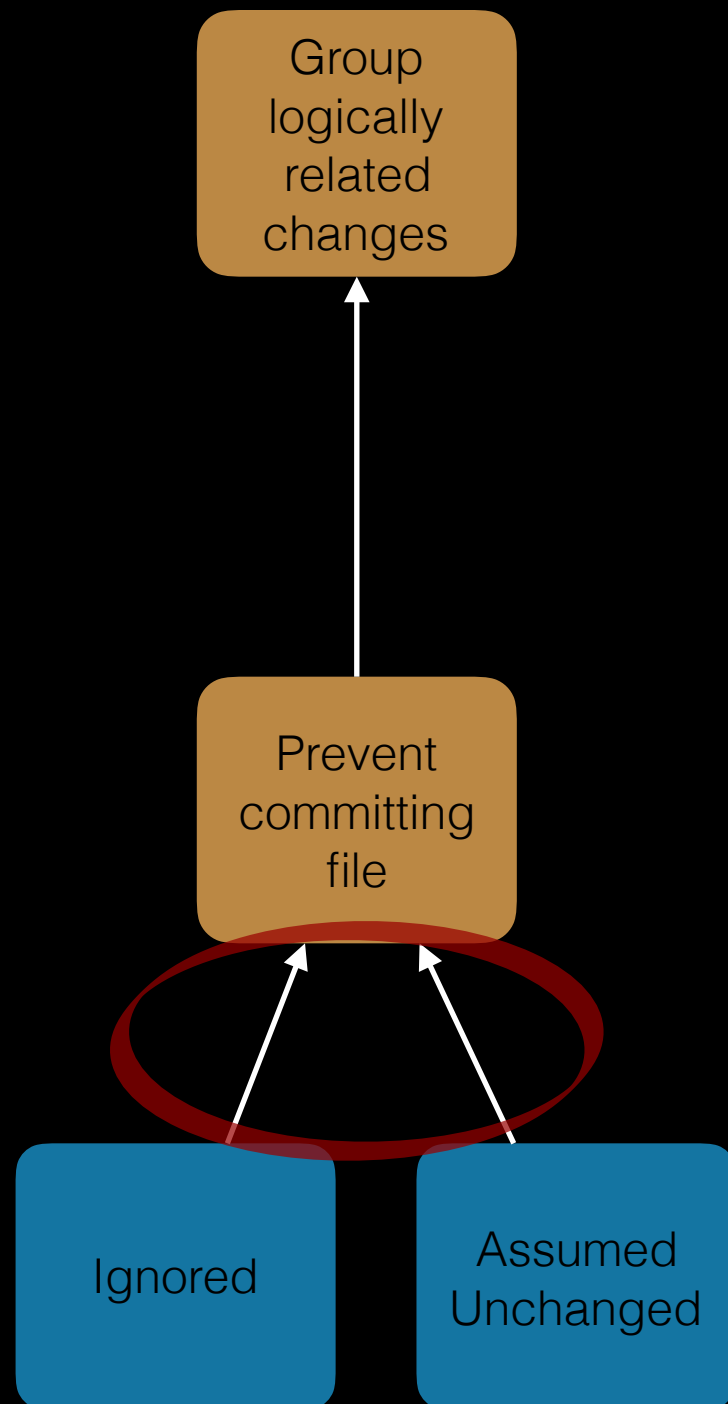
- ▶ head interferes with branching

Misfit: detached head

- ▶ realize that last few commits are wrong
- ▶ checkout old commit to start over again
- ▶ create new commits
- ▶ hard to switch from/to this line

3. Untracking file

Change Management



Problem: divided concept

- ▶ two concepts with same purpose

Misfit: untracking file

- ▶ want to ignore committed file
- ▶ .gitignore doesn't work
- ▶ need to mark file as assume unchanged

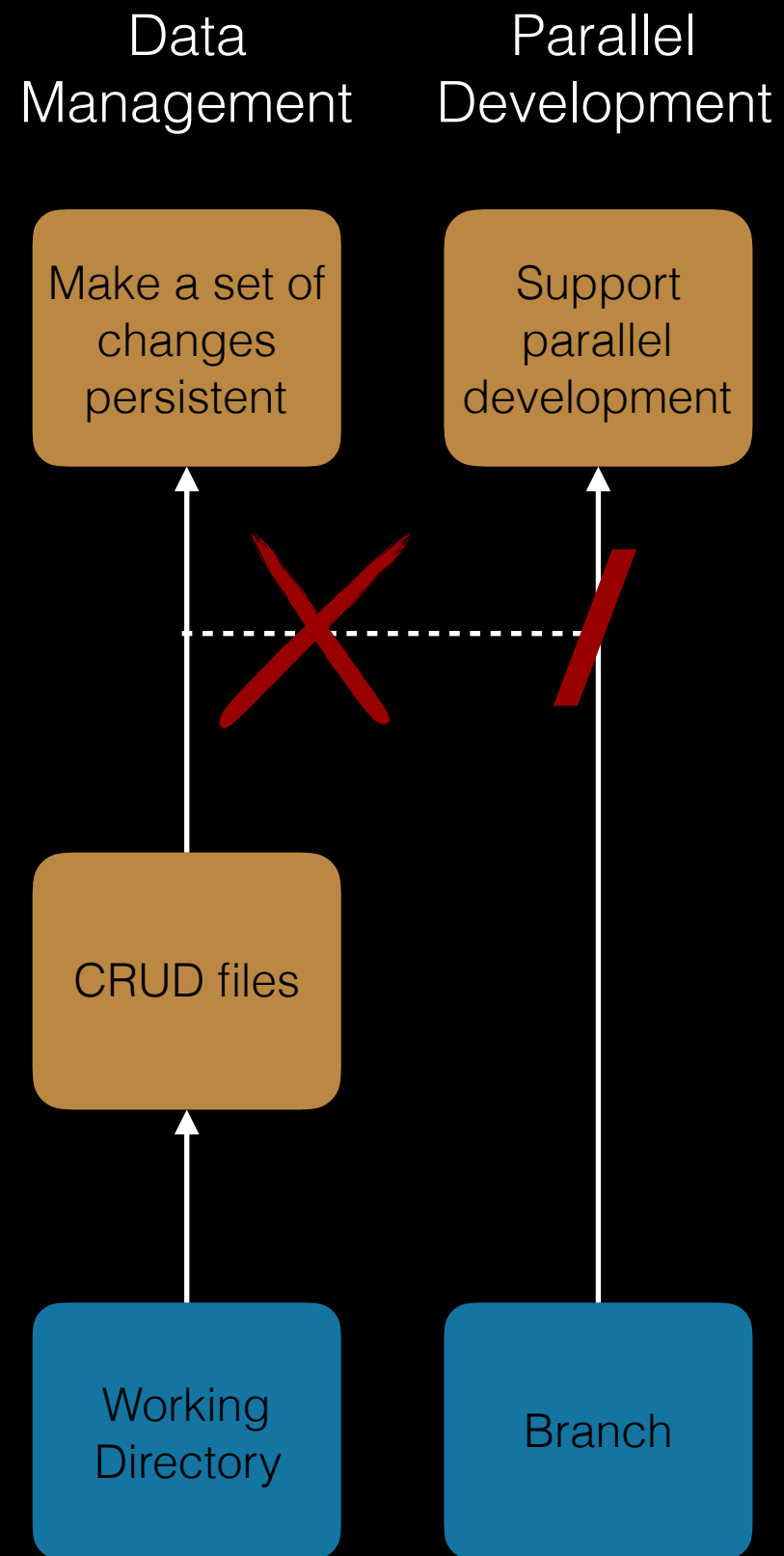
Gitless

a simple VCS built on top of Git

Gitless

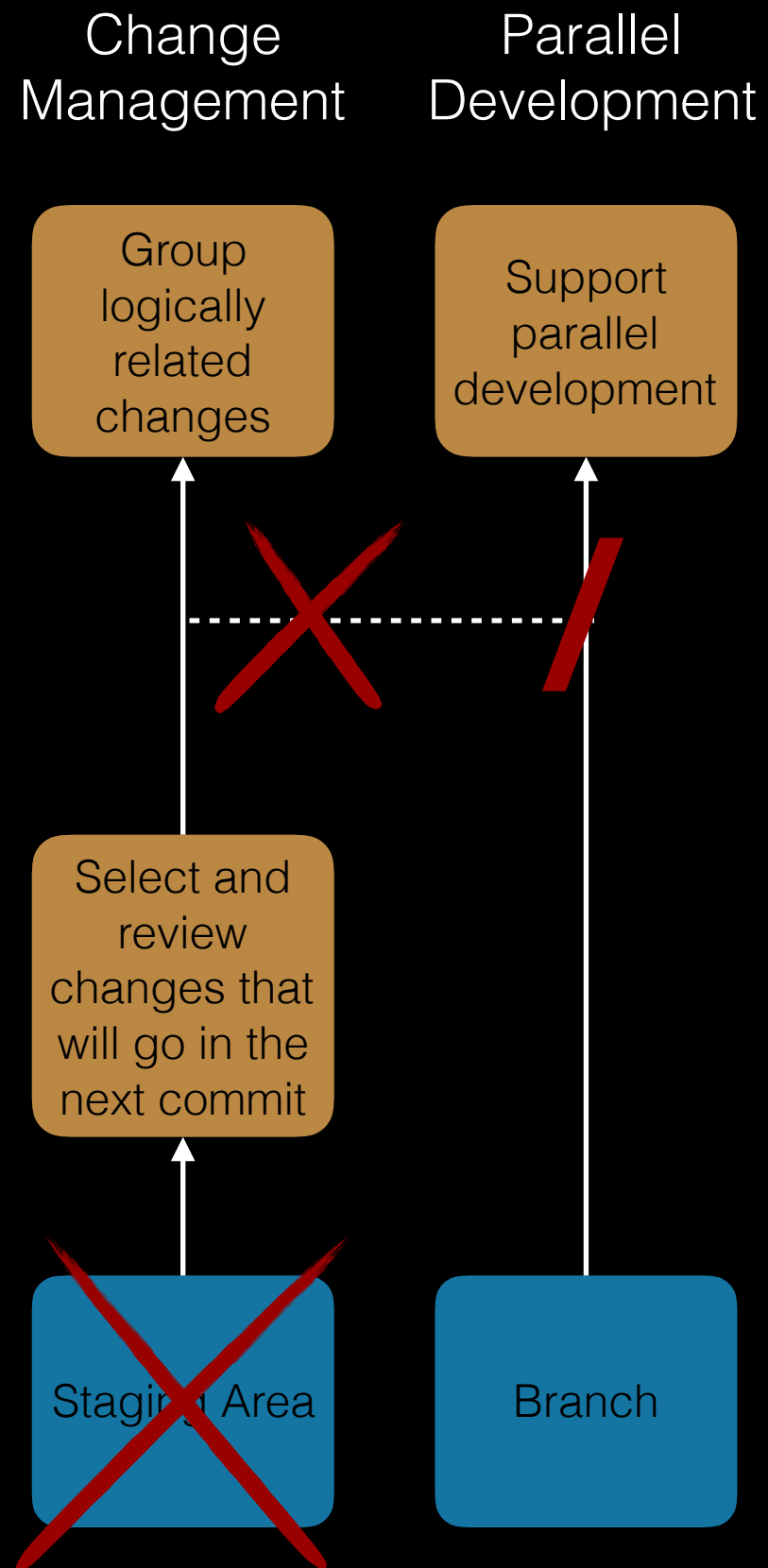
- ▶ VCS built on top of Git
- ▶ Presents different concept model to the user
- ▶ An experiment!

1. Switching branches



Branches include working dir

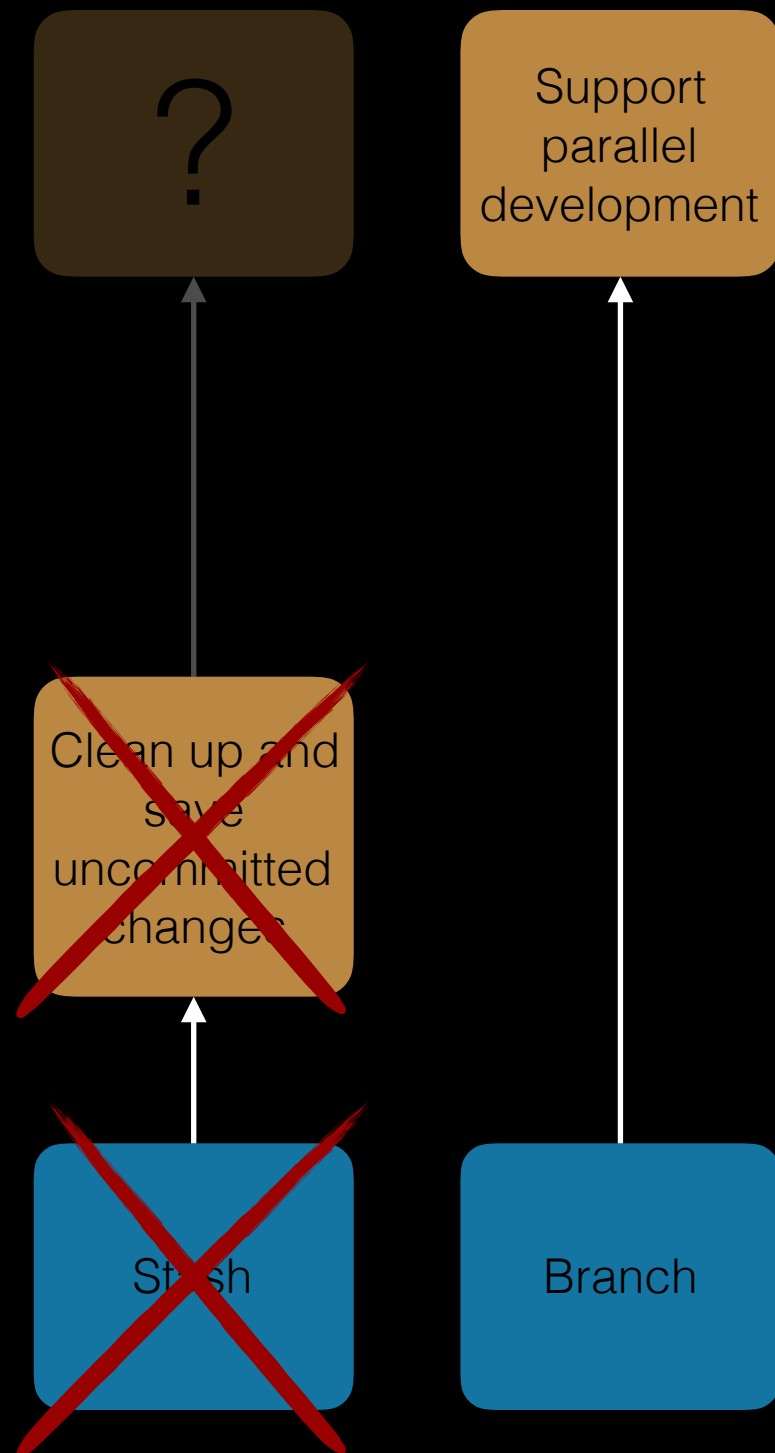
- uncommitted changes can't prevent switch
- can switch in the middle of a merge



Removed staging area

- ▶ staged contents can't prevent switch
- ▶ more flexible commit command

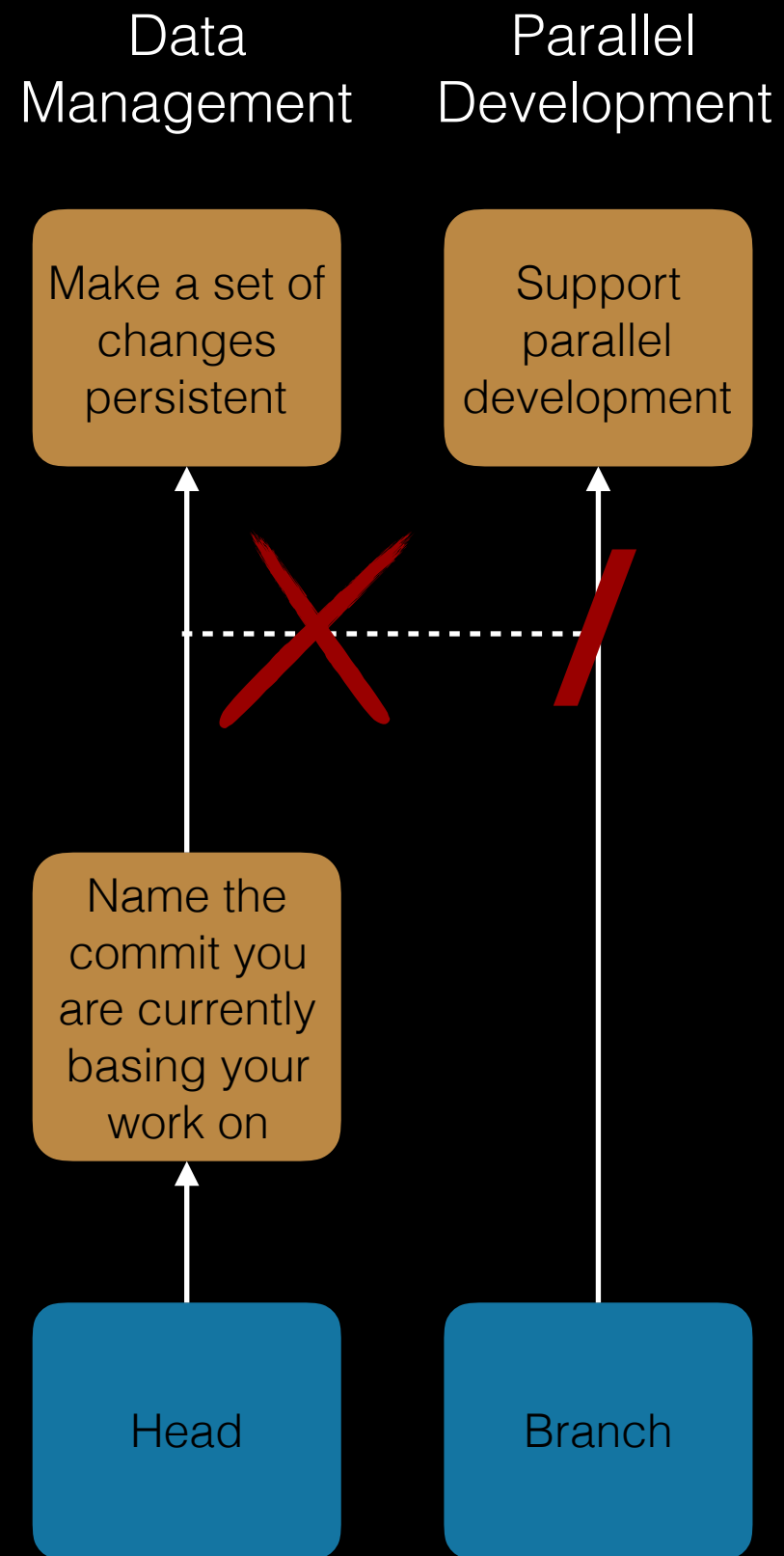
Parallel Development



Removed stash

- less need for stashing

2. Detached head

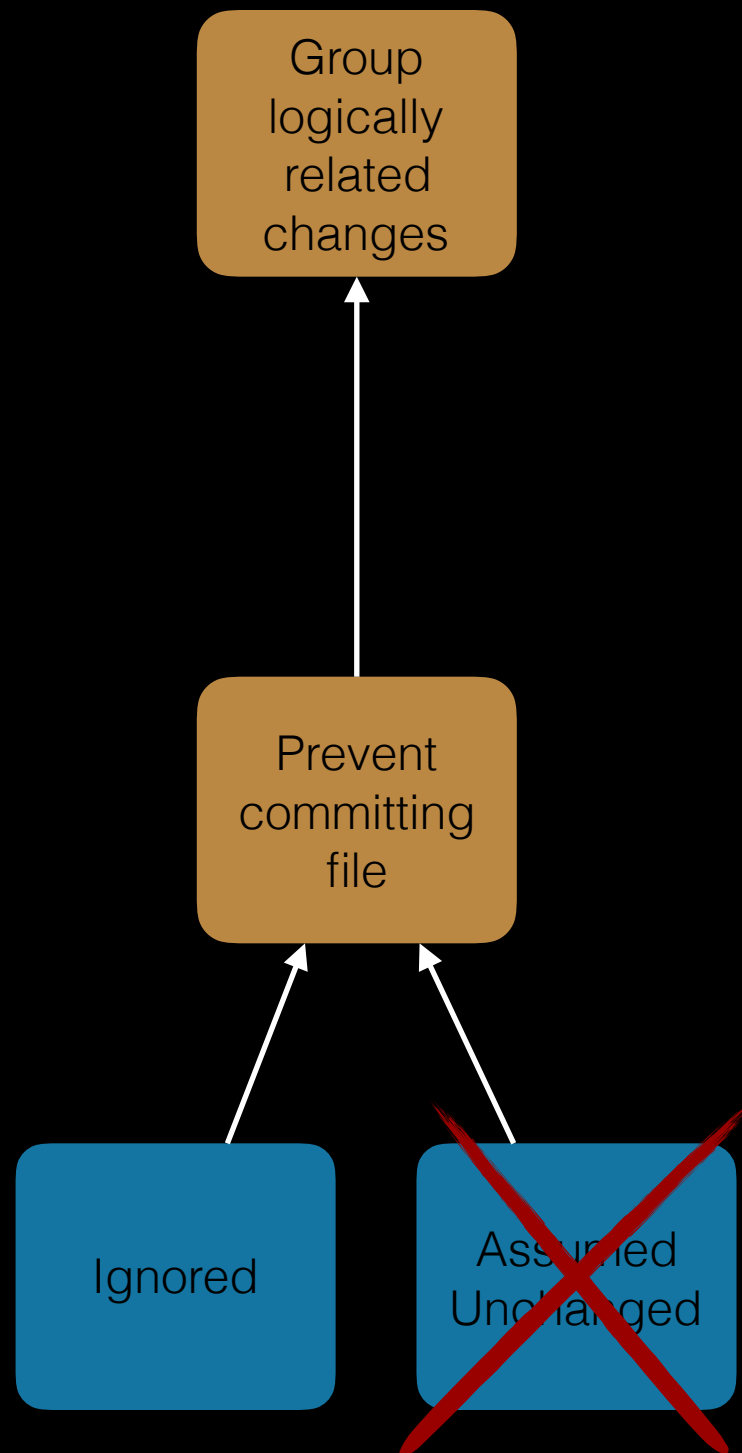


Head is a per-branch reference

- ▶ each branch has a head
- ▶ can't go into a detached head state

3. Untracking file

Change Management



Removed assumed unchanged

- ▶ committed files can be ignored or untracked

User study

Experiment design

- ▶ 2 sessions (Git, Gitless) of ~1 hour each
- ▶ six tasks per session (+ 1 practice task)
- ▶ survey after session + final survey

Participants

- ▶ 11 = 3 industry + 3 research + 5 student
- ▶ Git: 4 novices, 3 regular, 4 experts
- ▶ Gitless: none used before

Tasks

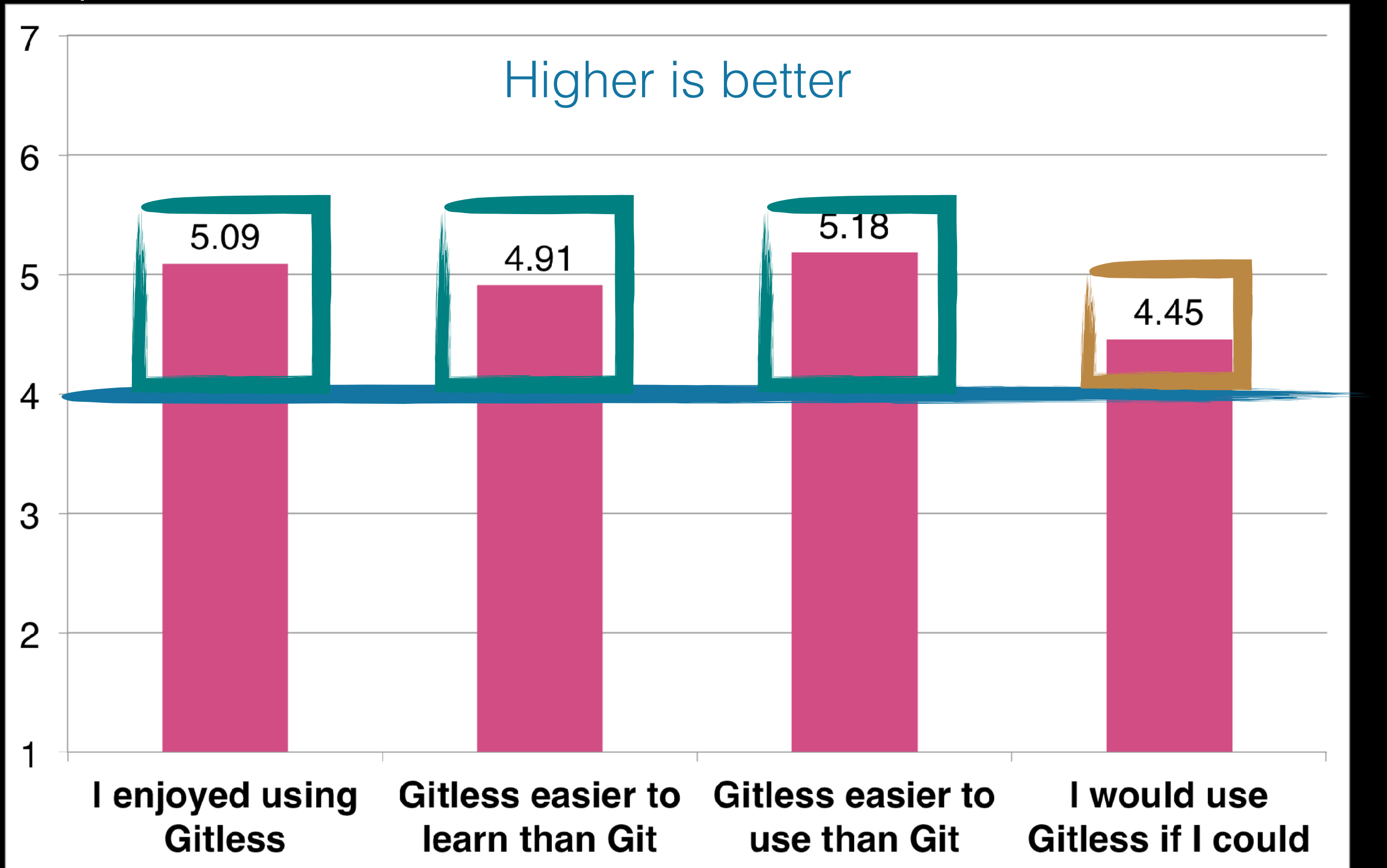
- ▶ commit staged modified file
- ▶ create and switch to branch
- ▶ switch with changes that conflict
- ▶ switch leaving changes behind
- ▶ switch in the middle of merge
- ▶ undo commit

Measures

- ▶ task success rate and completion time
- ▶ satisfaction, efficiency, difficulty, confusion and frustration
- ▶ Git vs Gitless

Post-study questionnaire results

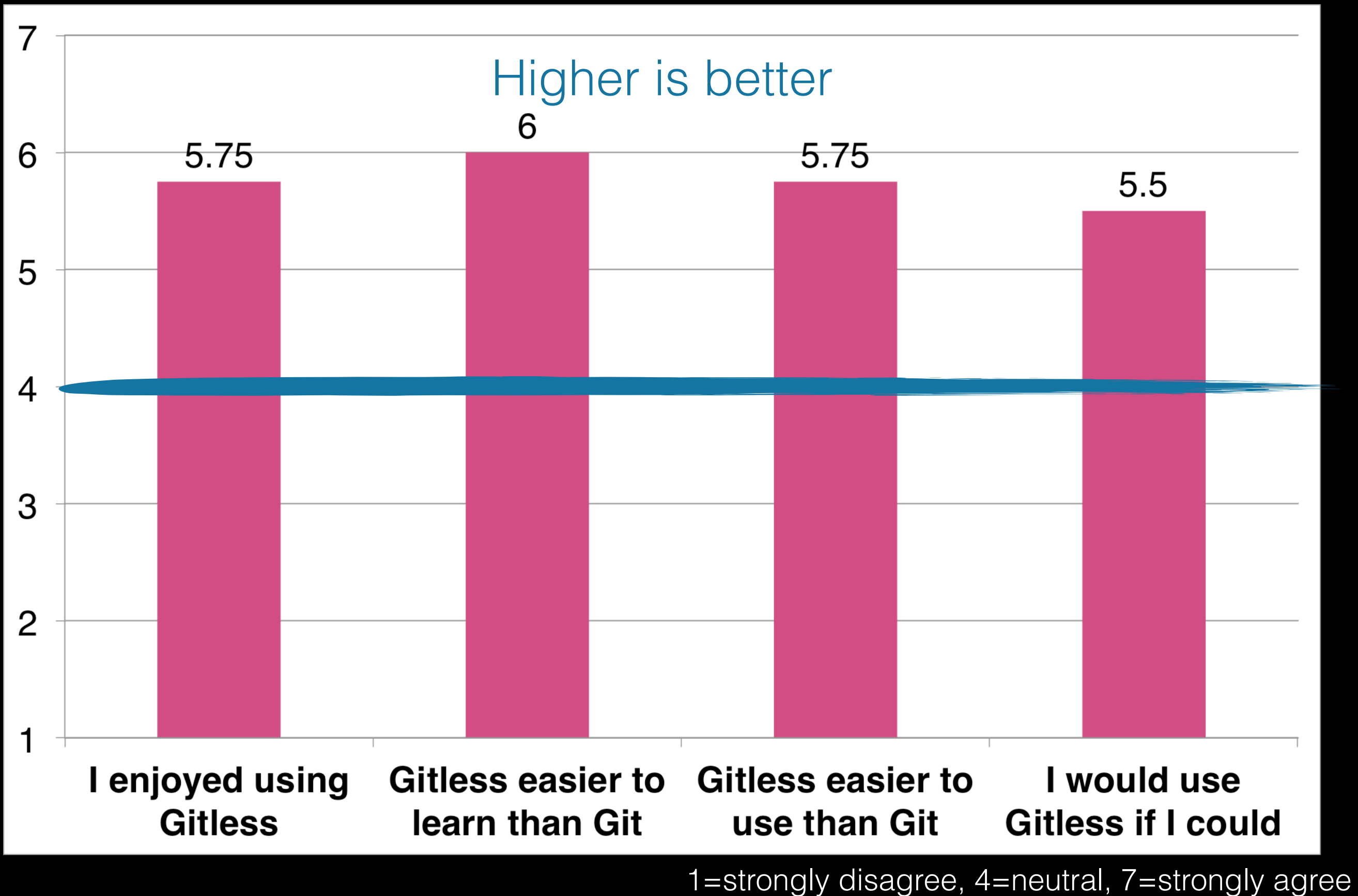
All proficiencies



1=strongly disagree, 4=neutral, 7=strongly agree

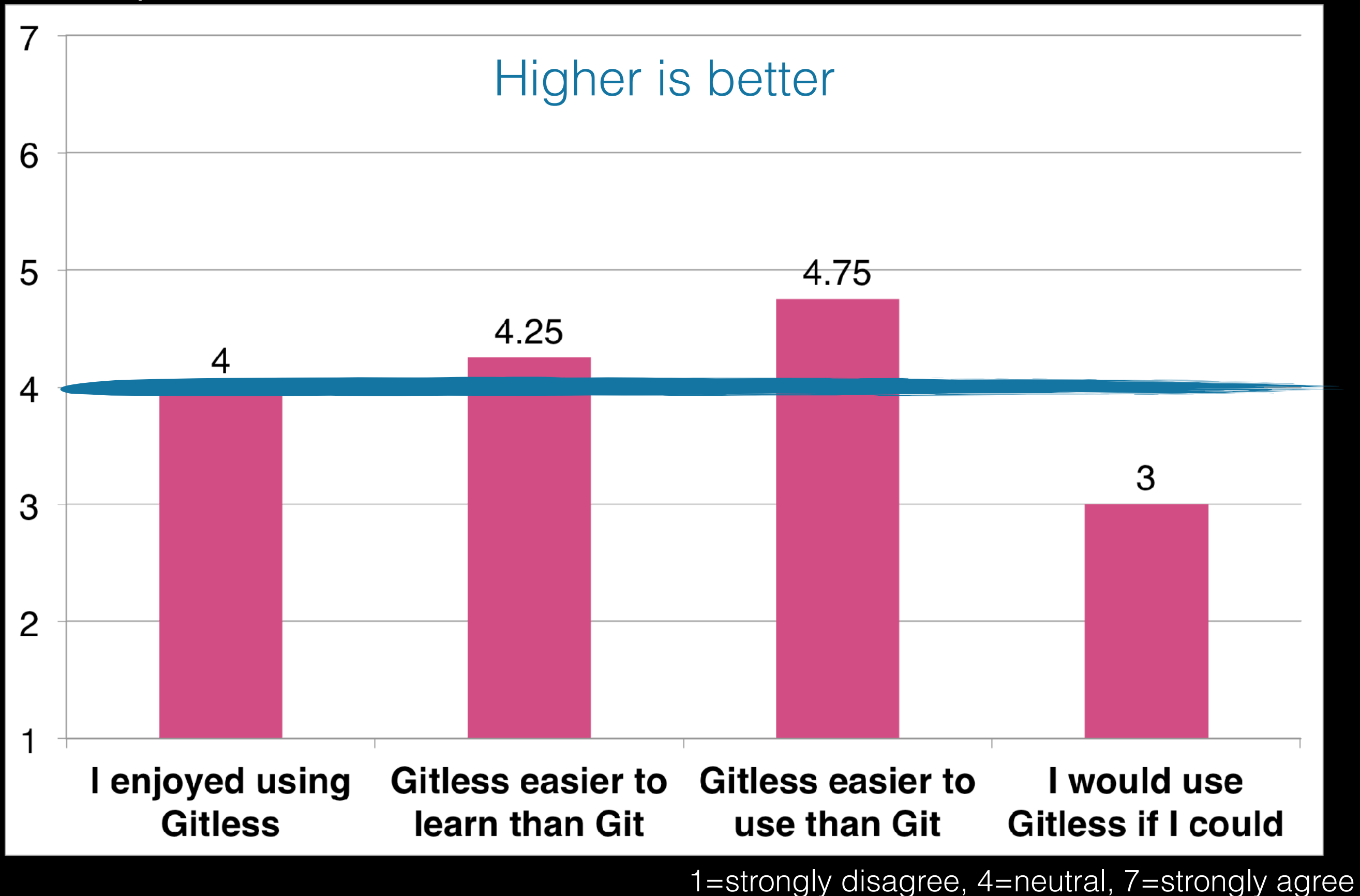
Post-study questionnaire results

Git novices



Post-study questionnaire results

Git experts



User study

this doesn't mean Gitless is a better VCS than Git

study focused on misfits and did so in a controlled environment

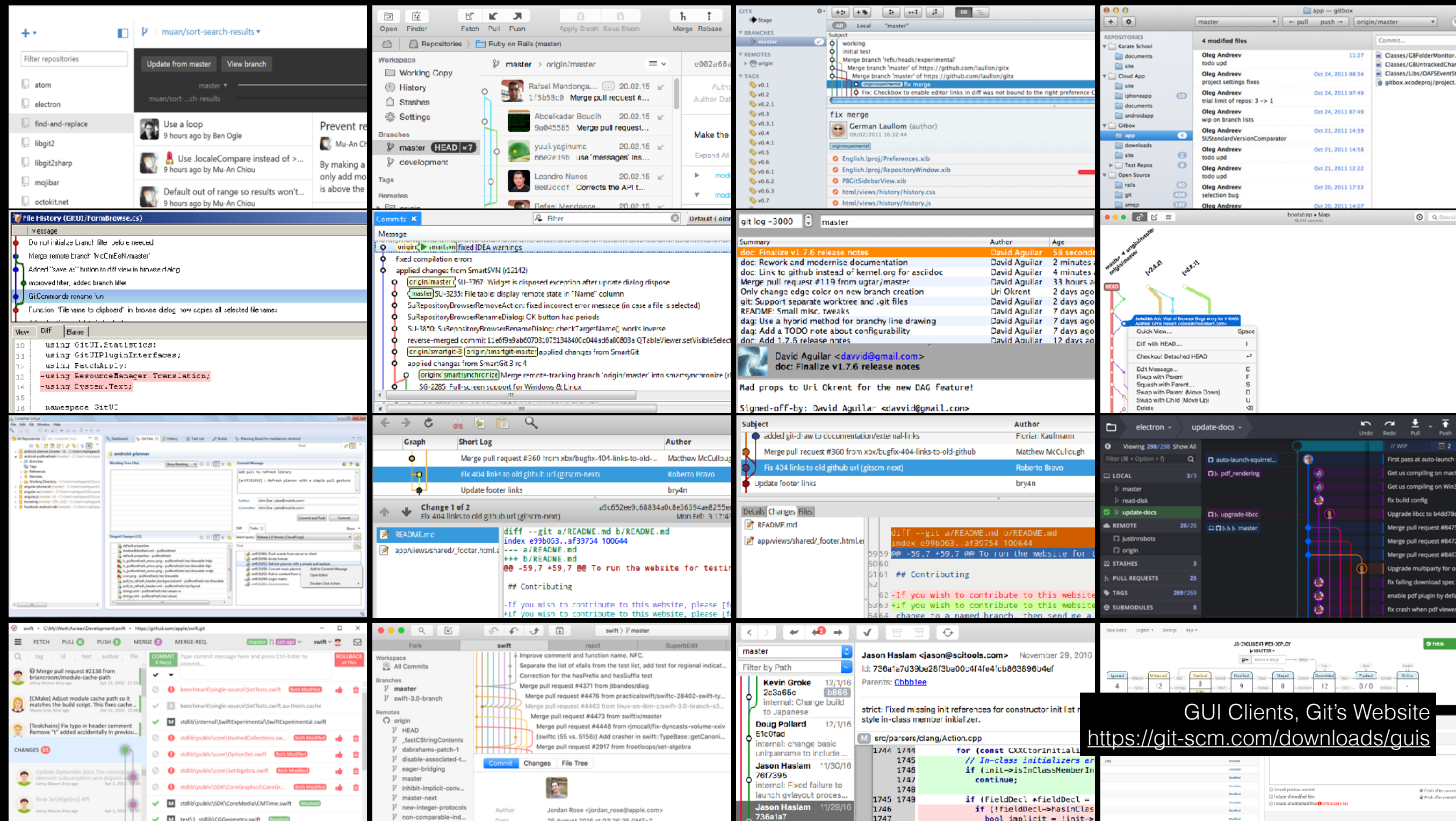
but it suggests that our approach can be useful

redesigning concepts could make Git easier to learn and use

Opportunity

use the theory to guide the design of new Git-compatible VCSs

Many Git clients



GUI Clients, Git's Website
<https://git-scm.com/downloads/guis>

Many Git clients

```
(Run 'eg help --all' for a more detailed list.)

Creating repositories
eg clone      Clone a repository into a new directory
eg init      Create a new repository

Obtaining information about changes, history, & state
eg diff      Show changes to file contents
eg log       Show history of recorded changes
eg status    Summarize current changes

Making, undoing, or recording changes
eg commit    Record changes locally
eg mv        Move or rename files (or directories or symlinks)
eg revert    Revert local changes and/or changes from previous commits
eg stage     Mark content in files as being ready for commit

Managing branches
eg branch    List, create, or delete branches
eg merge     Join two or more development histories (branches) together
eg switch    Switch the working copy to another branch

Collaboration
eg pull      Get updates from another repository and merge them
eg push     Push local commits to a published repository

Time saving commands
eg bisect    Find the change that introduced a bug by binary search
eg stash     Save and revert local changes, or apply stashed changes

Additional help:
eg help COMMAND      Get more help on COMMAND.
eg help --all        List more commands (not really all)
eg help topic        List specialized help topics.

(Detailed list of differences between eg and git)
```

The Interface

```
branches
  Get a nice pretty list of available branches.

sync [<branch>]
  Synchronizes the given branch. Defaults to current branch. Stash, Fetch, Auto-Merge/Rebase, Push, and
  Unstash. You can only sync published branches. (alias: sy )

resync <upstream-branch>
  Stashes unstaged changes, Fetches, Auto-Merge/Rebase upstream data from specified upstream branch,
  Performs smart pull+merge for current branch, Pushes local commits up, and Unstashes changes. Default
  upstream branch is 'master'. (alias: rs )

switch <branch>
  Switches to specified branch. Defaults to current branch. Automatically stashes and unstashes any changes.
  (alias: sw )

sprout [<branch>] <new-branch>
  Creates a new branch off of the specified branch. Switches to it immediately. (alias: sp )

harvest [<branch>] <into-branch>
  Auto-Merge/Rebase of specified branch changes into the second branch. (alias: ha , hv , har )

graft <branch> <into-branch>
  Auto-Merge/Rebase of specified branch into the second branch. Immediately removes specified branch. You can
  only graft unpublished branches. (alias: gr )

publish [<branch>]
  Publishes specified branch to the remote. (alias: pub )

unpublish <branch>
  Removes specified branch from the remote. (alias: unp )

install
  Installs legit git aliases.

help
  Displays help for legit command. (alias: h )
```

Commands:

```
----
W help          Display help for darcs or a single commands.
----

Changing and querying the working copy:
----
W add           Add one or more new files or directories.
W remove        Remove one or more files or directories from the repository.
W mv            Move/rename one or more files or directories.
N replace       Replace a token with a new value for that token.
Y revert        Revert to the recorded version (safe the first time only).
N unrevert      Undo the last revert (may fail if changes after the revert).
Y whatnew       Display unrecorded changes in the working copy.
----

Copying changes between the working copy and the repository:
----
Y record        Save changes in the working copy to the repository as a patch.
A unrecord      Remove recorded patches without changing the working copy.
Y amend-record  Replace a patch with a better version before it leaves your
repository.
N resolve       Mark any conflicts to the working copy for manual resolution.
----

Direct modification of the repository:
----
Y tag           Tag the contents of the repository with a version name.
A setpref       Set a value for a preference (test, predist, ...).
A rollback      Record an inverse patch without changing the working directory.
----

Querying the repository:
----
W diff          Create a diff between two versions of the repository.
```

and your custom aliases!

But mostly cosmetic changes

- ▶ same concepts, different presentation:
 - ▶ more attractive interfaces
 - ▶ more consistent terminology
 - ▶ focus on more commonly used workflows

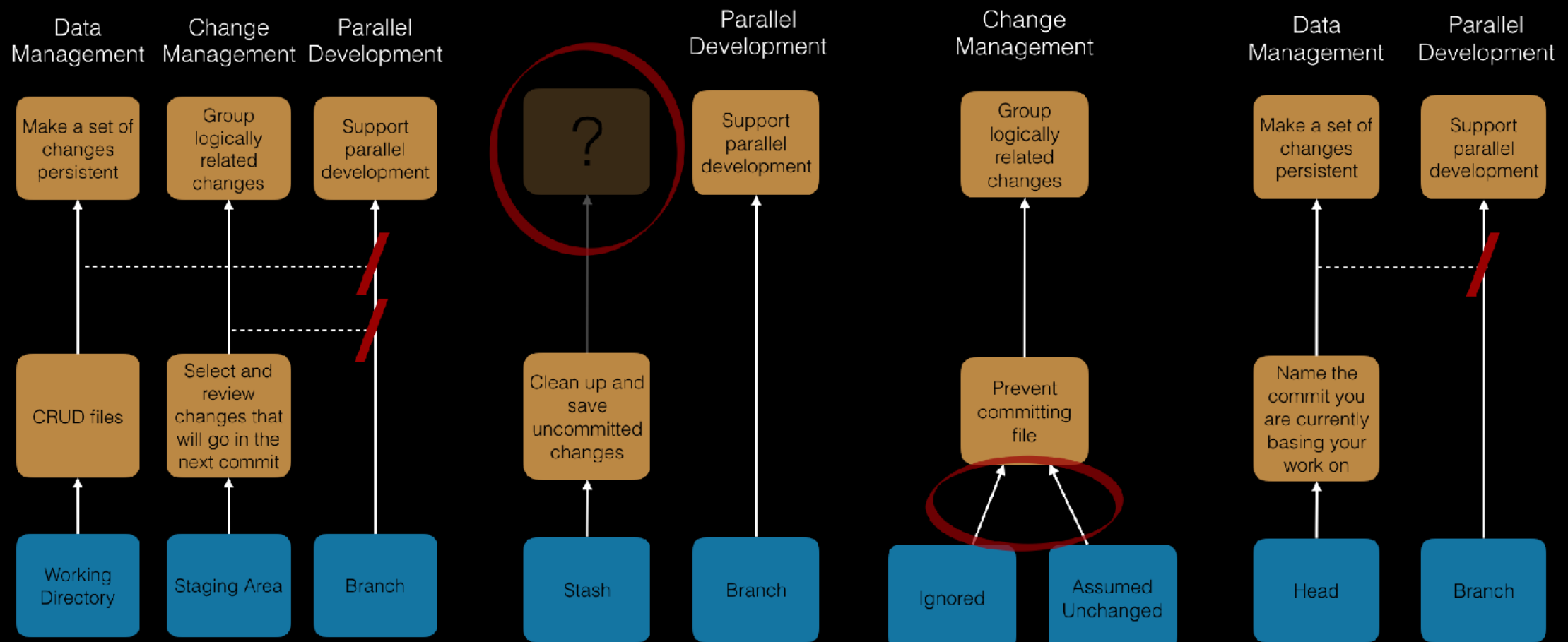
Need to go deeper

- ▶ different concepts, Git-compatible:
 - ▶ new VCSs that look very different to Git
 - ▶ domain-specific Gits

What's wrong with Git?

What's wrong with Git?

concepts!





Thank you!

- ▶ to try gitless visit gitless.com
- ▶ read our paper at tinyurl.com/gitless-paper