Advanced Git Topics

Join git_training_313 slack channel for links + discussion

Instructor: Daniel Nemergut

Helpers: Matthew Bourque, James Davies

Outline

- 10 - 12
  - Quick recap
  - External tools
  - .gitignore tips
  - Branches and tags
  - Branching models
  - Moving things
- 12 – 1
  - Lunch
- 1 – 2
  - Questions/common problems
  - Undoing actions
  - Dealing with large files
  - Bisect
  - Hooks

Quick Recap

- *init* – Start a repo
- *add* – Add files to staging
- *commit* – Commit staging
- *push* – Push commits to repo
- *pull* – Pull changes from repo
- *log* – View commit summaries
- *squash* – Combine commits
- *stash* – Store changes
- *rebase* – Move commits to another base
- *clone* – Copy a repository

Clone This…

git clone git@github.com:dpnemergut/branching-example.git

Visualization Tools

- **gitk (comes with git)**
- SourceTree (Windows, Mac)
- GitHub Desktop (Windows, Mac)
- Git Extensions (Windows, Mac, Linux)
- GitKraken (Windows, Mac, Linux)

Advise to only use for visualization (may not support merge conflicts)

Using gitk

- In your repository, run `` `*gitk*` ``
  - Windows users may need to make sure C:\Program Files (x86)\Git\bin\gitk or C:\Program Files (x86)\Git\cmd\gitk.cmd is in your PATH

- View history for one file with `` `*gitk <filename>*` ``

Merge Tools

Used for graphical merge conflict resolution

- **kdiff3**
- Meld (Windows, Linux)
- P4Merge (Windows, Mac, Linux)
- opendiff (Mac)
- vimdiff (for vim lovers)

Using Merge Tools

- Install kdiff3
  - http://kdiff3.sourceforge.net/
- Configure merge tool
  - *git config --global merge.tool kdiff3*
  - **Mac/Linux:** *git config --global mergetool.kdiff3.path '/Applications/kdiff3.app/Contents/MacOS/kdiff3*
  - **Windows:** *git config --global mergetool.kdiff3.cmd '"C:\\Program Files (x86)\\KDiff3\\kdiff3" $BASE $LOCAL $REMOTE -o $MERGED'*
- Cause a conflict
  - *git checkout master*
  - *git merge conflict_branch*
- Open merge tool to resolve conflicts
  - *git mergetool*
- Mark conflicts as being resolved and commit the merge
  - *git merge --continue*
- Can abort merge if things aren't going well
  - *git merge --abort*

Command Prompt Hints

For bash users,

Mac/Linux:

Add these lines to the end of *~/.bashrc:*

```
parse_git_branch() {
        git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/’
}
export PS1="\w\[\033[33m\] \$(parse_git_branch)\[\033[00m\] $ "
```

Windows:

```
cp C:/Program Files/Git/etc/profile.d/git-completion.bash ~/
cp C:/Program Files/Git/etc/profile.d/git-prompt.sh ~/
```

Add these lines to the end of *~/.bashrc:*

```
. git-completion.bash
. git-prompt.sh
GIT_PS1_SHOWDIRTYSTATE=true
PS1='\w\[\033[01;32m\]$(__git_ps1)\[\033[00m\]\$ '
```

.gitignore Tips

Things to put in a gitignore:

- Compiled files (.exe, .class, .pyc)
- Large files (.pdf)
- Configuration/password files (commit a default that can be copied)
- Packaged/build files (build/, .zip, .tar)
- Logs (.log)
- Database files (.sql)
- OS generated files (.DS_Store, .Trashes)
- IDE/editor files (\#*\#, .idea)

Things suggested to put in a gitignore:

- Data files (.fits)
- Merge conflict backups (.orig)

Things to not put in a gitignore:

- .gitignore
- .git/

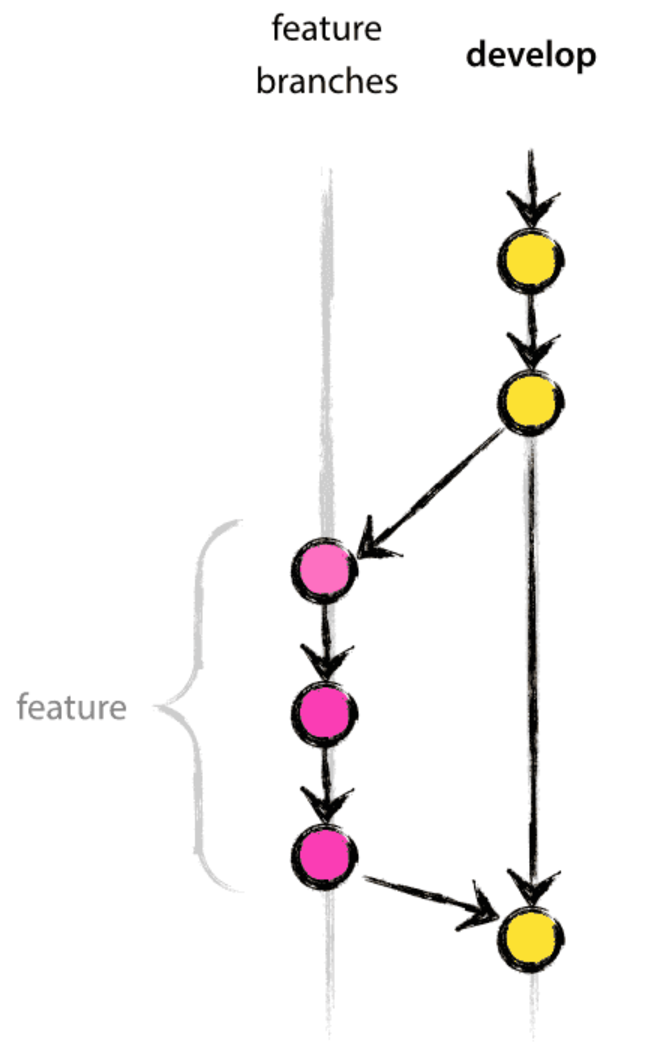
Different Merges

Fast-forward

- Merges branch without merge commit
- Aborts merge if it can't be done
- Ideal for updating a branch with remote
- *git merge --ff-only <branch>*

Non fast-forward

- Merges branch by creating merge commit
- Prompts for merge commit message
- Ideal for merging two branches
- *git merge --no-ff <branch>*



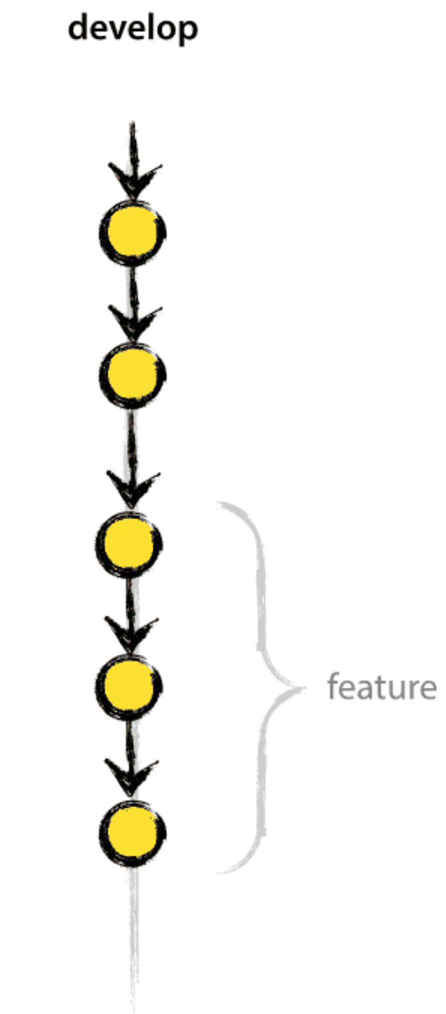


Image from nvie.com/posts/a-successful-git-branching-model/
by Vincent Driessen

Tags

Tags mark a point in history that you can return to

Tied to commits but won’t result in a headless state when checking them out

To create a tag:

git tag -a <tag> <SHA> -m <message>

To checkout a tag:

git checkout <tag>

To list tags:

git tag

To push tags:

git push --tags

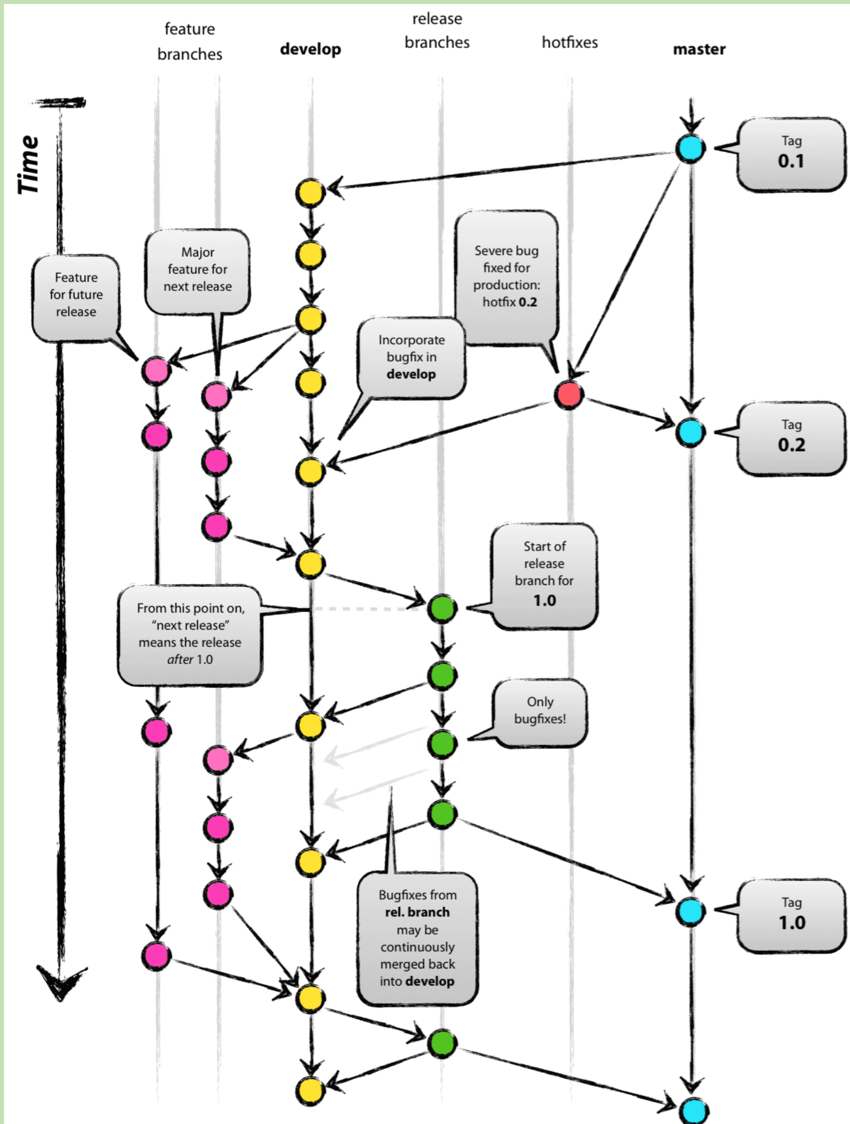
To delete tags:

git tag –d <tag>

Clone This…

git clone git@github.com:dpnemergut/gitflow-example.git

Gitflow Branching Model



Standard for released software

Focused on keeping master stable while doing parallel work

Builds up releases and merges them to master

Leaves room for hotfixing master

Directory Structure

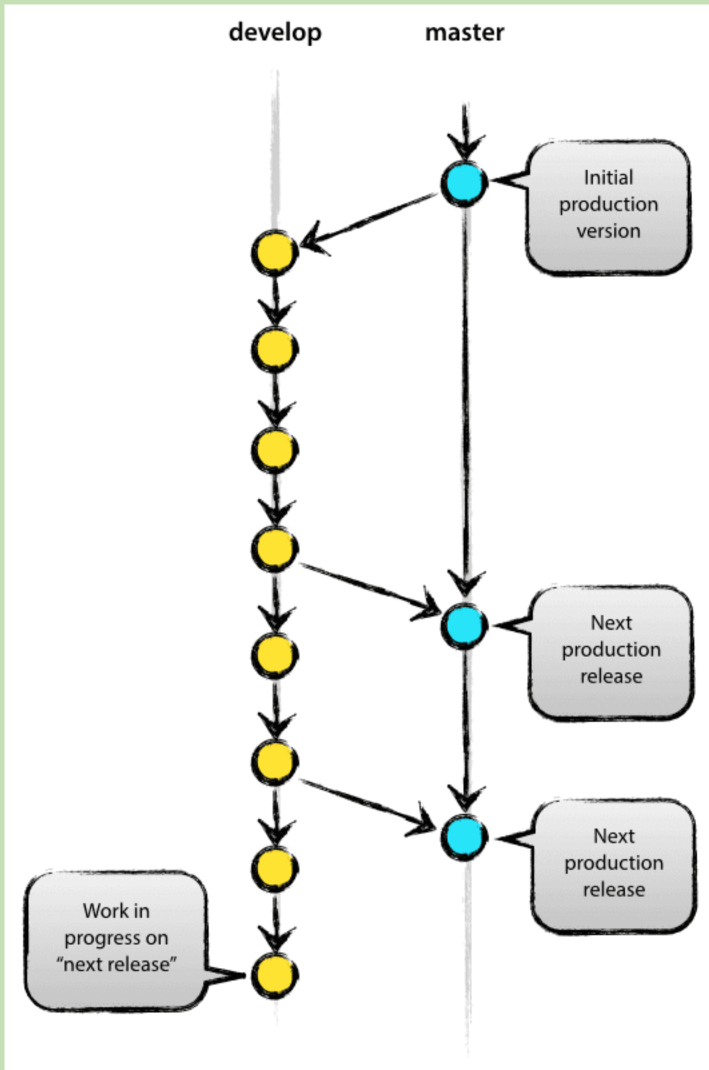
Structure isn't terribly important vs workflow

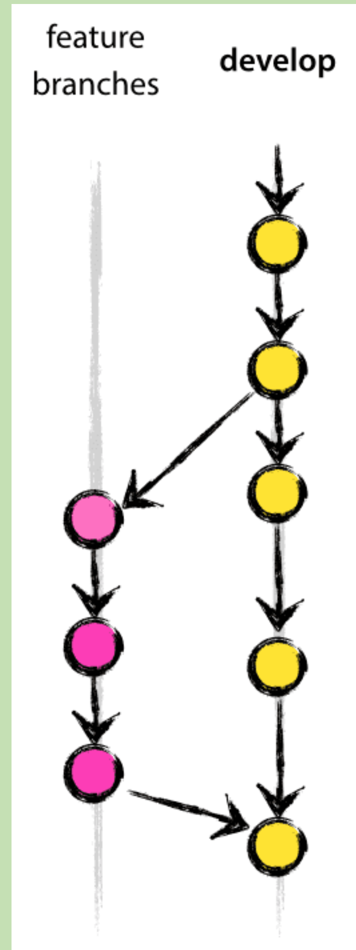
Keep compiled files in a single, ignored place

Ignore configuration files

```
src/
  - code/
  - resources/
tests/
  - code/
  - resources/
libraries/
build/
```

Feature Workflow





Users only get what's on master (released versions)

Develop is the next release, based on master

With multiple developers, feature branches minimize merge commits and conflicts

Making Feature Branches

Create feature branch

git checkout -b myfeature develop

Commit work on feature branch

Update with develop

git checkout develop

git pull develop

git checkout myfeature

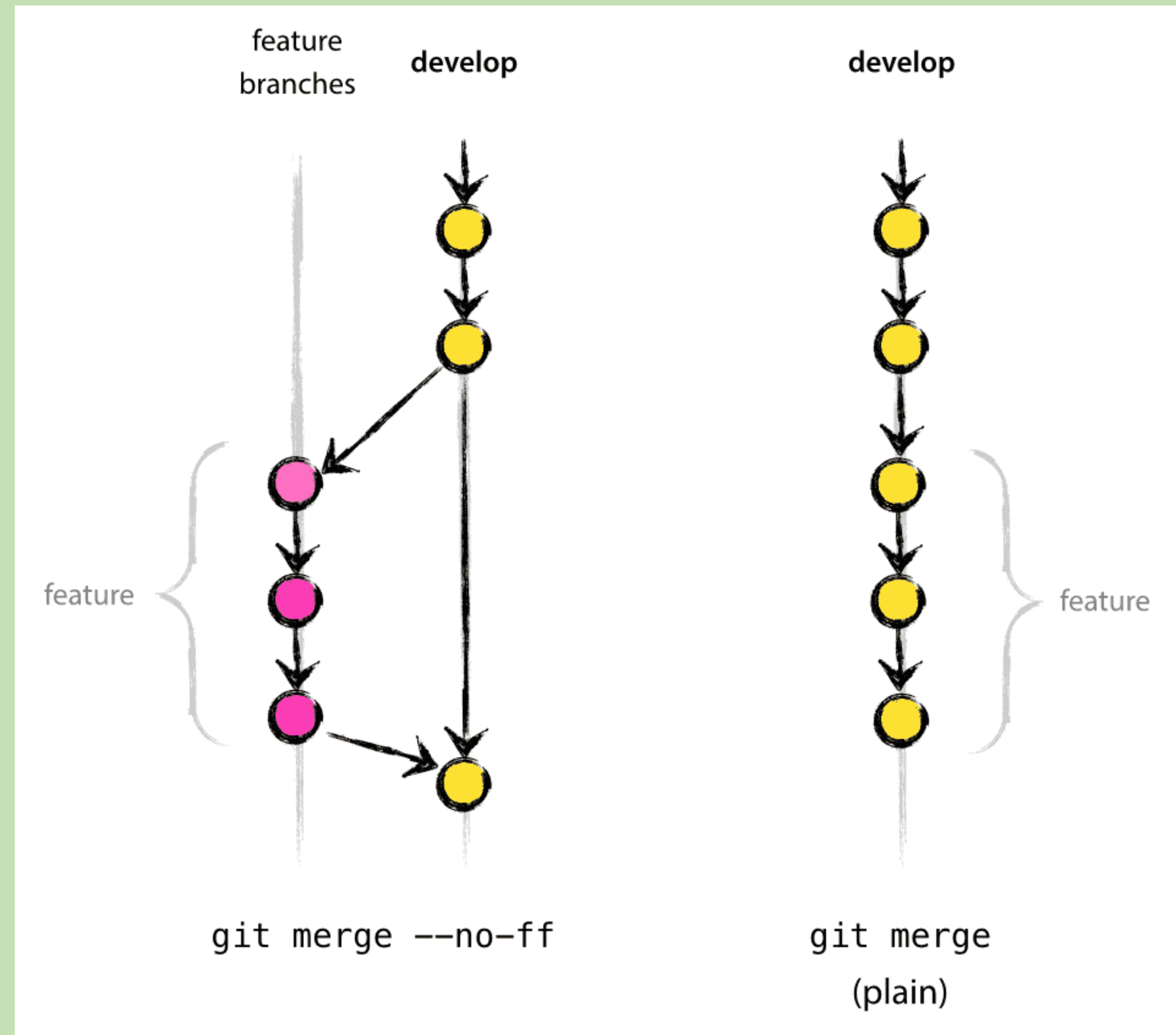
git merge --no-ff develop

Finish feature

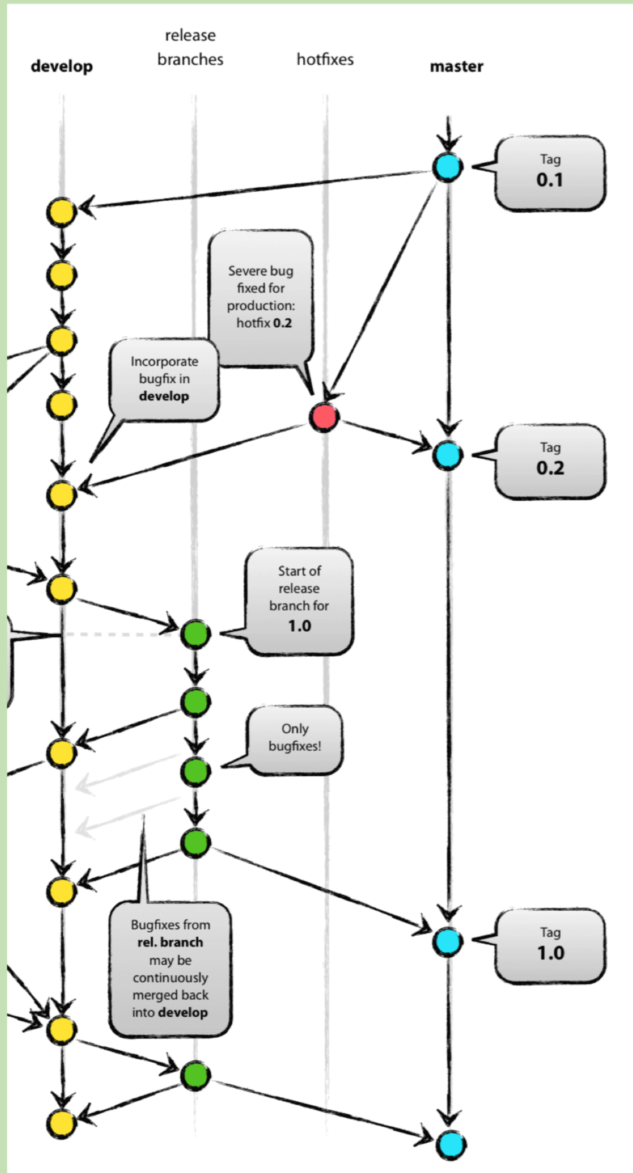
git checkout develop

git merge --no-ff myfeature

git branch -d myfeature



Release Workflow



Release branches can be made for early release/testing

Final commits can be made to release before delivering to master

Any extra commits should go back to develop to be in future release

Making Release Branches

Create release branch

git checkout -b release-1.0 develop

Create feature branches (if needed)

git checkout -b featureFor1.0 release-1.0

…

git checkout release-1.0

git merge --no-ff featureFor1.0

git checkout develop

git merge --no-ff featureFor1.0

Finish release

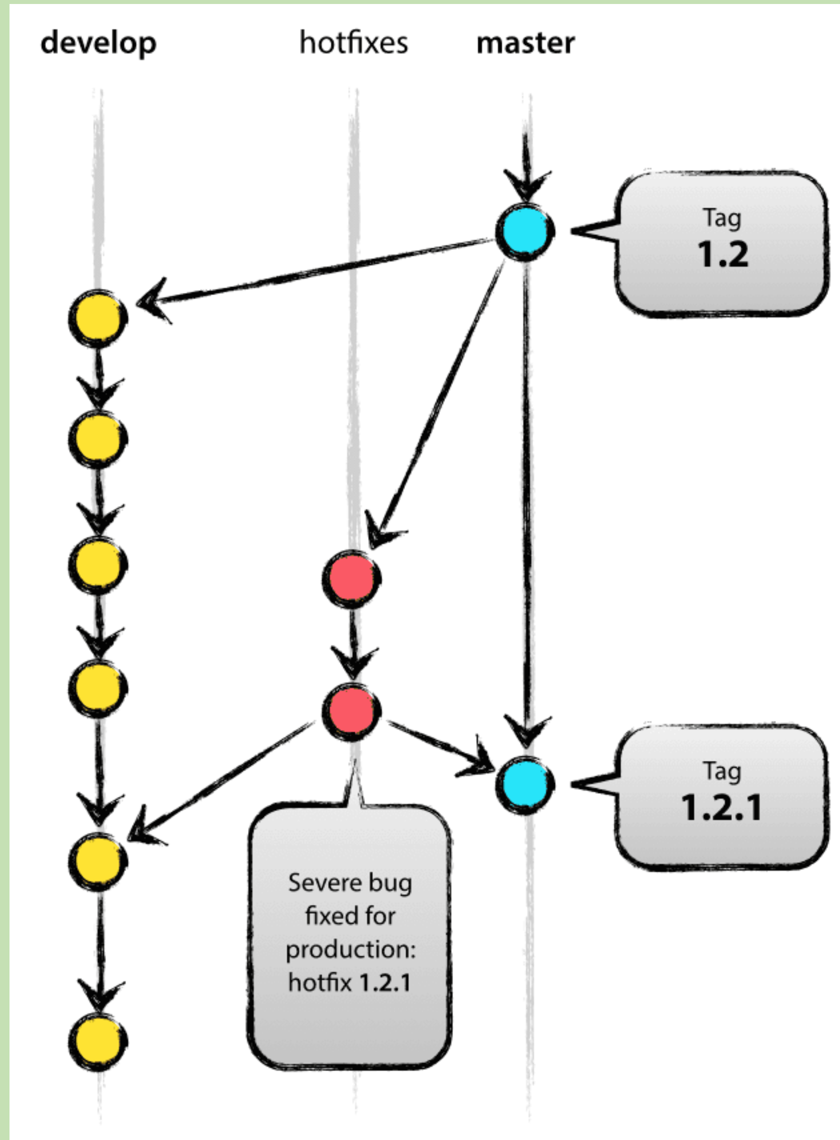
git checkout master

git merge --no-ff release-1.0

git tag -a 1.0 -m "Version 1.0"

git branch -d release-1.0

Hotfix Workflow



Hotfixes are urgent patches to master

Must also be merged to develop for future releases (and any release branches)

Making Hotfix Branches

Create hotfix branch

git checkout -b hotfix-1.0.1 master

Create feature branches (if needed, typically commit to hotfix branch)

git checkout -b featureForHotfix1.0.1 hotfix-1.0.1

Finish hotfix

git checkout master

git merge --no-ff hotfix-1.0.1

git tag -a 1.0.1 -m “Version 1.0.1”

merge to develop + releases

git branch -d hotfix-1.0.1

Data Science Workflow

Structure is more important than workflow

Model project as it works for you

Utilize branches off master for large features

Leverage tags for reproducing results

Ignore reproducible results and compiled analysis files (e.g. PDF from LaTeX)

```
core/
  - tests/
  - simulation.py
experiment_1/
  - tests/
  - data/
  - simulation.py
experiment_2/
  - tests/
  - data/
  - simulation.py
results/
analysis/
```

Moving Commits

Cherry picking copies a commit

git cherry-pick <commit>

Cherry pick multiple commits (commit1 not included)

git cherry-pick <commit1>..<commit2>

git cherry-pick <commit1>^..<commit2>

Cherry pick from another repo

git remote add <other-repository-name> <URL>

git fetch <other-repository-name>

git cherry-pick <commit>

Moving Branches

Rename a branch

git branch -m [<old-branch-name>] <new-branch-name>

Change a branch base

git rebase --onto <place-to-put-it> <last-change-that-should-NOT-move> <head to move>

Patches

Patch files are a diff stored in a text file (.patch extension)

From unstaged changes: *git diff > <patch_file>*

From staged changes: *git diff --cached > <patch_file>*

From a branch: *git format-patch <feature_branch> [-o <output_directory>]*

From a commit: *git format-patch <feature_branch> -1 <commit_hash>*

Between two tags: *git diff tag1 tag2 -- > the-patch.diff*

Apply a patch: *git apply <patch_file>*

Lunch!

Undoing Commits

git revert <SHA>

Commits the opposite changes of another commit

git reset

--soft

Undoes a commit but leaves the changes in the staging area

Used for adding changes to a commit

--mixed

Unstages changes (default action)

Used when you've accidentally added too much to the staging area

--hard

Undoes commits and throws away their changes. The nuclear option.

Used to throw away a commit

Reflog

Reflog records changes to HEAD

Useful for getting out of sticky situations (e.g. recovering a hard reset)

git reflog

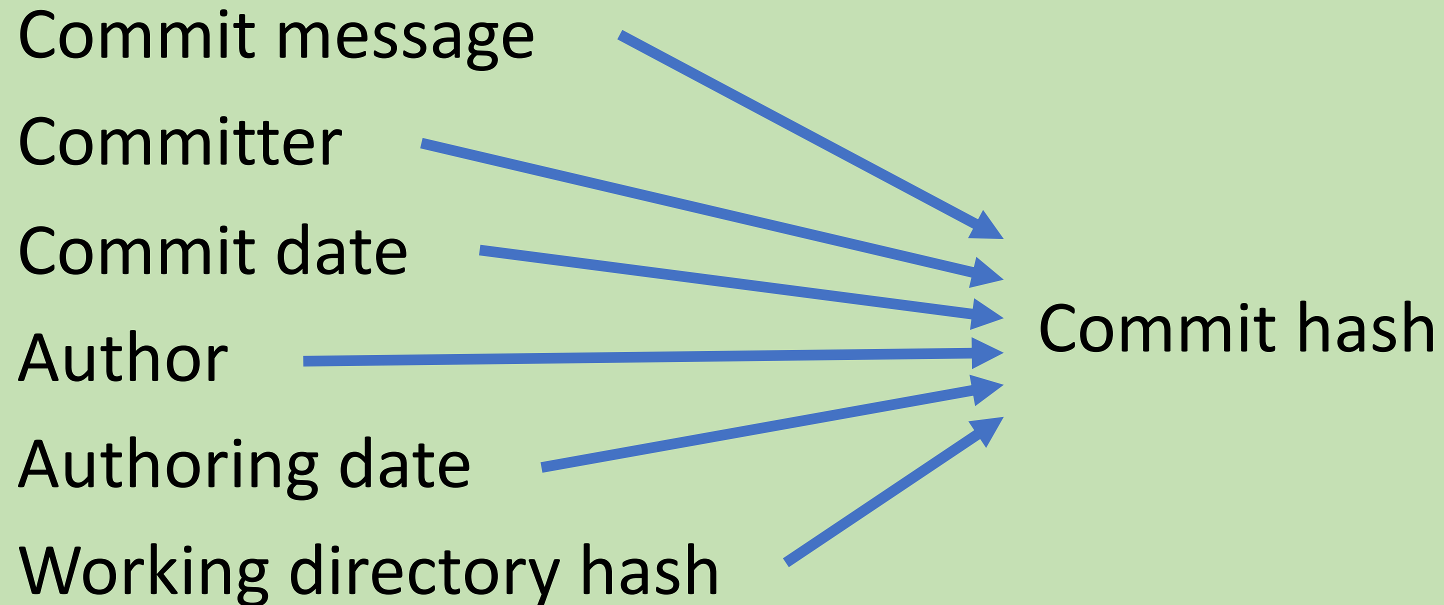
git reset --hard <reflog_SHA>

Debugging With Bisect

git bisect is used to find which commit introduced a bug

git bisect start	# Search start
git bisect bad	# Set point to bad commit
git bisect good <SHA>	# Set point to good commit/tag
git bisect bad	# Mark current commit as bad
git bisect good	# Mark current commit as good
git bisect reset	# Finish search

What’s in a commit?



Large Files

Once a file is committed it's forever in the repository, even after *git rm*

Changes to large files causes the repo size to grow rapidly

Large files that get updates should be ignored and versioned outside the repo (Box, Dropbox, rsync from server)

To completely remove a file from a repository, every commit must be edited with *git filter-branch* or BFG repo cleaner

Hooks

Hooks can be used to inject scripts before/after git actions

(e.g. display a warning before pushing)

Supports any executable script

Stored in the *.git/hooks* directory of a project (remove *.sample* ext.)