Version Control Therapy: Healing Your Relationship With Git, Branches, and Clean Code

Eduard Maievskyi, Ph.D.

www.linkedin.com/in/eduard-maievskyi-002618107

VanLUG, Burnaby Public Library (Metrotown) Branch
2025-04-05

Outline

- Introduction
- Main Concepts
- Branching Strategies
- Git Commands (some of them)
- Conclusion

INTRODUCTION

-Who am I? - Eduard Maievskyi, Ph. D.

- B. Sc. in Physics (June 2011)
  - V. N. Karazin Kharkiv National University, Kharkiv, Ukraine
- M. Sc. In Physics (May 2012)
  - V. N. Karazin Kharkiv National University, Kharkiv, Ukraine
- Ph. D. in Physics (October 2012 – July 2017)
  - W. Trzebiatowski Institute of Low Temperature and Structure Research Polish Academy of Sciences in Wroclaw, Poland
- Data Scientist (July 2017 -July 2018)
  - Physiolution Polska sp. z o.o., Wrocław, Poland
- Data Scientist (September 2018 – April 2019)
  - Yieldr Labs DI B.V., Amsterdam, Netherlands
- Data Scientist (April 2019 – October 2019)
  - Hotelchamp B.V., Amsterdam, Netherlands
- Data Scientist (December 2019 - March 2020)
  - Belvilla Services B.V., Amsterdam, Netherlands
- Research Assistant (March 2020 – December 2021)
  - W. Trzebiatowski Institute of Low Temperature and Structure Research Polish Academy of Sciences in Wroclaw, Poland
- Vice President, BI/Back-End Developer (January 2022 - February 2024)
  - BNY, Wrocław, Poland
- Lead Software Developer (April 2024 – October 2024)
  - Quantum World Technologies Inc. + Virtusa + BNY, Vancouver, BC

www.linkedin.com/in/eduard-maievskyi-002618107

The Creator – 20 Fun Facts (p.1)

- Named after a Nobel prize winner
- All the Torvalds in the world are relatives
- Commodore Vic 20 was his first computer (at the age of 10)
- Second Lieutenant Linus Torvalds
- He created Linux because he didn’t have money for UNIX
- Linux could have been called Freax
- Linux was his main project at University
- He married his student
- Linus has an asteroid named after him
- Linus had to battle for the trademark of Linux

https://itsfoss.com/linus-torvalds-facts/

The Creator – 20 Fun Facts (p.2)

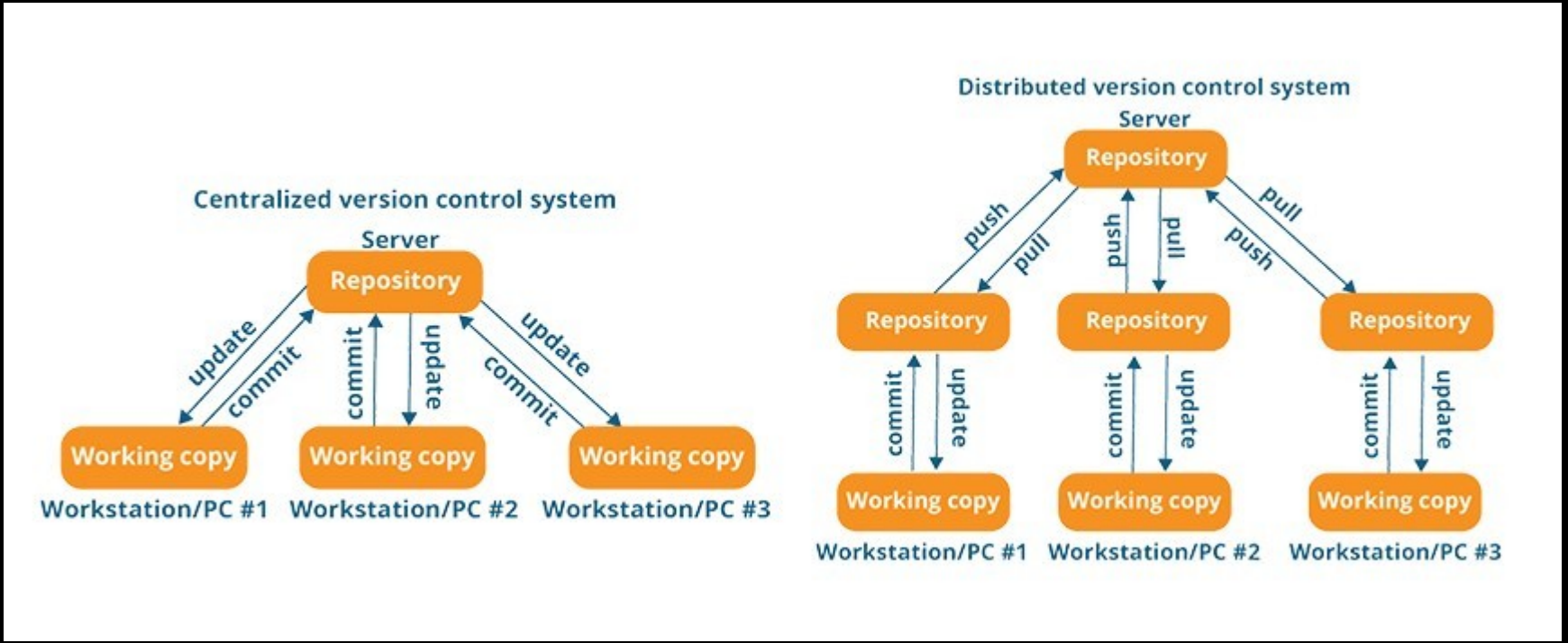
- Steve Jobs wanted him to work on Apple's macOS
- Linus also created **Git**
- Linus hardly codes these days
- Torvalds hates C++
- Even Linus Torvalds found Linux difficult to install (you can feel good about yourself now)
- He loves scuba diving
- The foul-mouthed Torvalds has improved his behavior
- He is too shy to speak in public
- Not a social media buff
- Torvalds is settled in the USA

https://itsfoss.com/linus-torvalds-facts/

So what is Git? :)

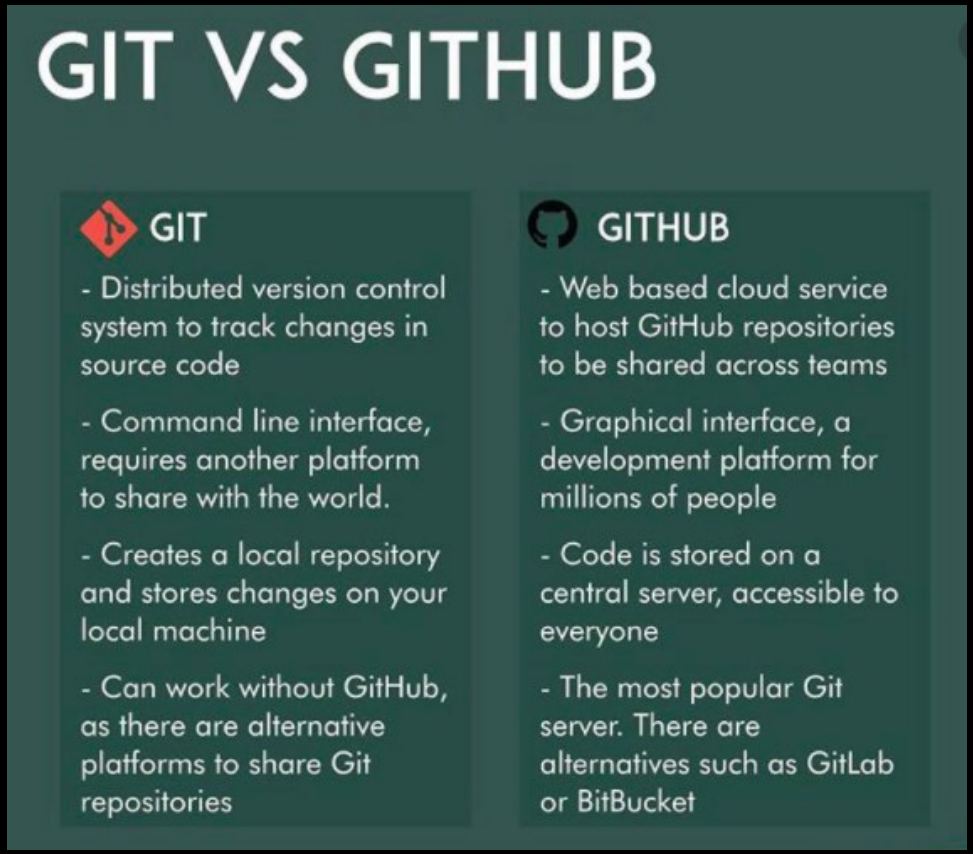
So what is Git?

Distributed Version Control System !



So what is Git?

Git ≠ GitHub or GitLab or BitBucket



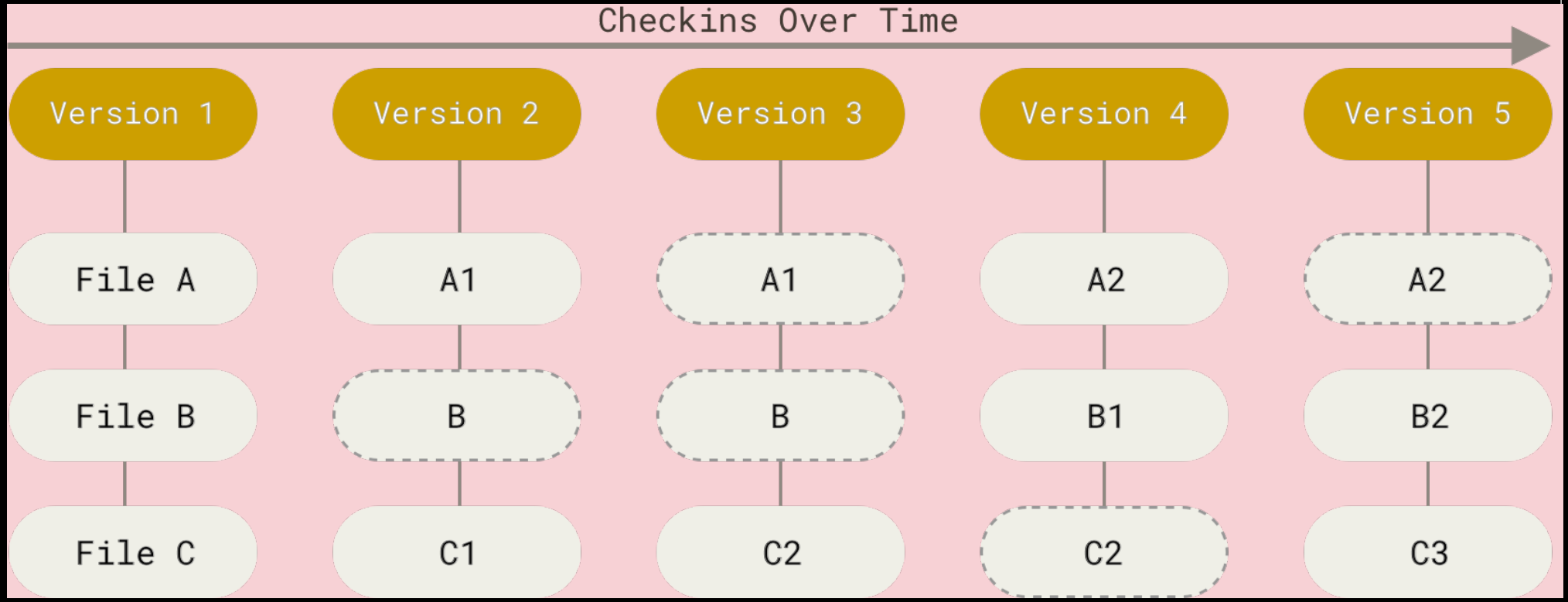
https://www.google.com/url?sa=i&url=https%3A%2F%2Fjain-rishabh.medium.com%2Farticle-on-git-and-github-6c4b98b

Short History of Git Creation

- **Early 2005: The Need Arises**
  - BitKeeper revokes its free license for the Linux kernel project, creating an urgent need for a new DVCS.
- **April 2005: Linus Torvalds Starts Git**
  - Torvalds begins developing Git specifically for Linux kernel development.
  - Key goals: Speed, data integrity, distributed nature, non-linear workflows (branching/merging).
- **April 7, 2005: Initial Release**
  - Git becomes self-hosting (manages its own source code), marking its first functional release.
- **July 2005: Maintainership Handover**
  - Torvalds passes maintainership to Junio C Hamano, who continues to lead the project.
- **December 2005: Git 1.0 Released**
  - The first major stable version, Git 1.0.0, is released, solidifying its foundation.

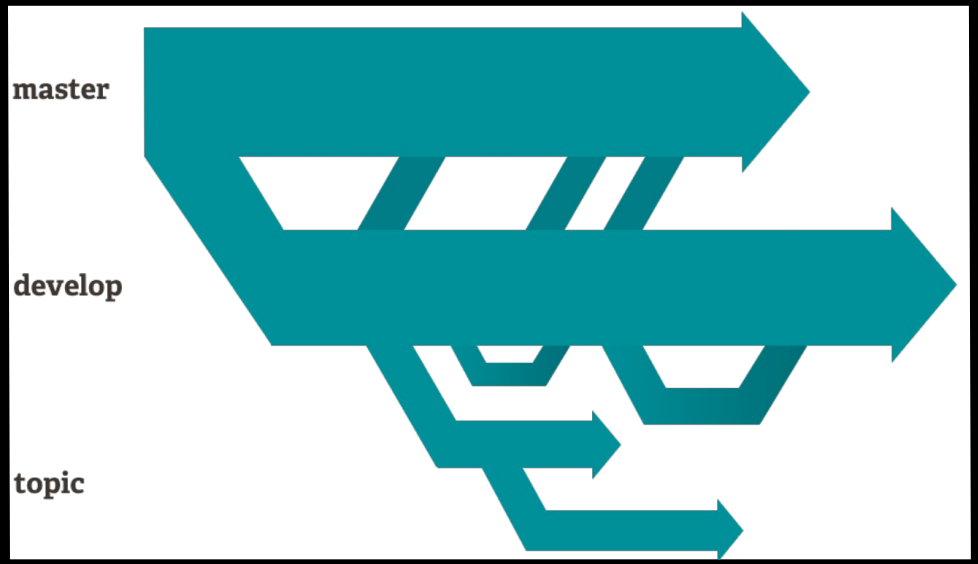
MAIN CONCEPTS

Main Concepts – Stream of Snapshots



https://git-scm.com/book/ms/v2/Getting-Started-What-is-Git%3F

Main Concepts – Branching and Merging



- **Frictionless Context Switching.** Create a branch to try out an idea, commit a few times, switch back to where you branched from, apply a patch, switch back to where you are experimenting, and merge it in.
- **Role-Based Codelines.** Have a branch that always contains only what goes to production, another that you merge work into for testing, and several smaller ones for day to day work.
- **Feature Based Workflow.** Create new branches for each new feature you're working on so you can seamlessly switch back and forth between them, then delete each branch when that feature gets merged into your main line.
- **Disposable Experimentation.** Create a branch to experiment in, realize it's not going to work, and just delete it - abandoning the work—with nobody else ever seeing it (even if you've pushed other branches in the meantime).

https://git-scm.com/about/branching-and-merging

Main Concepts – Branching and Merging

Main Concepts – Small and Fast

smaller is faster

Operation		Git	SVN	
Commit Files (A)	Add, commit and push 113 modified files (2164+, 2259-)	0.64	2.60	4x
Commit Images (B)	Add, commit and push a thousand 1 kB images	1.53	24.70	16x
Diff Current	Diff 187 changed files (1664+, 4859-) against last commit	0.25	1.09	4x
Diff Recent	Diff against 4 commits back (269 changed/3609+,6898-)	0.25	3.99	16x
Diff Tags	Diff two tags against each other (v1.9.1.0/v1.9.3.0)	1.17	83.57	71x
Log (50)	Log of the last 50 commits (19 kB of output)	0.01	0.38	31x
Log (All)	Log of all commits (26,056 commits – 9.4 MB of output)	0.52	169.20	325x
Log (File)	Log of the history of a single file (array.c – 483 revs)	0.60	82.84	138x
Update	Pull of Commit A scenario (113 files changed, 2164+, 2259-)	0.90	2.82	3x
Blame	Line annotation of a single file (array.c)	1.91	3.04	1x

times in seconds

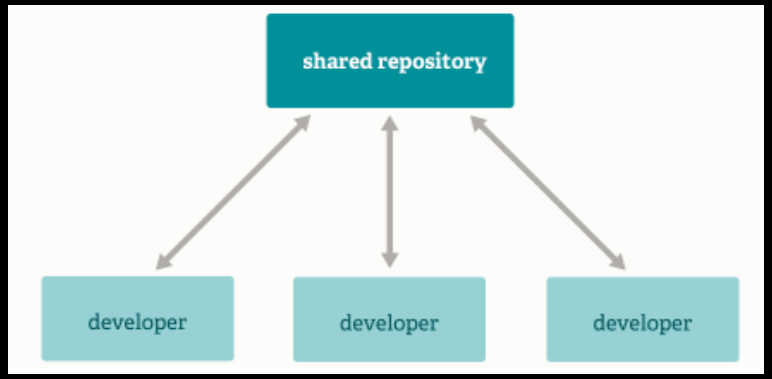
For testing, large AWS instances were set up in the same availability zone. Git and SVN were installed on both machines, the Ruby repository was copied to both Git and SVN servers, and common operations were performed on both.

https://git-scm.com/about/small-and-fast

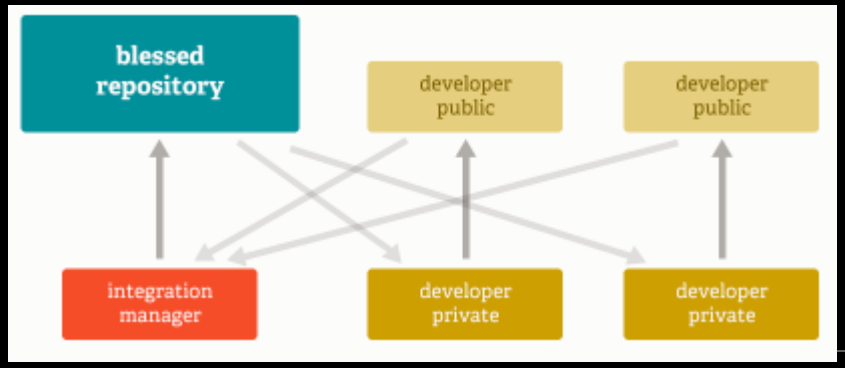
Main Concepts – DISTRIBUTED!!! (again)

Multiple Backups and ANY Workflow

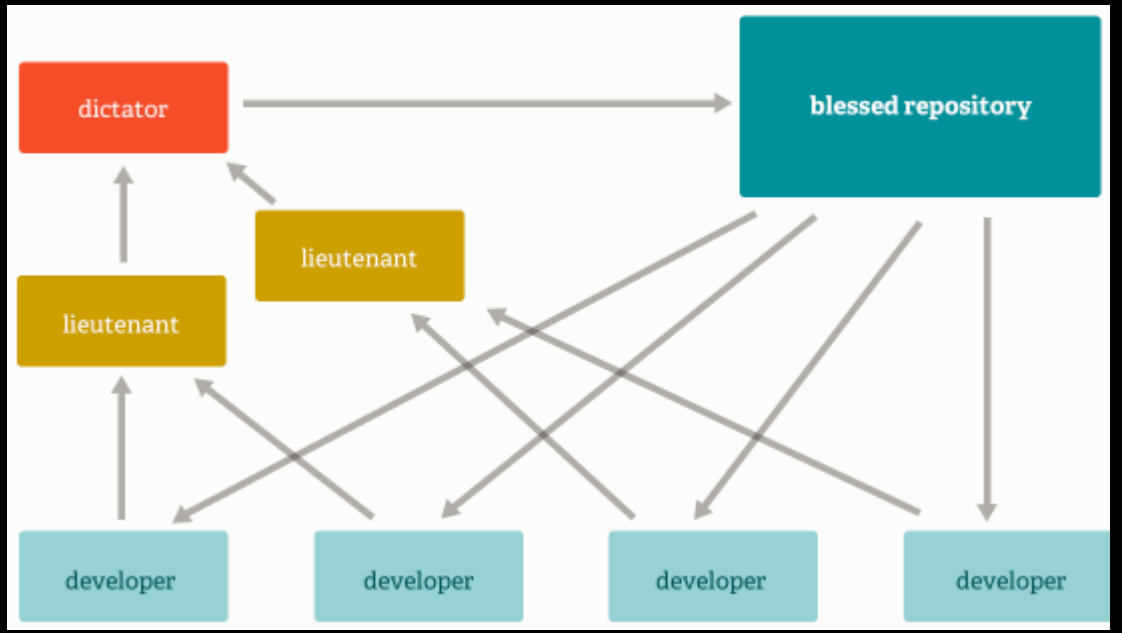
Subversion-Style Workflow



Integration Manager Workflow



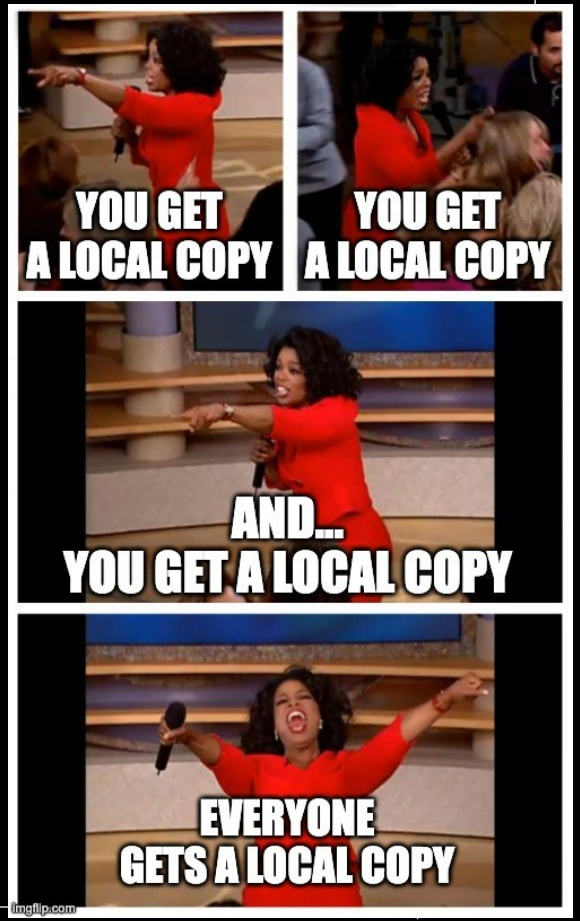
Dictator and Lieutenants Workflow



https://git-scm.com/about/distributed

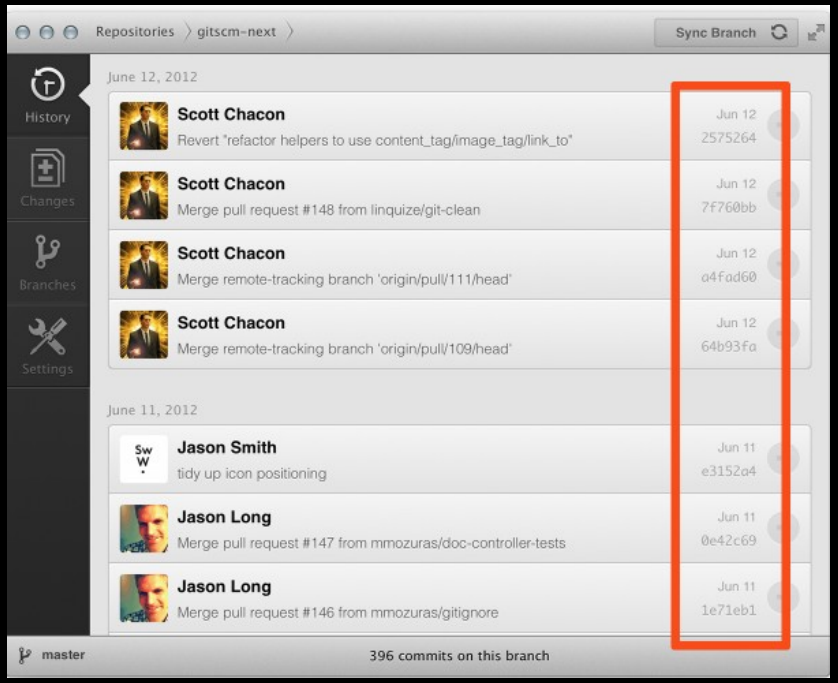
Main Concepts – Nearly Every Operation Is Local

For example, to browse the history of the project, Git doesn't need to go out to the server to get the history and display it for you – it simply reads it directly from your local database. This means you see the project history almost instantly. If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally.



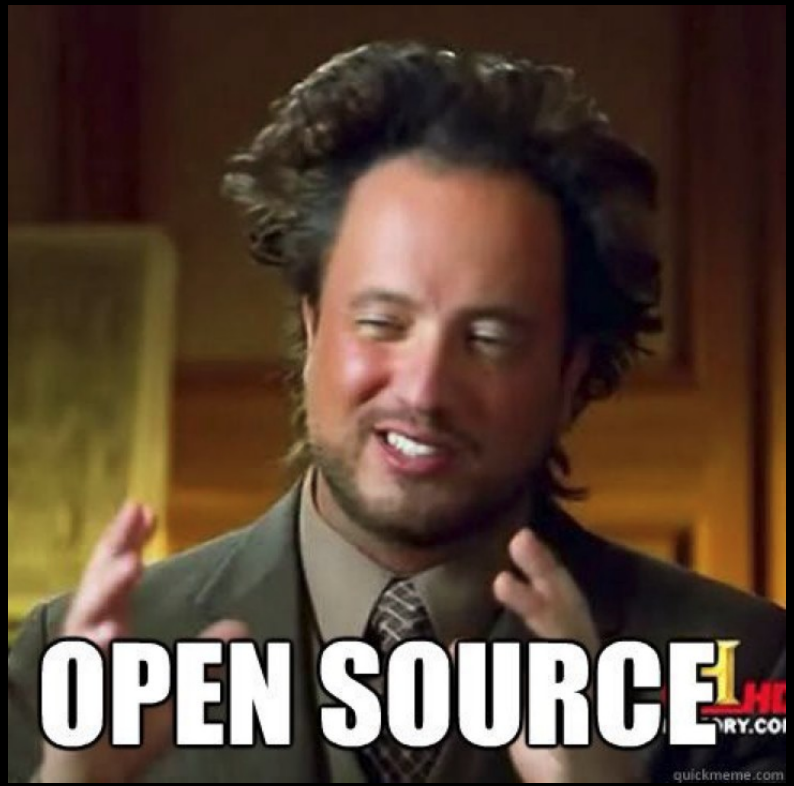
https://git-scm.com/book/ms/v2/Getting-Started-What-is-Git%3F

Main Concepts – Data Assurance & Integrity



https://git-scm.com/about/data-assurance

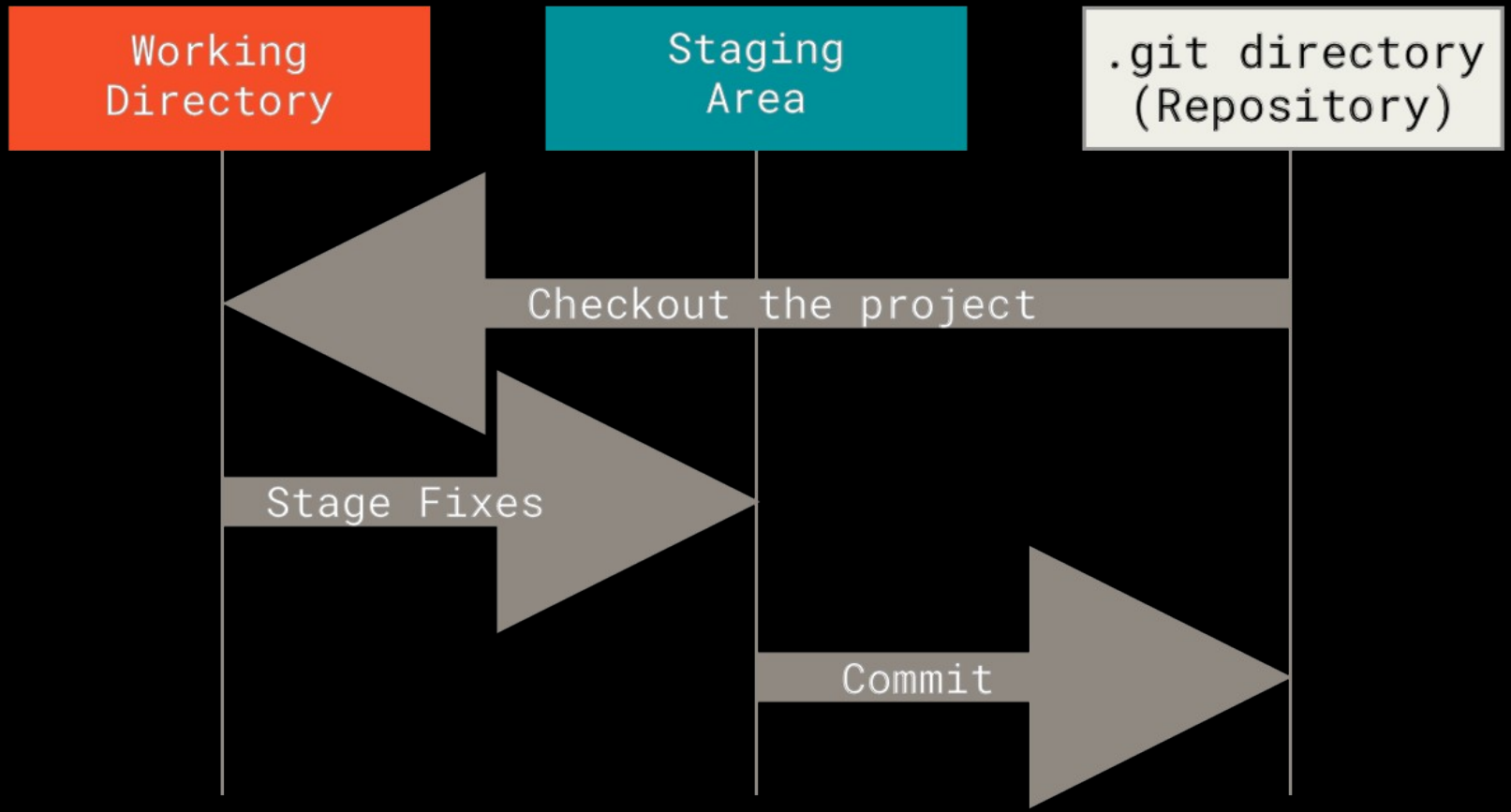
Main Concepts – Open-Source & FREE!!!



https://git-scm.com/about/free-and-open-source

Main Concepts – Git Generally Only Adds Data

Main Concepts — The Three States

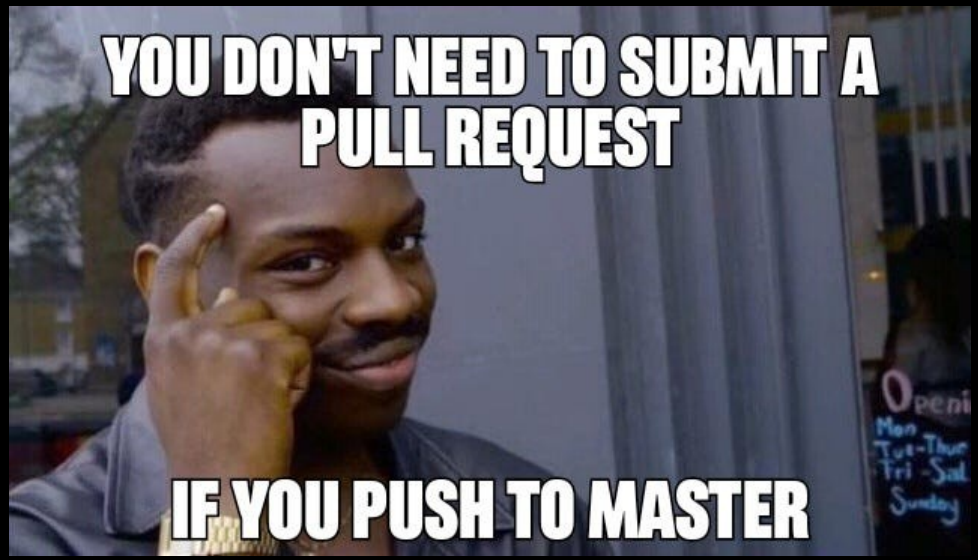


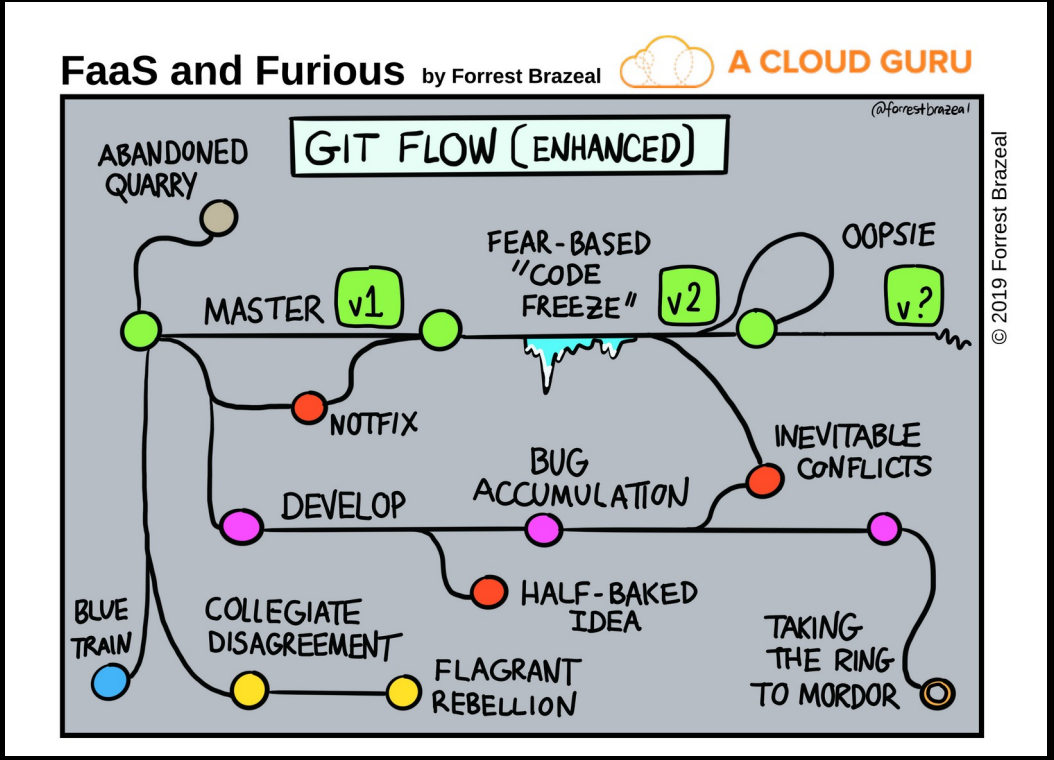
https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

Main Concepts – The Three States :)

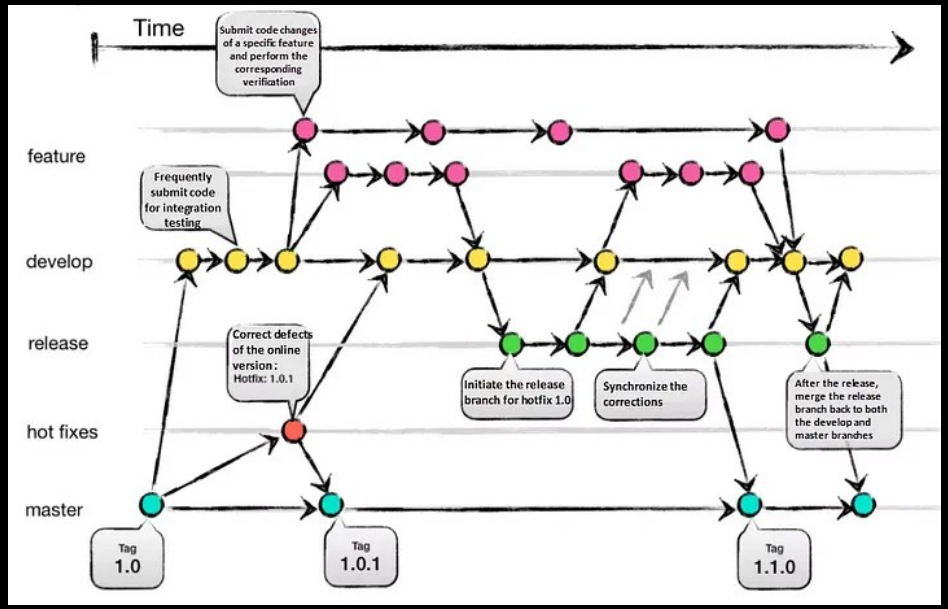
BRANCHING STRATEGIES

Meme Time :)


YOU DON'T NEED TO SUBMIT A PULL REQUEST
IF YOU PUSH TO MASTER


FaaS and Furious by Forrest Brazeal
A CLOUD GURU
@forrestbrazeal
GIT FLOW (ENHANCED)
ABANDONED QUARRY
MASTER
v1
FEAR-BASED "CODE FREEZE"
v2
OOPSIE
v?
NOTFIX
INEVITABLE CONFLICTS
BUG ACCUMULATION
DEVELOP
HALF-BAKED IDEA
BLUE TRAIN
COLLEGIATE DISAGREEMENT
FLAGRANT REBELLION
TAKING THE RING TO MORDOR
© 2019 Forrest Brazeal

Git-Flow



Pros:

- Well-suited for large teams and aligning work across multiple teams.

- Effective handling of multiple product versions.

- Clear responsibilities for each branch.

- Allows for easy navigation of production versions through tags.

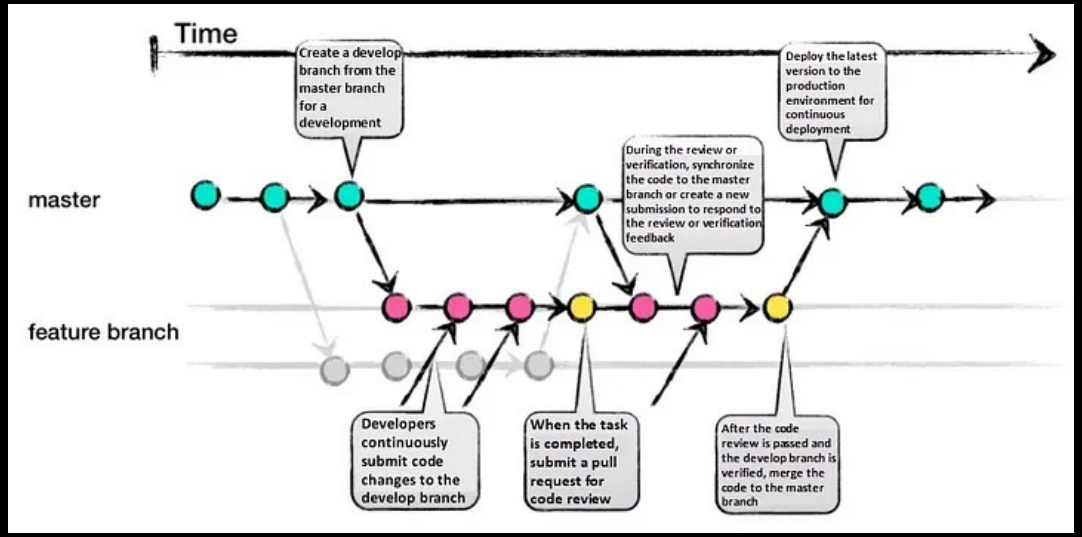
Cons:

- Complexity due to numerous branches, potentially leading to merge conflicts.

- Development and release frequency may be slower due to multi-step process.

- Requires team consensus and commitment to adhere to the strategy.

GitHub-Flow



Pros:

- Faster feedback cycles and shorter production cycles.

- Ideal for asynchronous work in smaller teams.

- Agile and easier to comprehend compared to Git-Flow.

Cons:

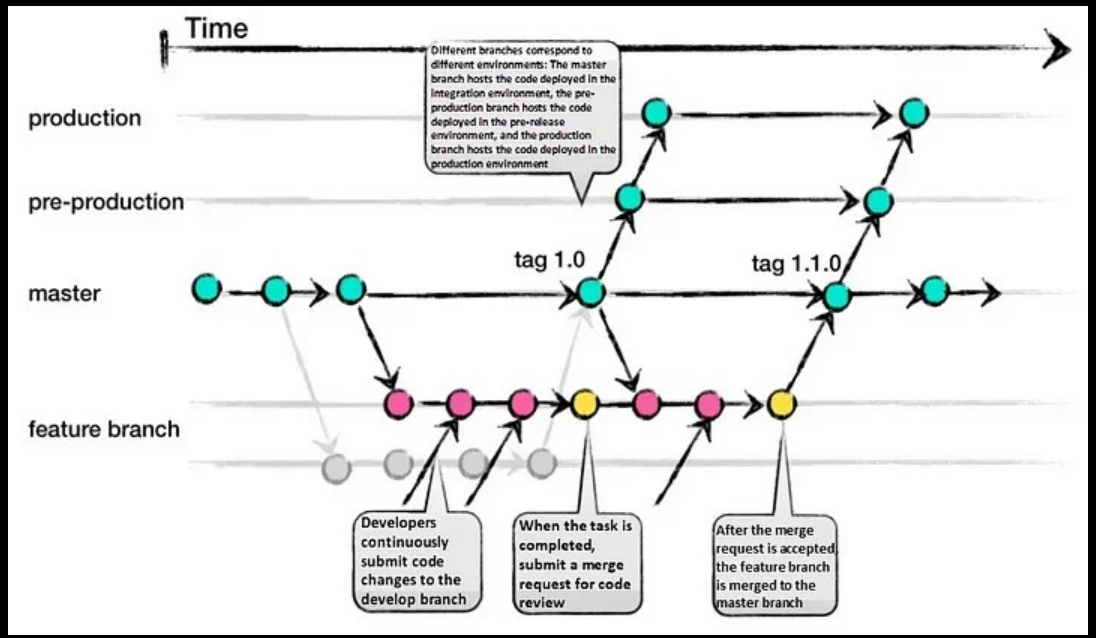
- Merging a feature branch implies it is production-ready, potentially introducing bugs without proper testing and a robust CI/CD process.

- Long-living branches can complicate the process.

- Challenging to scale for larger teams due to increased merge conflicts.

- Supporting multiple release versions concurrently is difficult.

GitLab-Flow (my favourite ♥)



Pros:

- Can handle multiple release versions or stages effectively.

- Simpler than Git-Flow.

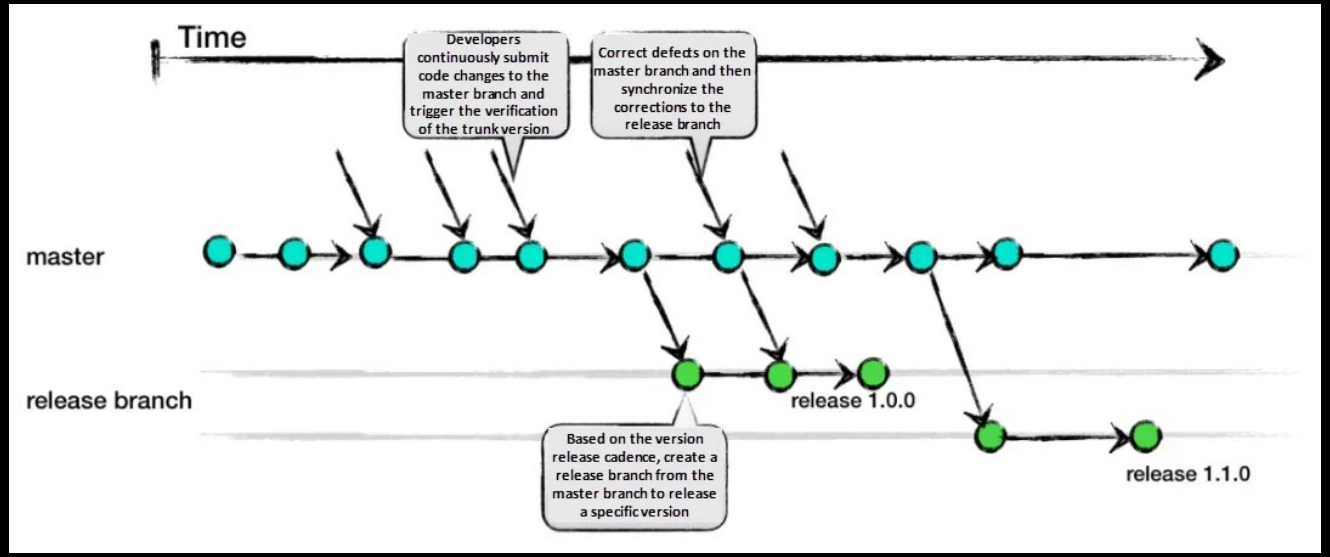
- Focuses on quality with a lean approach.

Cons:

- Complexity increases when maintaining multiple versions.

- More intricate compared to GitHub-Flow.

Trunk Based Development



Pros

- Encourages DevOps and unit testing best practices.

- Enhances collaboration and reduces merge conflicts.

- Allows quick releases.

Cons

- Requires an experienced team that can slice features appropriately for regular integration.

- Relies on strong CI/CD practices to maintain stability.

Pssst...

Hey, folks!

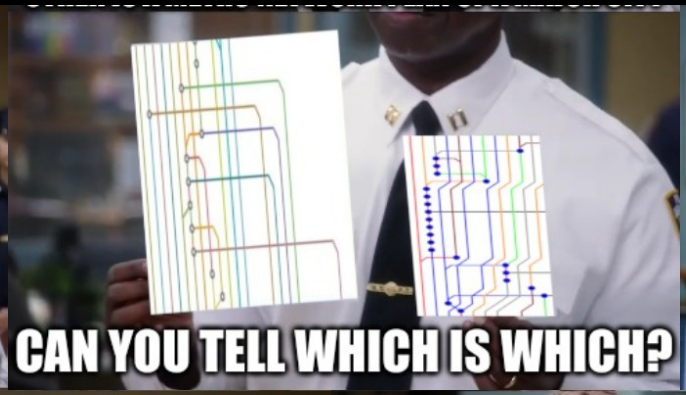
If you want to know more about projects forking and creation of PRs in GitHub check THIS video of this exceptional professional...

https://youtu.be/RkCKG-sruKc?si=nvCEeGROVDRDnKjx

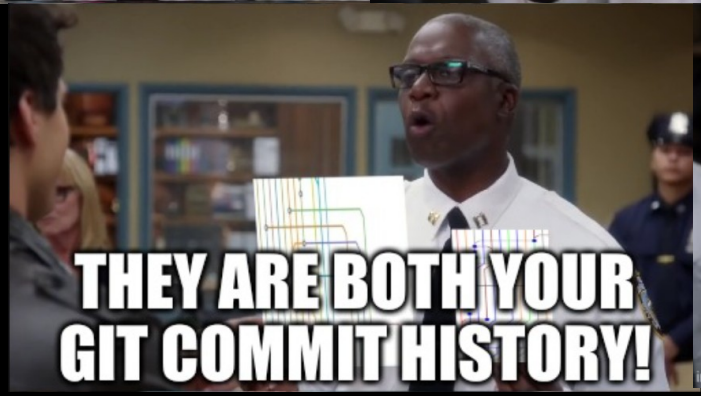
Waldirio Pinheiro

Meme Time :)


SENIOR DEV
ONE IS YOUR GIT COMMIT HISTORY, THE OTHER IS A METRO NETWORK PLAN OF A MAJOR CITY


CAN YOU TELL WHICH IS WHICH?

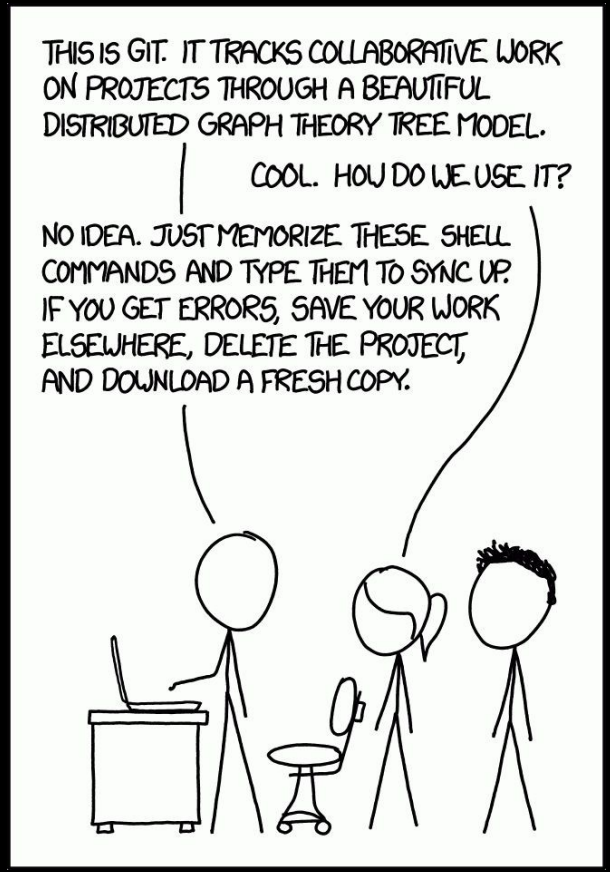

JUNIOR DEV
THAT'S THE METRO MAP?


THEY ARE BOTH YOUR GIT COMMIT HISTORY!
imgflip.com


HE'S GOOD...

GIT COMMANDS
(some of them)

How well do we know Git?



https://youtu.be/aolI_Rz0ZqY?si=GFDW4k-z29H6Wz1l

How well do we know Git?

PORCELAIN commands ≈82

(High-level commands intended for direct user interaction in the repository)

≈44 main commands (add, commit, push ...)

≈11 manipulators (config, reflog, ...)

≈17 interrogators (blame, fsck, rerere,...)

≈10 interactors (send-email, p4, svn, ...)

PLUMBING commands ≈63

(Low-level commands used for scripting and direct repository manipulation; often used by Porcelain commands)

≈19 manipulators (apply, commit-tree, ...)

≈21 interrogators (cat-file, ...)

≈5 syncing (fetch-pack, send-pack, ...)

≈18 internal (check-attr, sh-i18n, ...)

https://youtu.be/aolI_Rz0ZqY?si=GFDW4k-z29H6Wz1l (Feb 2024)

git init vs git clone

`git init`
One Person Starting a New Repository Locally

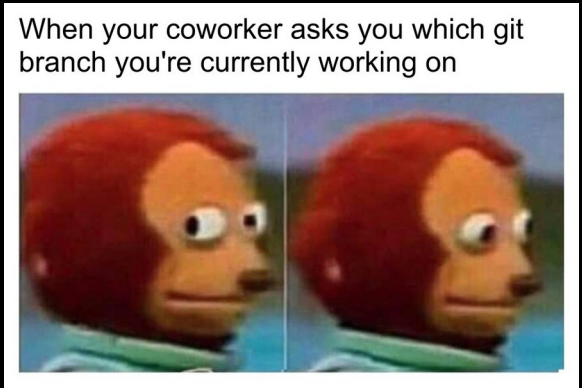
`git clone ssh.../https...`
Use when remote already exist

If you’re planning to work with a huge project it is worth to check docs and info about shallow clones...

https://git-scm.com/docs/git-clone

Branch Creation, Removal and Navigation

`git branch` or `git branch --list`
Feel free to add pattern if you use second option and look for something specific



Switch to branch
`git checkout <branchname>`

Removal
`git branch -d <branchname>`
`--delete <branchname>`
or `-D` for `--delete --force`

Renaming
`git branch -m [<oldbranch>] <newbranch>`
or `-M` for `--move --force`

Copy
`git branch -c [<oldbranch>] <newbranch>`
or `-C` for `--move --force`

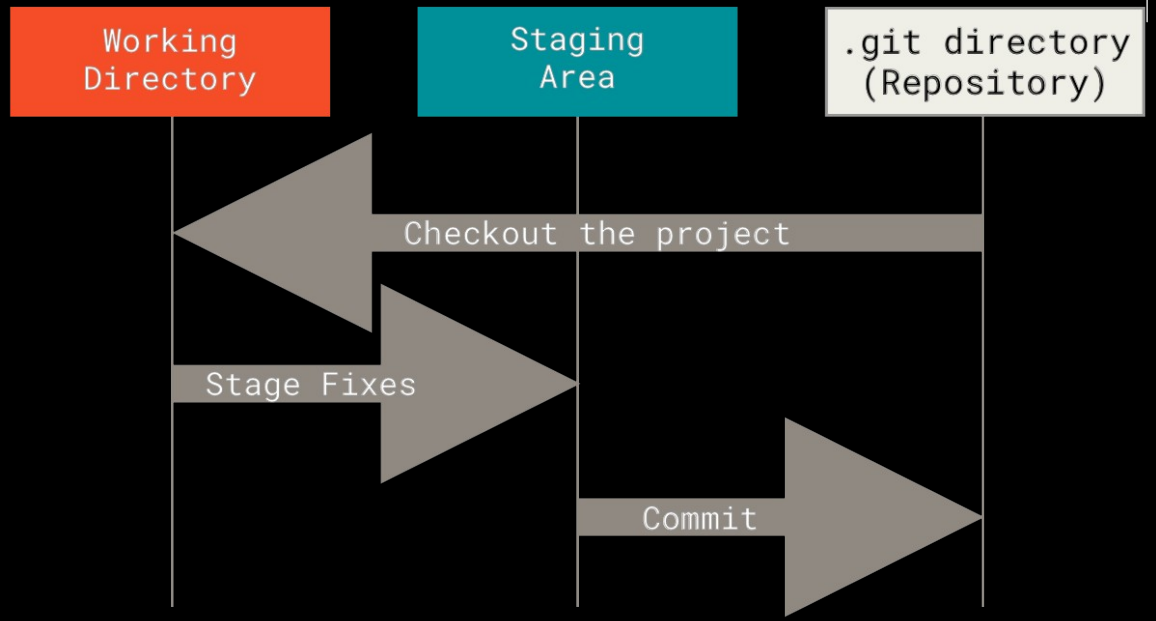
Create & Switch
`git checkout -b <newbranch>`
or `-B` for force resetting if branch exists

https://git-scm.com/docs/git-branch https://git-scm.com/docs/git-checkout

Staging & Committing

`git status`
provides summary of files with staged and unstaged changes

`git add <filename1> <filename2>`
adds selected files to the staging area.
Be careful with using `*` and all other methods, especially when you are not sure about your `.gitignore` file configuration

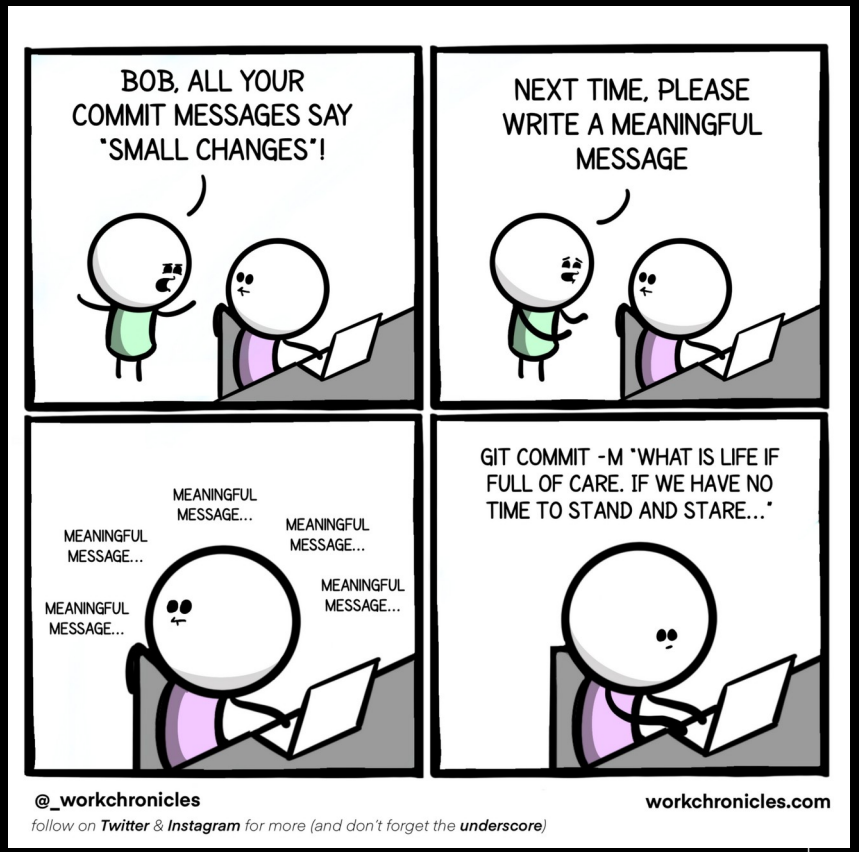


`git commit`
Record changes to repo

`git commit --amend`
Amends the last commit, can be used to amend a merge commit

https://git-scm.com/docs/git-add https://git-scm.com/docs/git-commit

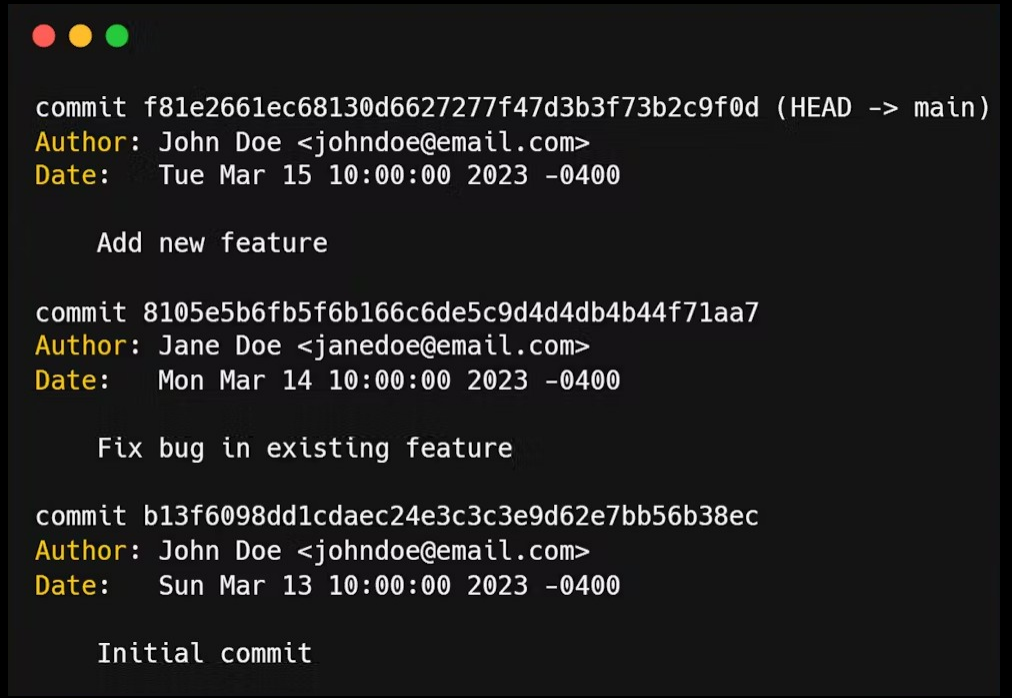
Staging & Committing



git log

```
git log [<options>] [<revision-range>]
```

List commits that are reachable by following the parent links from the given commit(s), but exclude commits that are reachable from the one(s) given with a ^ in front of them. The output is given in reverse chronological order by default.



https://git-scm.com/docs/git-log

git reflog — knows (almost) everything

```
git reflog [show] [<log-options>] [<ref>]
```

Reference logs, or "reflogs", record when the tips of branches and other references were updated in the local repository. Reflogs are useful in various Git commands, to specify the old value of a reference. For example, HEAD@{2} means "where HEAD used to be two moves ago", master@{one.week.ago} means "where master used to point to one week ago in this local repository", and so on.

https://git-scm.com/docs/git-reflog

git stash

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.

The modifications stashed away by this command can be listed with **git stash list**, inspected with **git stash show**, and restored (potentially on top of a different commit) with **git stash apply**. Calling **git stash** without any arguments is equivalent to **git stash push**. A stash is by default listed as "WIP on **branchname** …", but you can give a more descriptive message on the command line when you create one.

The latest stash you created is stored in refs/stash; older stashes are found in the reflog of this reference and can be named using the usual reflog syntax (e.g. stash@{0} is the most recently created stash, stash@{1} is the one before it, stash@{2.hours.ago} is also possible). Stashes may also be referenced by specifying just the stash index (e.g. the integer n is equivalent to stash@{n}).

https://git-scm.com/docs/git-stash

git reset

git reset [<mode>] [<commit>]

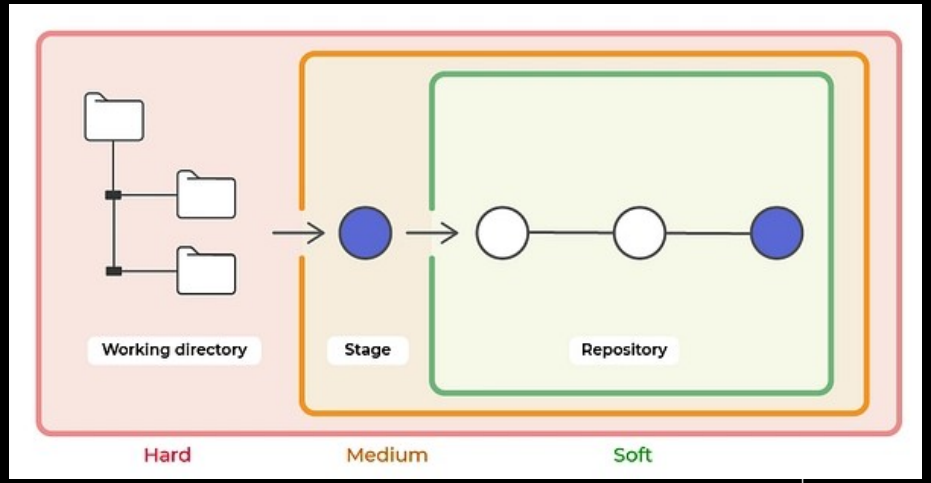
This form resets the current branch head to <commit> and possibly updates the index (resetting it to the tree of <commit>) and the working tree depending on <mode>. Before the operation, ORIG_HEAD is set to the tip of the current branch. If <mode> is omitted, defaults to --mixed. You can find the 2 most frequently used <mode>s below:

--soft

Does not touch the index file or the working tree at all (but resets the head to <commit>, just like all modes do). This leaves all your changed files "Changes to be committed", as git status would put it.

--hard

Resets the index and working tree. Any changes to tracked files in the working tree since <commit> are discarded. Any untracked files or directories in the way of writing any tracked files are simply deleted.



https://git-scm.com/docs/git-reset https://dev.to/meghasharmaaaa/what-is-git-reset-25n3

git rebase

Reapplies commits on top of another base tip

The primary reason for rebasing is to maintain a linear project history. For example, consider a situation where the main branch has progressed since you started working on a feature branch. You want to get the latest updates to the main branch in your feature branch, but you want to keep your branch's history clean so it appears as if you've been working off the latest main branch. This gives the later benefit of a clean merge of your feature branch back into the main branch.

```
          A---B---C topic
         /
    D---E---F---G master
```

`git rebase master` or `git rebase master topic`

```
                  A'--B'--C' topic
                 /
    D---E---F---G master
```

This command also can be used as a git commit --amend on steroids :)

```
git rebase --interactive <base>

pick 2231360 some old commit
pick ee2adc2 Adds new feature

# Rebase 2cf755d..ee2adc2 onto 2cf755d (9 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
```

https://git-scm.com/docs/git-rebase https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase

git merge

git merge will combine multiple sequences of commits into one unified history. In the most frequent use cases, git merge is used to combine two branches.

```
          A---B---C topic                          A---B---C topic
         /                    git merge topic     /         \
D---E---F---G master                     D---E---F---G---H master
```

Merge Methods

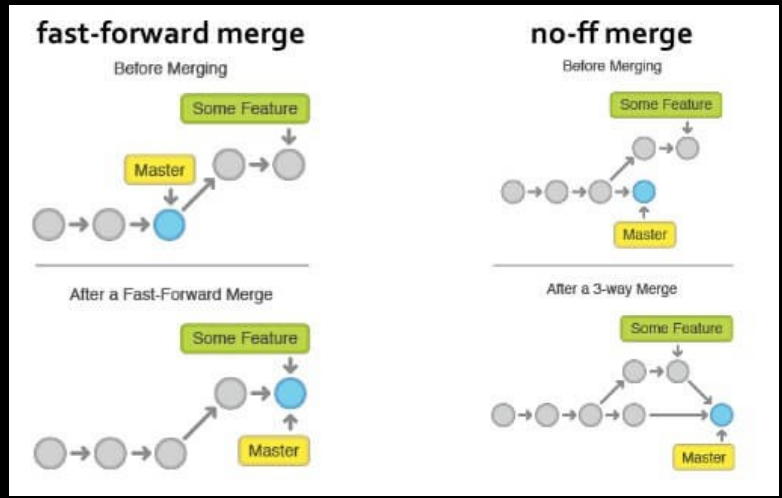
--ff
--no-ff
--ff-only

Specifies how a merge is handled when the merged-in history is already a descendant of the current history. --ff is the default unless merging an annotated (and possibly signed) tag that is not stored in its natural place in the refs/tags/ hierarchy, in which case --no-ff is assumed.

With --ff, when possible resolve the merge as a fast-forward (only update the branch pointer to match the merged branch; do not create a merge commit). When not possible (when the merged-in history is not a descendant of the current history), create a merge commit.

With --no-ff, create a merge commit in all cases, even when the merge could instead be resolved as a fast-forward.

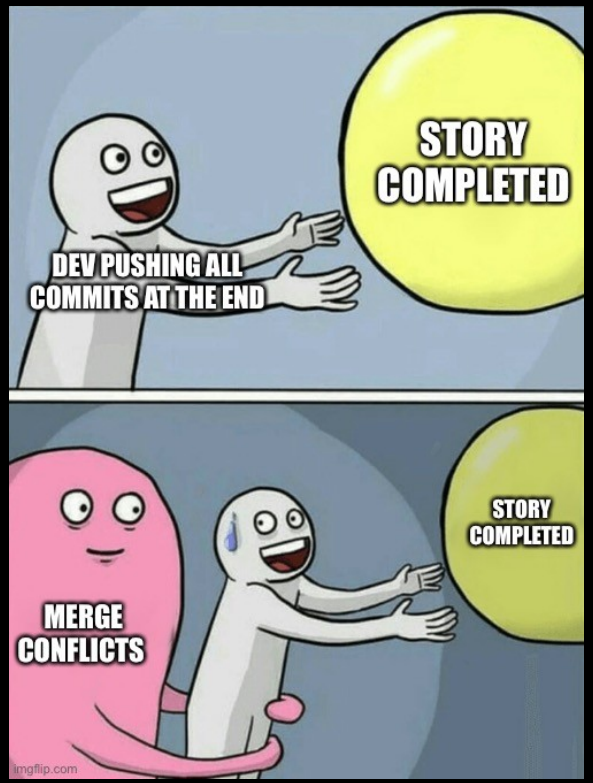
With --ff-only, resolve the merge as a fast-forward when possible. When not possible, refuse to merge and exit with a non-zero status.



https://git-scm.com/docs/git-merge

https://www.atlassian.com/git/tutorials/using-branches/git-merge
https://docs.gitlab.com/user/project/merge_requests/methods/

Resolving Conflicts



Conflicts generally arise when two people have changed the same lines in a file, or if one developer deleted a file while another developer was modifying it. In these cases, Git cannot automatically determine what is correct. Conflicts only affect the developer conducting the merge, the rest of the team is unaware of the conflict. Git will mark the file as being conflicted and halt the merging process. It is then the developers' responsibility to resolve the conflict.

https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts

Resolving Conflicts

Solving conflicts is not so scary and complex

git will explain (almost) everything and provide suggestions

```
On branch main
Unmerged paths:
(use "git add/rm ..." as appropriate to mark resolution)
both modified: hello.py
```

```
here is some content not affected by the conflict
<<<<<<< main
this is conflicted text from main
=======
this is conflicted text from feature branch
>>>>>>> feature branch
another part of the content not affected by the conflict
```

https://git-scm.com/docs/git-merge

https://www.atlassian.com/git/tutorials/using-branches/git-merge
https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts

git cherry-pick

git cherry-pick is a useful tool but not always a best practice. Cherry picking can cause duplicate commits and many scenarios where cherry picking would work, traditional merges are preferred instead.

```
    a---b---c---d   Main
          \
           e---f---g  Feature
```

```
git checkout main

git cherry-pick commit_f_sha
```

```
a---b---c---d---f   Main
          \
            e---f---g Feature
```

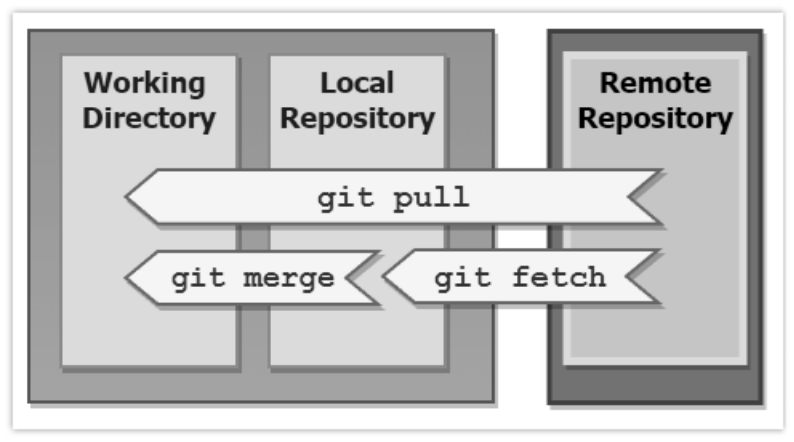
https://git-scm.com/docs/git-cherry-pick https://www.atlassian.com/git/tutorials/cherry-pick

git fetch & git pull

The **git pull** command is used to fetch and download content from a remote repository and immediately update the local repository to match that content. Merging remote upstream changes into your local repository is a common task in Git-based collaboration work flows. The **git pull** command is actually a combination of two other commands, **git fetch** followed by **git merge**. In the first stage of operation **git pull** will execute a **git fetch** scoped to the local branch that **HEAD** is pointed at. Once the content is downloaded, **git pull** will enter a merge workflow. A new merge commit will be-created and **HEAD** updated to point at the new commit.

On a user's workstation, a Git installation includes the following items:

- The local Git repository where the history of all commits across all branches are maintained.
- A working directory where a developer actively edits and updates files that Git tracks.



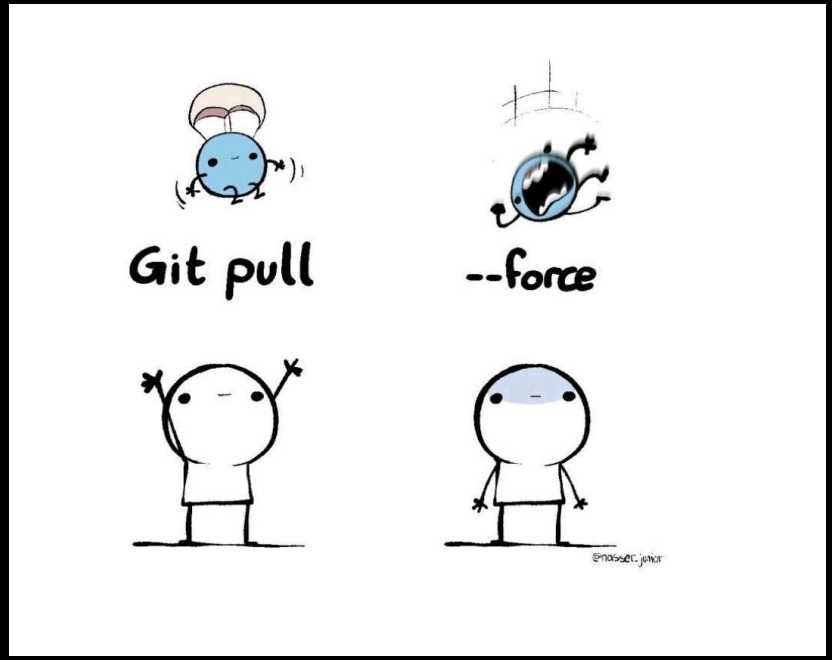
https://git-scm.com/docs/git-pull

https://www.atlassian.com/git/tutorials/syncing/git-pull

https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/Git-pull-vs-fetch-Whats-the-difference#:~:text=Difference%20between%20Git%20fetch%20and,git%20pull%20command%20does%20both.

git fetch & git pull

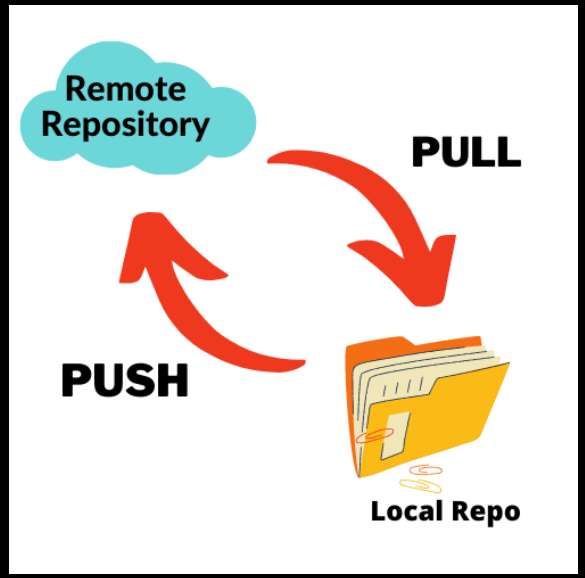
Caution!!!
Increased probability of making your life harder

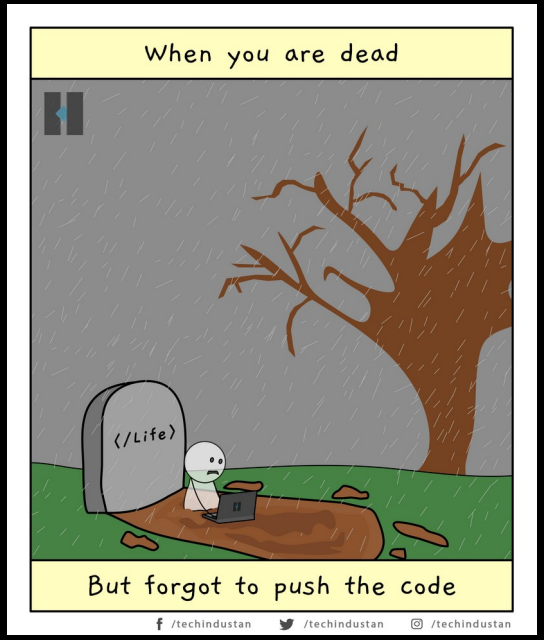


Consider execution of
`git pull --rebase=interactive`

git push

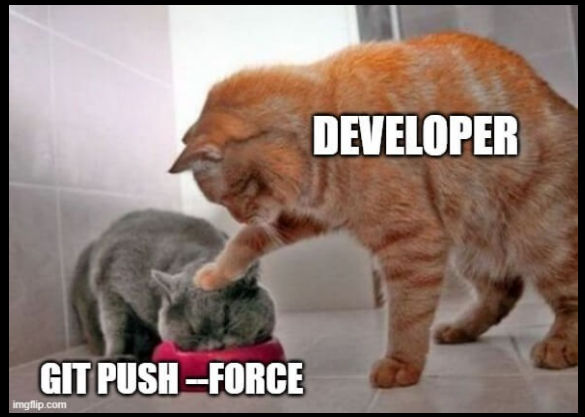
The **git push** command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo. It's the counterpart to **git fetch**, but whereas fetching imports commits to local branches, pushing exports commits to remote branches. Remote branches are configured using the **git remote** command. Pushing has the potential to overwrite changes, caution should be taken when pushing.





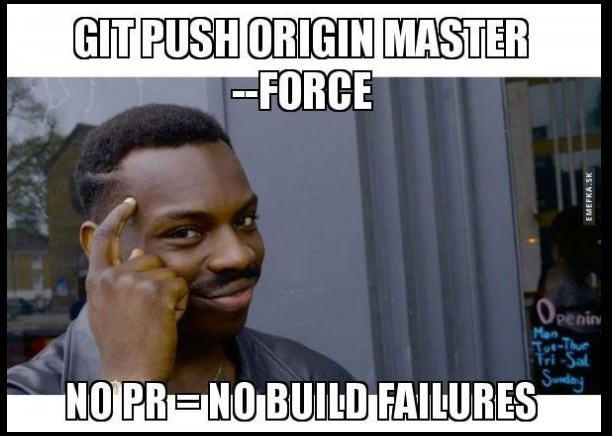
https://git-scm.com/docs/git-push https://www.atlassian.com/git/tutorials/syncing/git-push

git push


DEVELOPER
GIT PUSH –FORCE
imgflip.com


GIT PUSH --FORCE
THESE ARE THE COMMITS YOU'RE LOOKING FOR
made on imgur


When you work with people who don't understand git:
Why are my changes gone?!


GIT PUSH ORIGIN MASTER
--FORCE
NO PR = NO BUILD FAILURES

git push


GIT PUSH --FORCE
PROBLEM SOLVED
makeameme.org


GIT PUSH -F
DON'T MESS WITH HISTORY!
imgflip.com


I KILLED A MAN REWORKED OLD MEME
FORCE PUSH
DUDE, WTF?!
SRGRAFO
Posted in r/SrGrafo by u/LeKian
reddit

git push --force-with-lease

Try to use **git push --force-with-lease** as a more cautious and safer version of git push --force. While both commands allow you to overwrite the remote branch with your local branch, **--force-with-lease** adds a crucial safety mechanism to prevent accidental overwrites, especially in collaborative environments.

Key Concepts and Mechanics:

1) Tracking Branches: When you clone a repository or set up a remote, Git creates tracking branches (e.g., **origin/main** for a local **main** branch tracking the **main** branch on the **origin** remote). These tracking branches store the last known state of the remote branches.

2) Lease Mechanism: **--force-with-lease** works by taking a "lease" on the remote branch. Before attempting the force push, it checks if the remote branch's **HEAD** commit matches the commit that your local tracking branch is pointing to.

3) Conditional Push:

- **If the remote branch's HEAD matches your local tracking branch:** The force push proceeds, overwriting the remote branch with your local branch.
- **If the remote branch's HEAD is different:** The force push is rejected with an error message. This indicates that the remote branch has been updated by someone else since you last fetched, and your force push would potentially overwrite their changes.

Conclusion

- **Version control is like time management** - inevitable but can be mastered with practice
- **Git reflects relationships** - requires understanding, patience, and embracing its distributed nature
- **Choose your branching strategy wisely** - like communication styles, each has its purpose
- **The moral of our story:** Healthy codebases come from healthy version control habits
- **Face conflicts directly** - they're inevitable but resolvable
- **It's okay to stash problems** - but eventually commit to solutions
- **Use reset wisely** - sometimes starting fresh helps (but beware of --hard)
- **Git tracks your growth** - the reflog never forgets, but shows your progress
- **Push thoughtfully** - choose --force-with-lease over --force to respect others' work
- **May your merges be conflict-free, your commits meaningful, and your branches always productive!**

Q&A

Thank You!