Git for Teams

Philipp Reinecke

ReineckeP@cardiff.ac.uk

Version control

“[V]ersion control, also known as revision control or source control, is the management of changes to documents, computer programs, large web sites, and other collections of information.”

https://en.wikipedia.org/wiki/Version_control (accessed 20 February 2018)

“Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.”

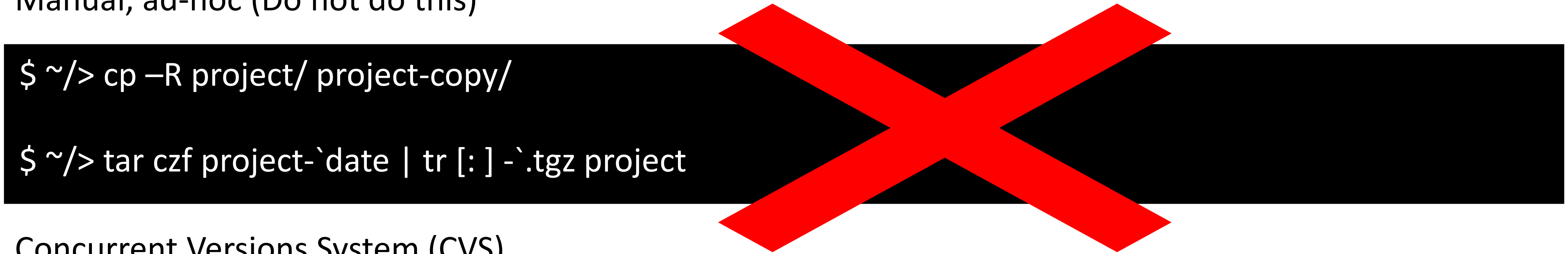
https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control (accessed 20 February 2018)

Why use version control?

- Separation of concerns
    - Let people work in parallel
    - Merge parallel copies
- Quality control: Ensuring project works
- Revision history: Who did what?
- Fixing mistakes: Going back in time

Version control systems

- Manual, ad-hoc (Do not do this)

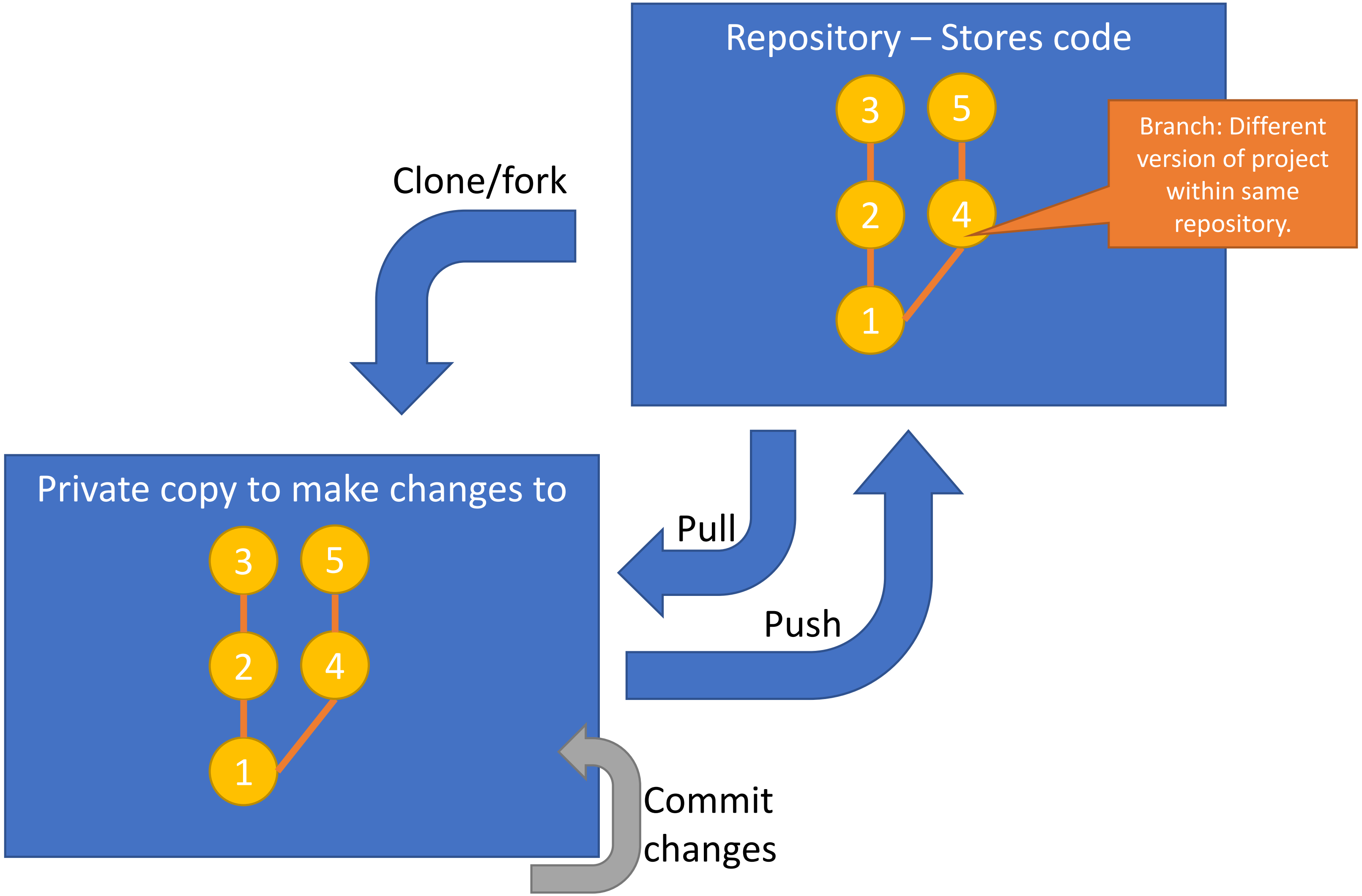


```
$ ~/> cp –R project/ project-copy/

$ ~/> tar czf project-`date | tr [: ] -`.tgz project
```

- Concurrent Versions System (CVS)
  - Developed in 1986
  - Mostly obsolete today
- Apache Subversion (SVN)
  - Developed in 2000
  - Still in use today
- Git
  - Developed in 2005
  - Distributed system
  - Very widely used

Git: General concepts



Repository – Stores code

Branch: Different version of project within same repository.

Clone/fork

Private copy to make changes to

Pull

Push

Commit changes

Version control for teams

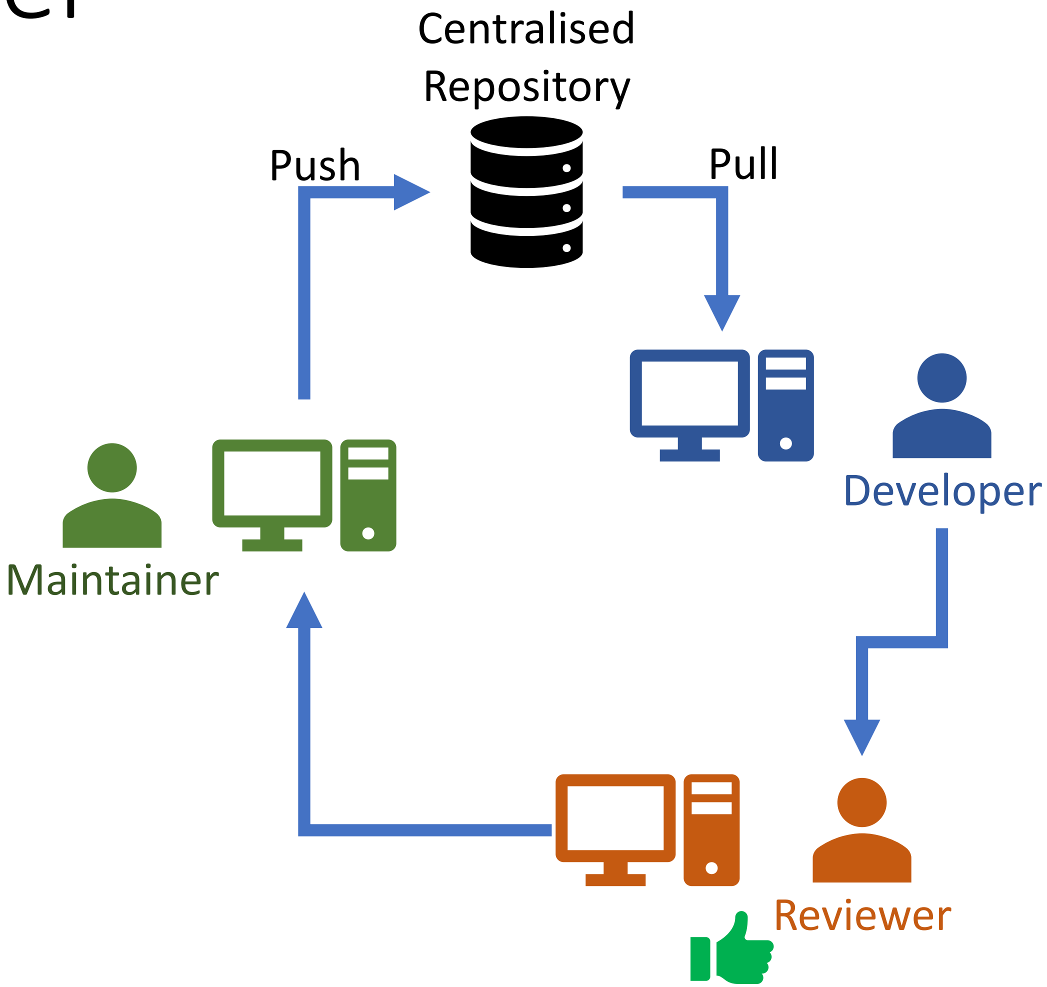
- Who is responsible for the whole project?
  - Management of changes
  - Incorporation of changes
  - Consistency
  - Code quality
  - Reliability
- How do we organise new work?
  - Individual ideas
  - Features
  - Bugs (and fixes)
  - New versions

Who is responsible?

- Can everyone update the code?
- Access-control models
  - Dispersed contributor repositories
  - Collocated contributor repositories
  - Single repository, shared maintenance
- Differences:
  - Way code is shared
  - Who has access to what

Dispersed Contributor Model

- All code stored in centralised repository
- Only maintainer has write access
- Process:
  1. Developer pulls code into local repository
  2. Developer works on local repository
  3. When finished, developer creates a list of changes (“diff”)
  4. Diff sent to reviewer
  5. Reviewer applies diff to code and tests the new version
  6. When the new version works, the diff is submitted to the maintainer
  7. The maintainer updates the code in the repository

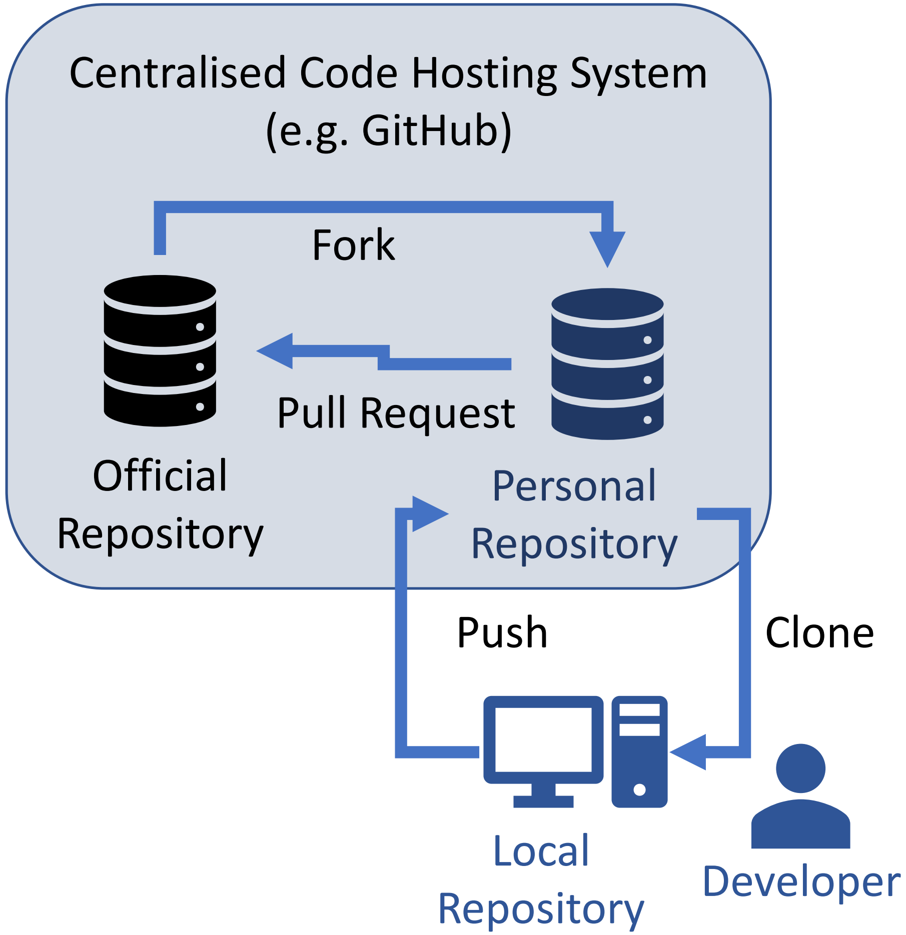


Dispersed Contributor Model

- Archaic – very manual
- Some advantages:
  - Does not require specific version control software
  - Encourages “whole idea” thinking – submit work for review only when it is finished
  - Ensures stability, since all code has to be reviewed

Collocated Contributor Repositories Model

- Centralised Hosting System stores repositories
- Developer forks/clones into personal repository
- Developer clones into local repository to work
- When finished, developer pushes to personal repository
- Developer issues a pull request – request for code to be merged into official repository

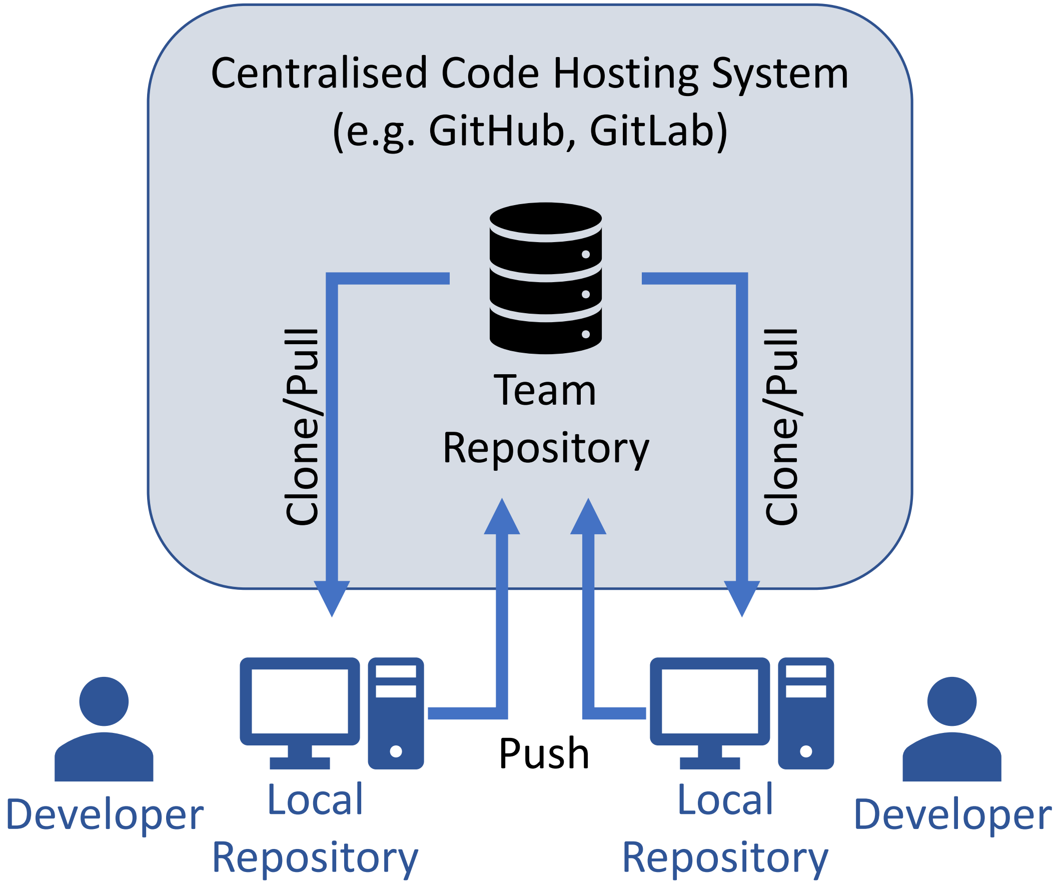


Collocated Contributor Repositories Model

- “GitHub” model
- Very suitable for Open-Source projects
  - Anyone can contribute
  - Contributions only included if maintainers approve pull request
- Also suitable for internal projects with strict quality assurance
  - Changes only get included if QA team agrees

Shared Maintenance Model

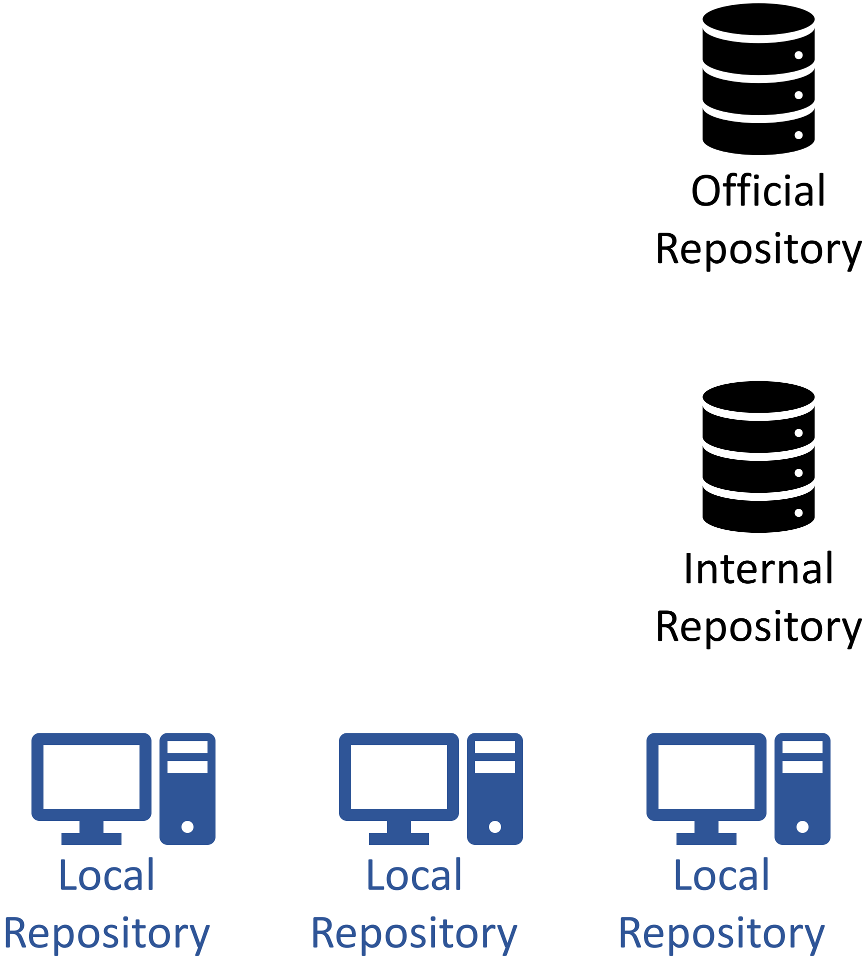
- Central hosting system stores team repository
- Developers pull code to local repositories to work on
- When finished, code is pushed into team repository



Shared Maintenance Model

- Every developer has write access
- Inherent trust amongst team members
- Assumptions:
  - Code is checked and verified before committing to main branch
  - Developers can be trusted
- Appropriate for internal teams

Combined model



Official Repository

Write-access only for maintainers or QA team

Internal Repository

Shared maintenance: Access for all developers

Local Repository

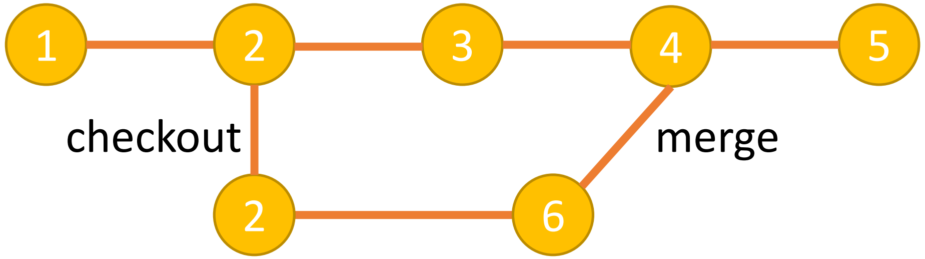
Local Repository

Local Repository

Access for individual developers

Branching Strategies

- Branches: In-repository split where new work begins

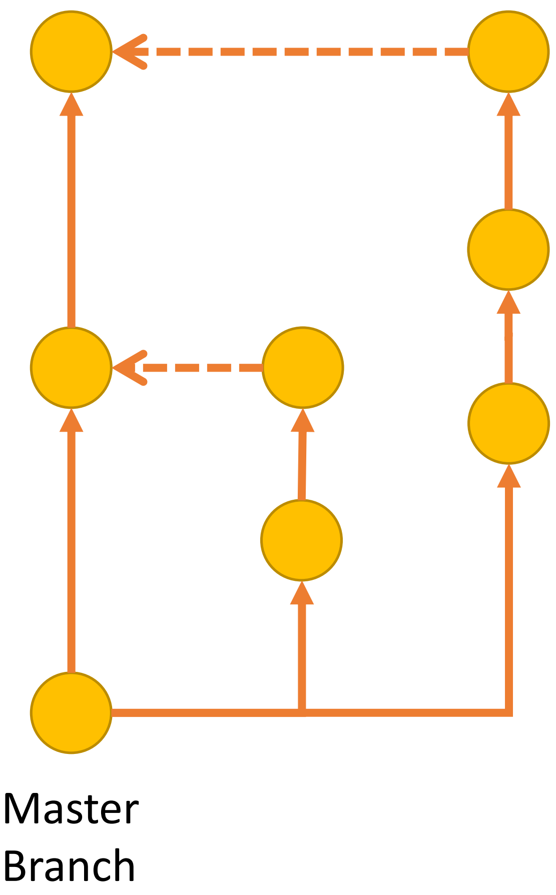


Using Branching

- By definition, branches are just splits
- Semantics depend on use
- Typical convention:
  - Long-lived branches are public
  - Short-lived branches are private for individual developers
- Closely tied to deployment strategies
- Strategies:
  - Mainline branching
  - Branch-per-feature
  - State branching

Mainline branching

- One single, central branch
  - Always deployment-ready
  - Only contains tested code
- Developers branch off for new ideas or features and merge back in

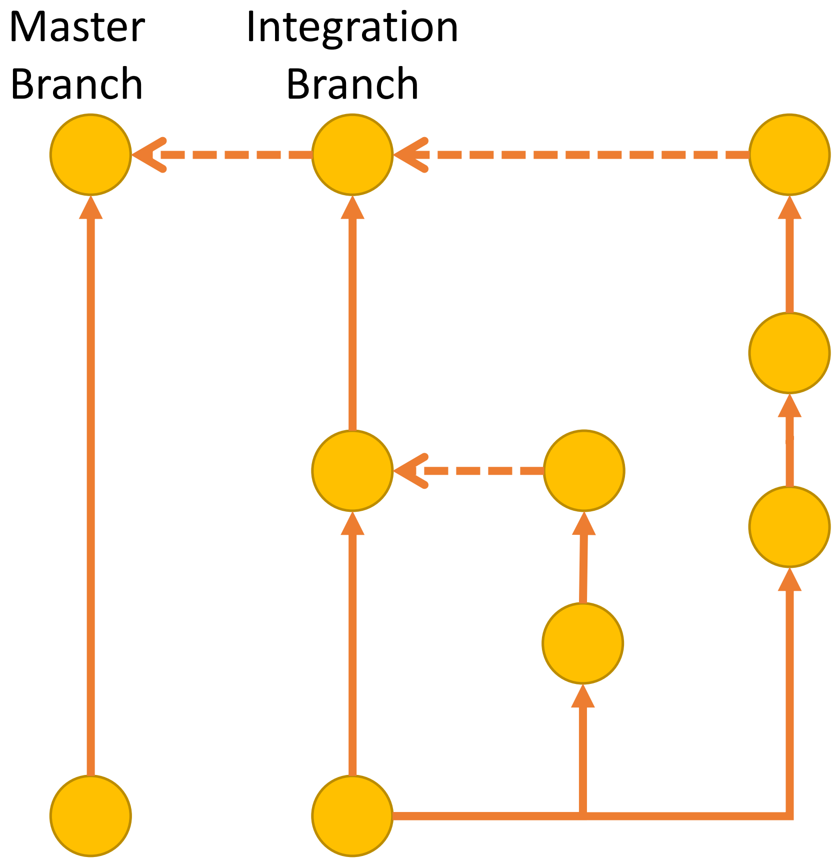


Mainline branching

- Encourages regular integration
- Suitable for continuous deployment: Regularly updates project
- Advantages:
  - Not very many branches – less confusion
  - Small commits – easy debugging
  - Any code in main branch is ready for deployment – few emergency fixes needed
- Main disadvantage:
  - Risky – without thorough testing, project can be broken

Branch Per Feature

- One branch for each feature
- Feature branch: Should contain only one idea
- Integration branch:
  - Synchronises work
  - Integrates all features
- Deployment: Master branch built by selecting features from integration branch

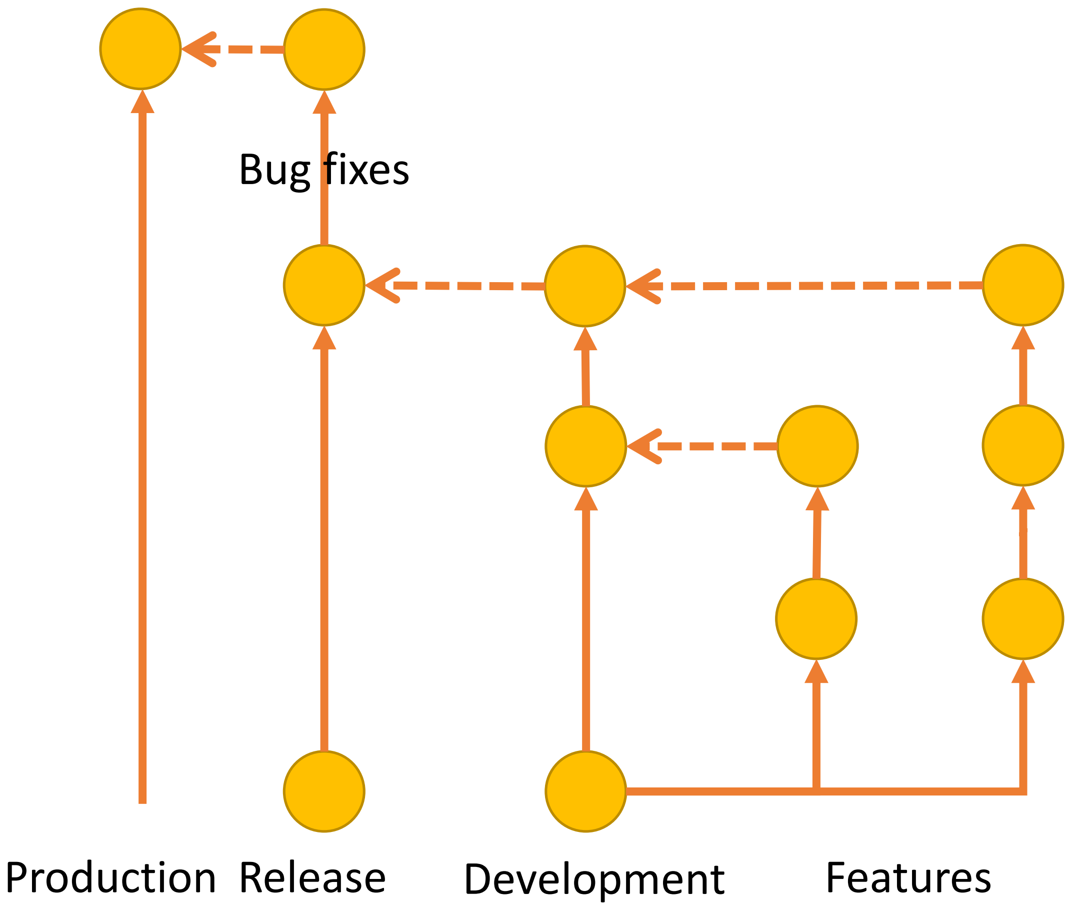


Branch Per Feature

- Deployment-ready code always available in master branch
- Advantages:
    - Rapid deployment possible
    - Optional build step – only selected features get deployed
- Disadvantages:
    - Old branches must be removed once they are merged into master branch
    - Code in feature branch must be kept up-to-date with master branch
    - Naming of branches may be confusing

State Branching and Scheduled Deployment

- Dev branch contains ongoing development
- "Feature freeze":
  - Release branch created
  - No new features, only bug-fixes
- Fixed code goes into Production branch

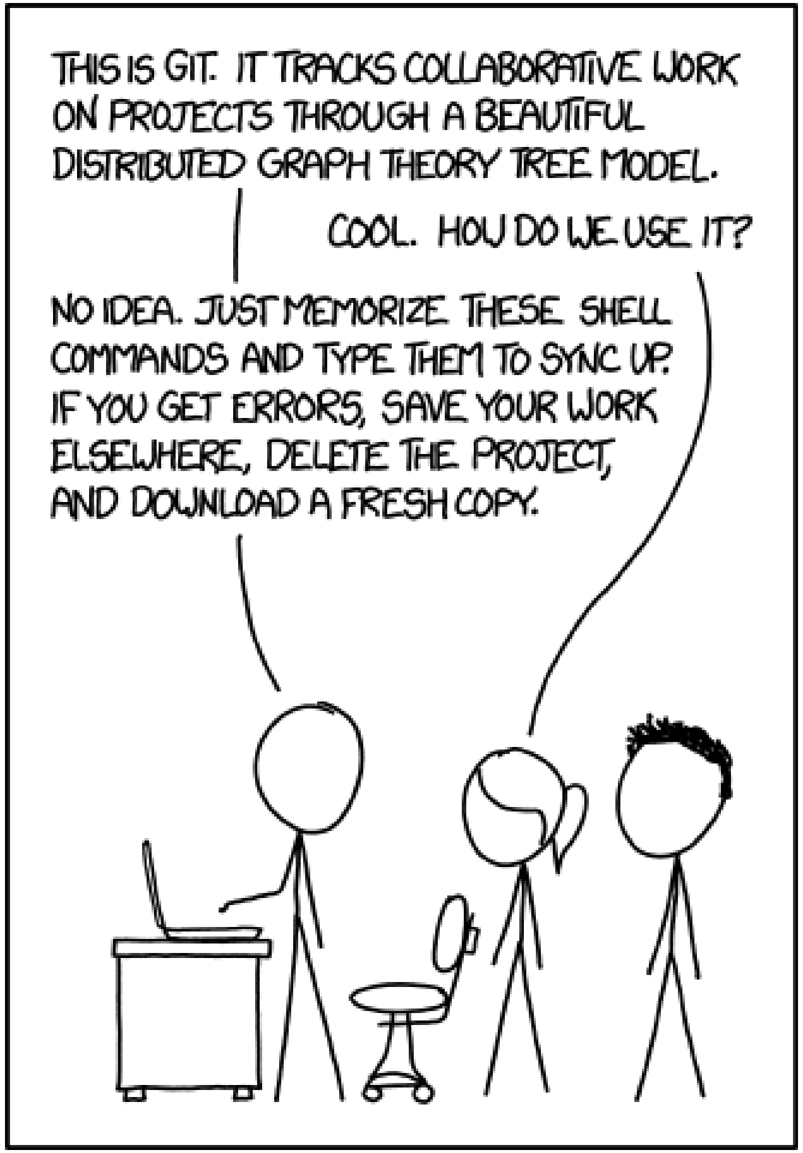


State Branching and Scheduled Deployment

- Useful for scheduled deployment, e.g. milestones
- Advantages:
  - Branch-names context specific and clear
  - Can always select correct branch (typically, Development)
  - Production code is always stable
- Disadvantages:
  - Not always obvious where to start from
  - Meaning of branches can be very specific

Conclusion

- Version control is important for teamwork
  - Git can support good development habits
  - Git can support team organisation
- Choose appropriate model
  - Dispersed contributor
  - Collocated contributor
  - Shared maintenance
  - Combined model?
- Choose appropriate branching strategy
  - Mainline branching
  - Branch per feature
  - State branching/scheduled deployment
- Choices depend on
  - Team size & dynamics
  - Team culture
  - Trust amongst team members


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.

https://xkcd.com/1597/ (Creative Commons Attribution-NonCommercial 2.5 License)

Sources and further reading

- Emma Jane Hogbin Westby: Git for Teams (O’Reilly 2015)
- Scott Chacon, Ben Straub: Pro Git (https://git-scm.com/book/en/v2)
- Git cheat sheets, e.g.
  - https://www.git-tower.com/blog/git-cheat-sheet/
  - http://rogerdudler.github.io/git-guide/
- Helpful software/platforms:
  - GitHub – for public projects
  - GitLab – for private projects