
In case of fire
1. git commit
2. git push
3. leave building

CPEN 321

Teamwork, Version Control

Yingying Wang, UBC

Software Development – a Team Effort

- A number of teams work in conjunction with others toward a common goal: developing a product that will provide the best experience possible for end users
- Communication and collaboration are the key to success of the teams

How to deliver a consistently high-quality system?

- Continuous integration and testing
- Communication, communication, communication!
- Version control: a system that records changes to files over time
  - So a number of different users can change the files
  - So one can recall a specific version later

How to deliver a consistently high-quality system?

- Continuous integration and testing
- Communication, communication, communication!
- **Version control: a system that records changes to files over time**
  - **So a number of different users can change the files**
  - **So one can recall a specific version later**

What to track?

- Source code
  - As opposed to generated files
- Tests
- Docs
- Configuration files
- …

Benefits of Version Control

- Allows team members to work concurrently and independently
- Makes it possible to go back to the last working version
- Connects changes made to the system to project management and bug tracking software such as Jira
- Annotates each change with a message describing the purpose and rationale of the change
- Supports continuous integration

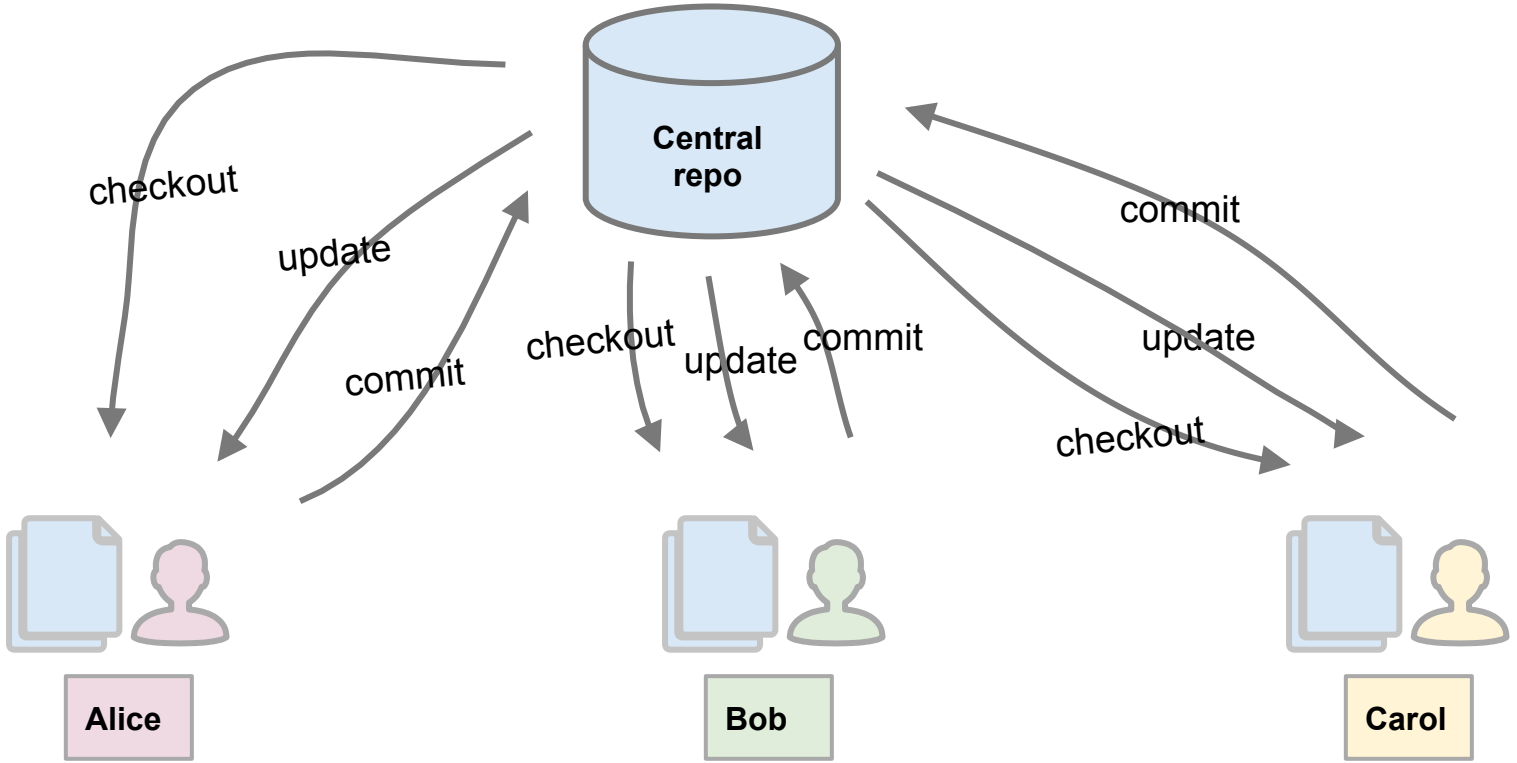
Version Control Systems: Outline

- Types of Version Control Systems
- Git
  - Branching
  - Merging
  - Rebasing
  - Squashing
  - Cherry-pick
  - Remote repositories
- GitHub
  - Cloning vs. Forking
  - Making pull request
- Branching strategies

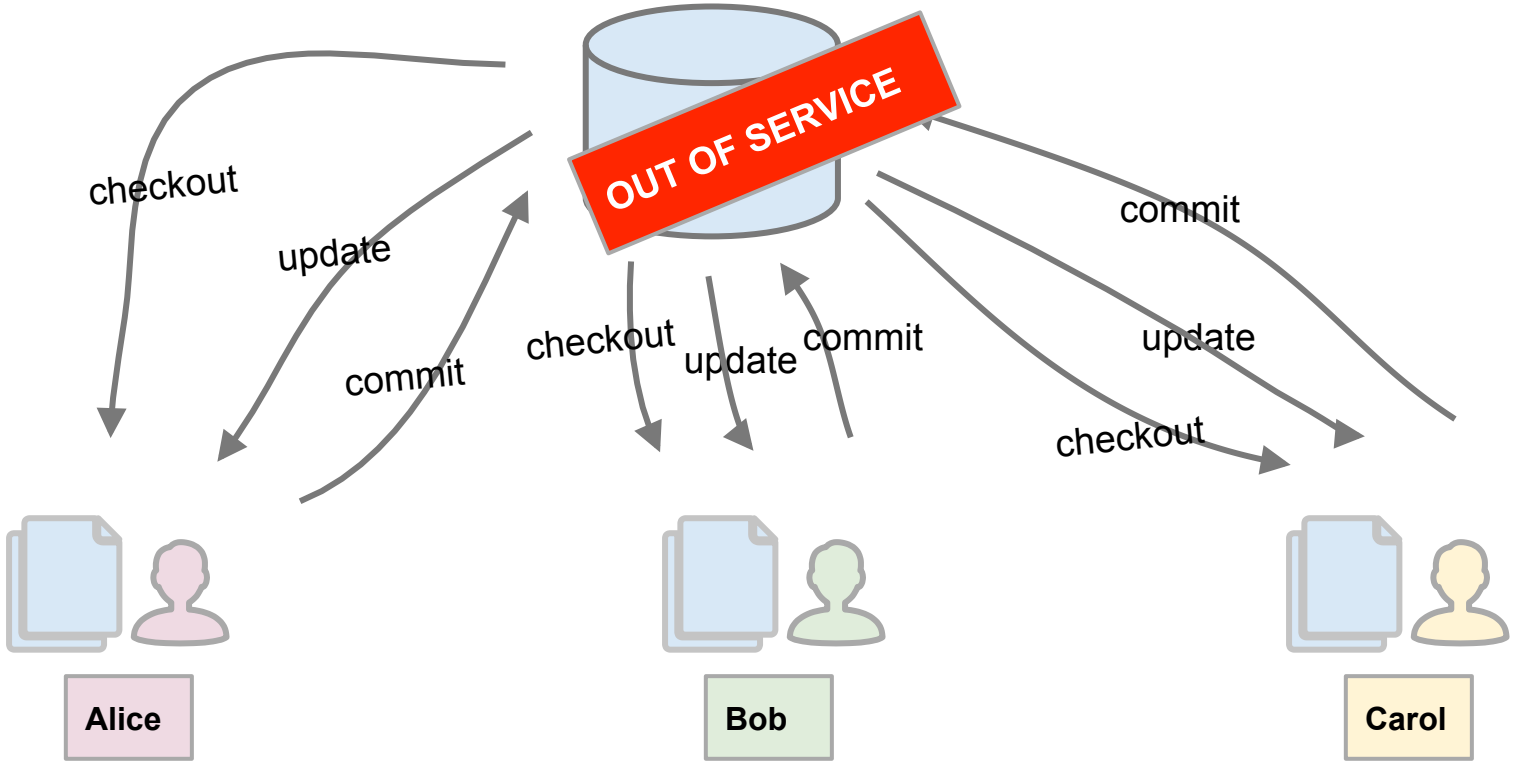
Centralized Version Control Systems (e.g., SVN, CVS)

- A central repository contains all data and histories
- All commits are made to the central repo
- Each developer only has a snapshot of the repo

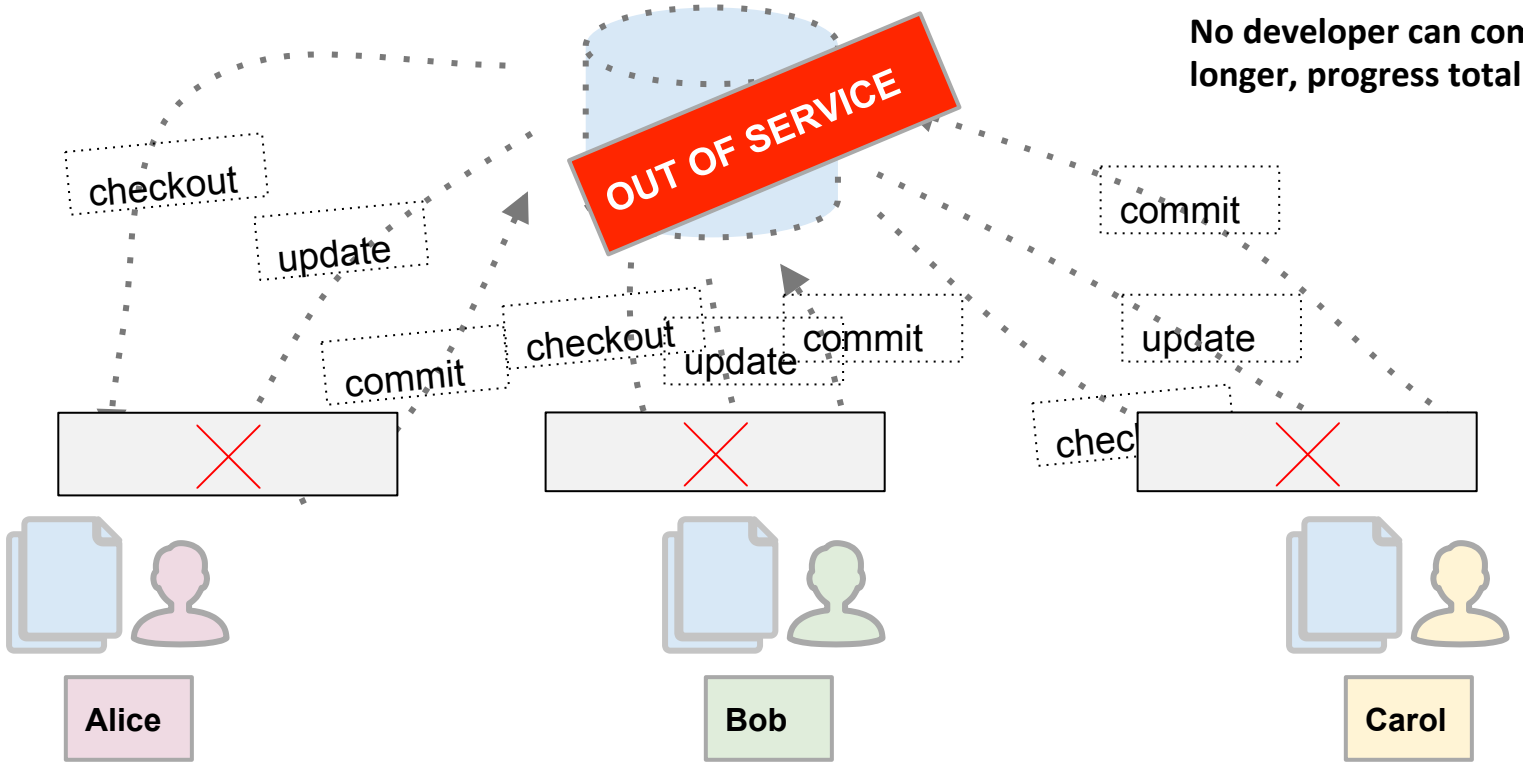
Centralized Version Control Systems (e.g., SVN, CVS)



Centralized Version Control Systems (e.g., SVN, CVS)



Centralized Version Control Systems (e.g., SVN, CVS)

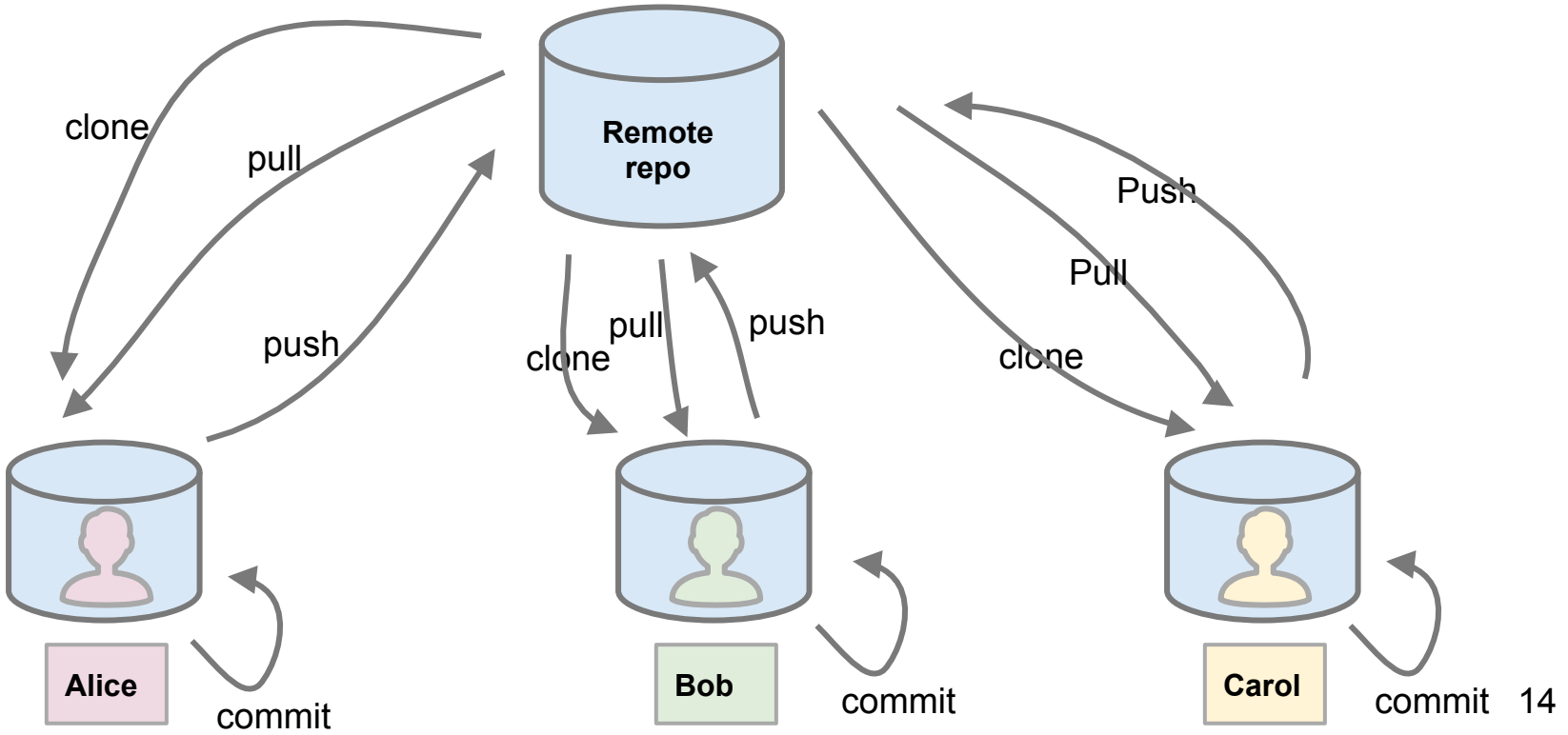


No developer can commit any longer, progress totally halted.

Centralized Version Control Systems (e.g., SVN, CVS)

- Pros:
  - Provide developers with a clear view (everyone knows at some level what the others do)
  - Allows file locking, avoids conflicts
- Cons:
  - If the main server goes down, the developers can’t save versioned changes (single point of failure)
  - Developers cannot keep track of their own changes without sharing with others

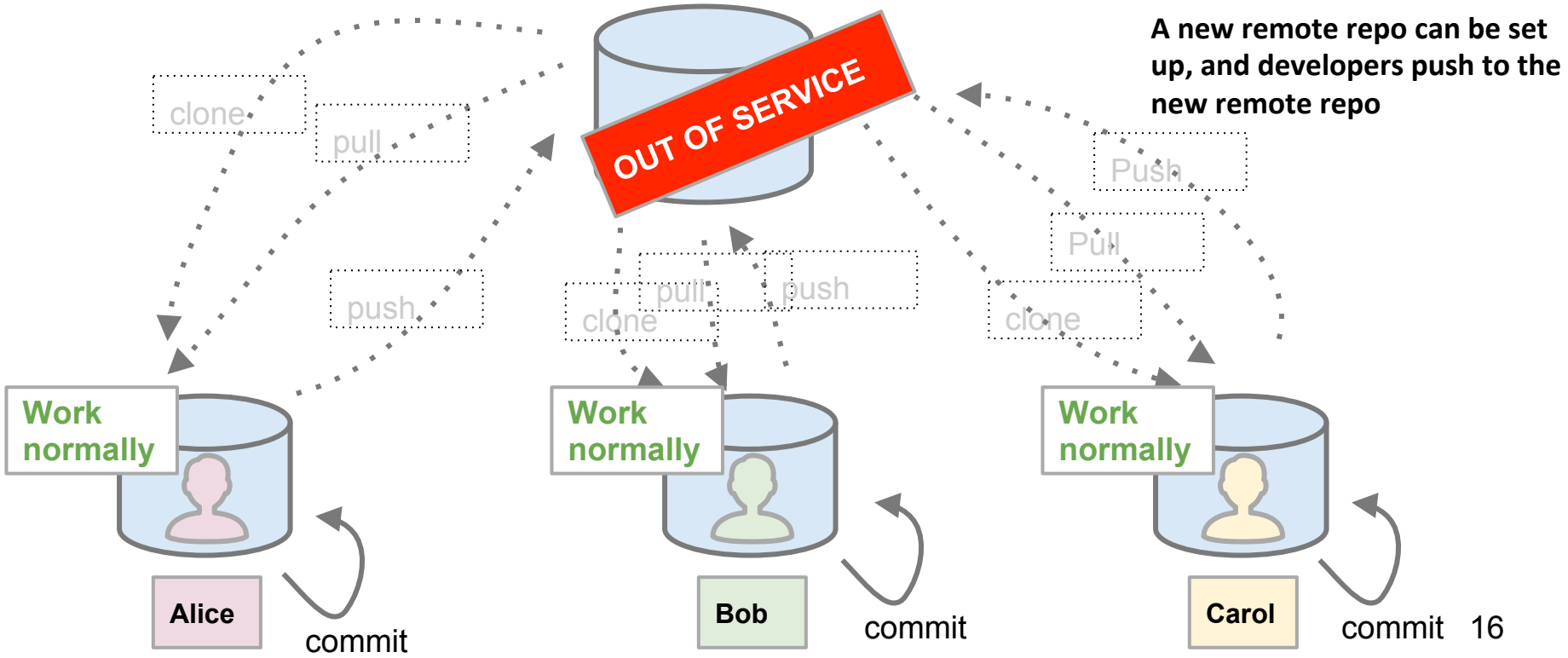
Distributed Version Control Systems (e.g., git, mercurial)



Distributed Version Control Systems (e.g., git, mercurial)

- In distributed version control systems, each copy of the repo is a full repository
  - Includes data of current version and full history of the repo
- In developers' own repo
  - They can commit locally
  - If they want their commits visible to others, then push to the remote repo.
- There is not centralized repo, changes could go to any remote repo
  - E.g., Alice can commit changes to Bob's repository

Distributed Version Control Systems (e.g., git, mercurial)



Distributed Version Control Systems (e.g., git, mercurial)

- Pros:
  - You do local commits. The full history is always available.
  - You don’t need to access a remote server and that makes you work very fast, especially for remote, off-shore developers .
  - You can commit your changes continuously.
  - Flexible: different workflow types
- Cons:
  - More complex synchronization mechanism
  - Requires a large amount of space when working with a lot of binary files that cannot be easily compressed

Version Control Systems: Outline

- Types of Version Control Systems
- Git
  - Branching
  - Merging
  - Rebasing
  - Squashing
  - Cherry-pick
- GitHub
  - Cloning vs. Forking
  - Making pull request
- Branching strategies

Git

- ***Git* is the most popular distributed VCS**

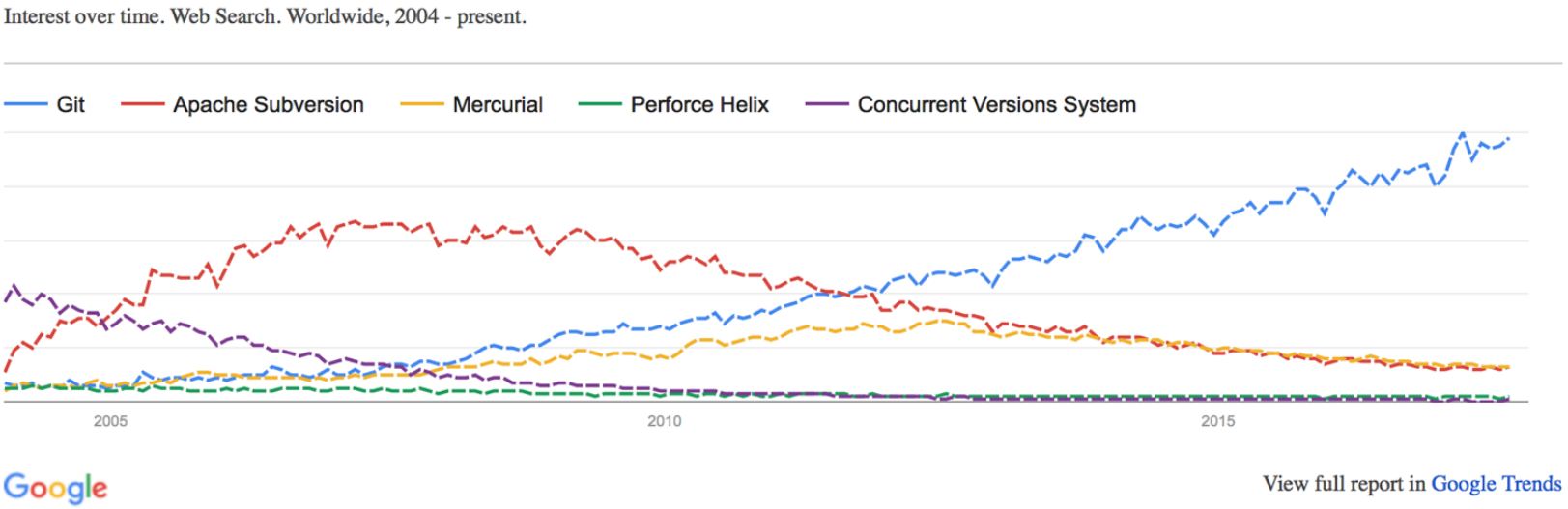
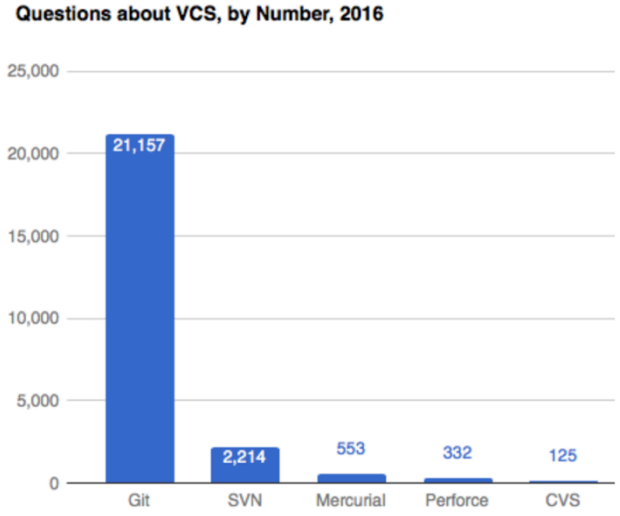


Figure1: Google Trends report on version control systems till 2016

ref: https://rhodecode.com/insights/version-control-systems-2016

Git

- ***Git* is the most popular distributed VCS**



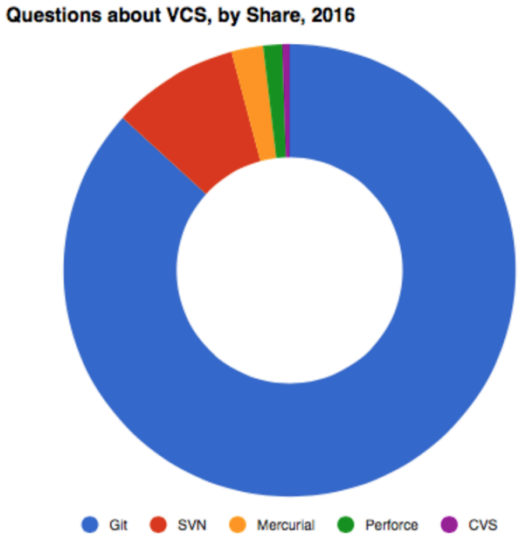


Figure2: Stackoverflow's statistics of questions about VCS

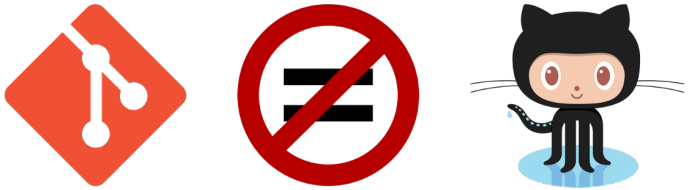
https://rhodecode.com/insights/version-control-systems-2016

Git vs. GitHub (Gitlab, Bitbucket, etc.)

- **Git** is the **version control system**, a **tool** to manage source code history.

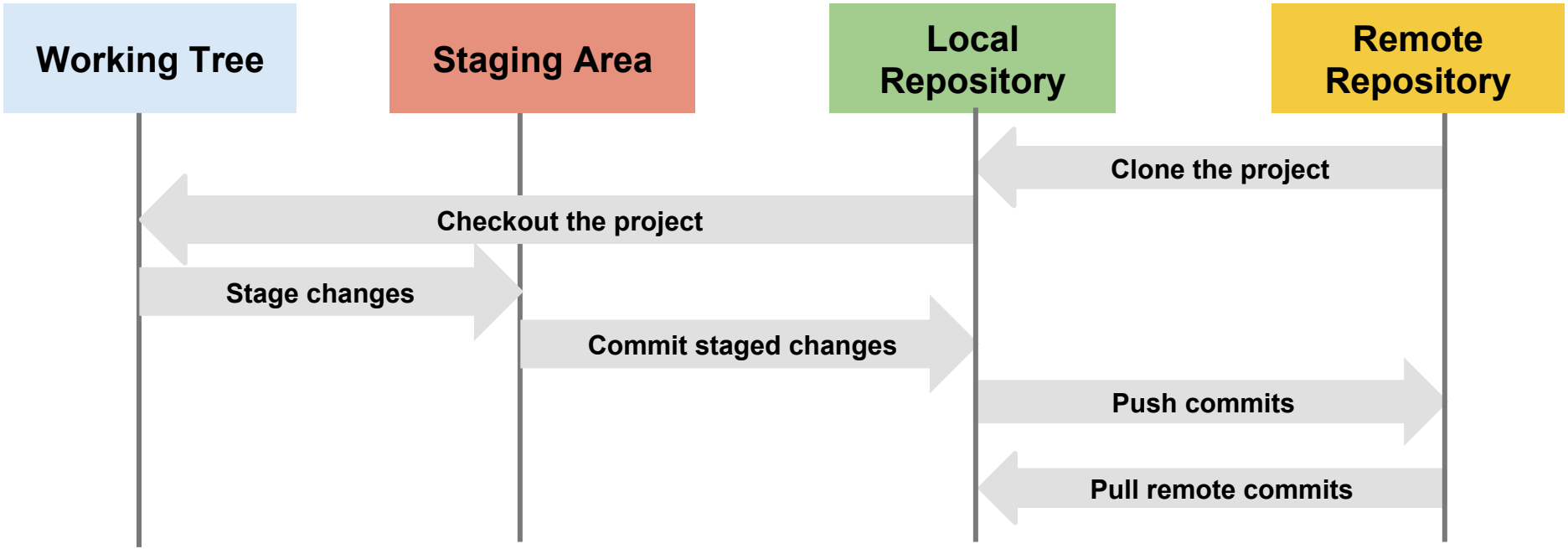
  Commands such as ***git commit, git push, git checkout***, etc.
- **GitHub** is a hosting service for Git repositories. Replacements are

  Bitbucket, Gitlab, etc.

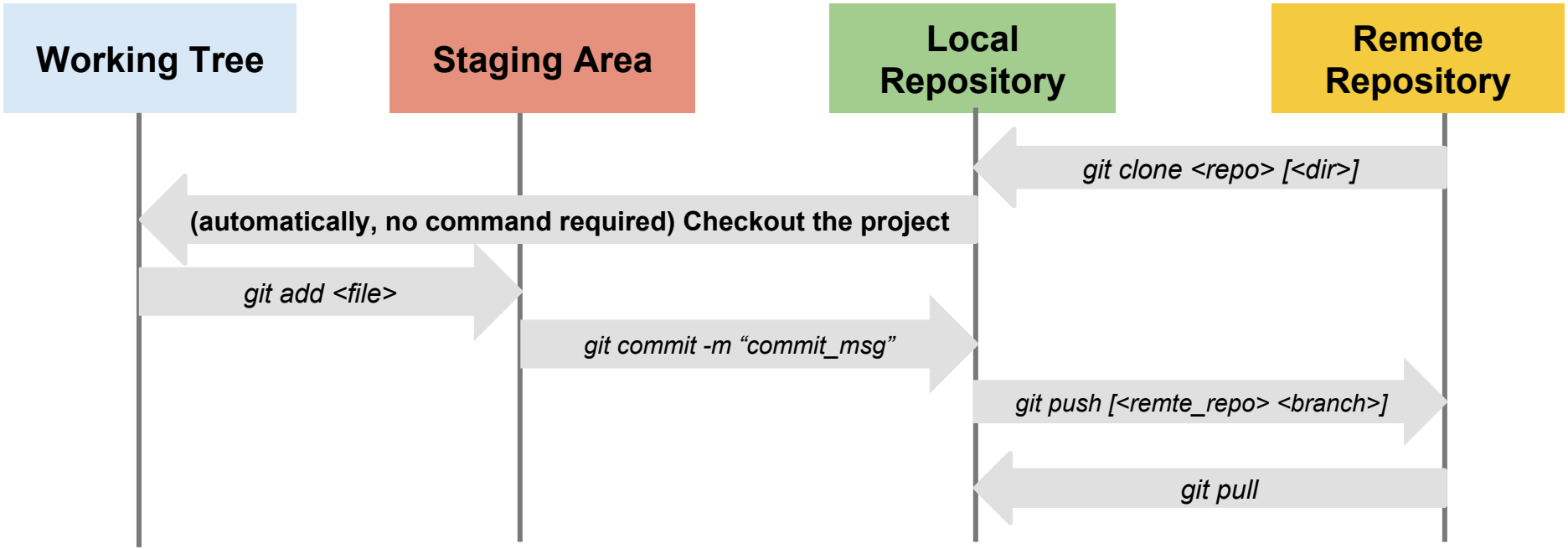


- **Git** is the **tool**, **GitHub** is **the service for projects that use Git**.

Git in a Nutshell



Git in a Nutshell (command)



Git demo

https://goo.gl/xiwqkZ

Version Control Systems: Outline

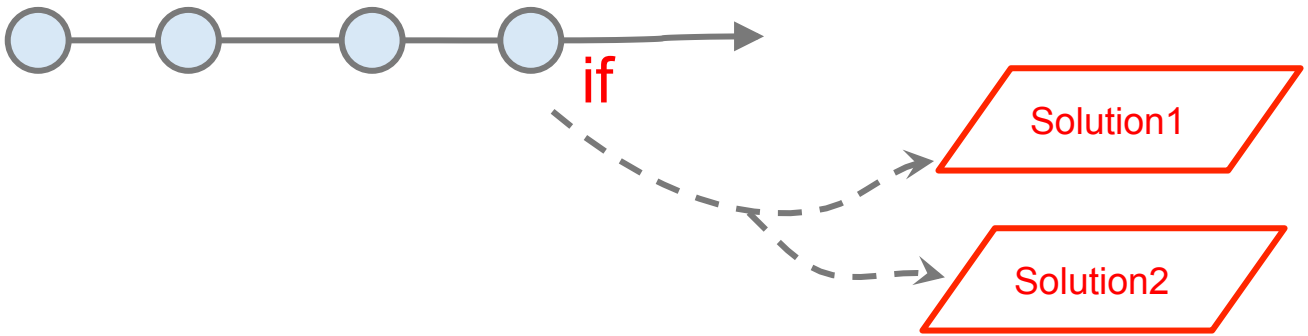
- Types of Version Control Systems
- Git
  - Branching
  - Merging
  - Rebasing
  - Squashing
  - Cherry-pick
  - Remote repositories
- GitHub
  - Cloning vs. Forking
  - Making pull request
- Branching strategies

Branching – why branching

Scenario1: Suppose we are dealing with a problem, which has two possible solutions:

- Solution1
- Solution2

But we are not sure which solution can work out/ work better



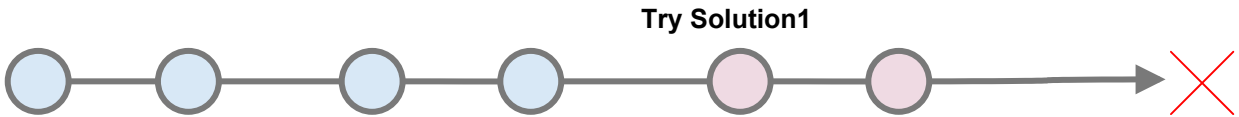
Branching – why branching

Scenario1: Suppose we dealing with a problem, which has two possible solutions:

- Solution1
- Solution2

But we are not sure which solution can work out/ work better

Option1: try one solution, if not working out, roll back, try another solution



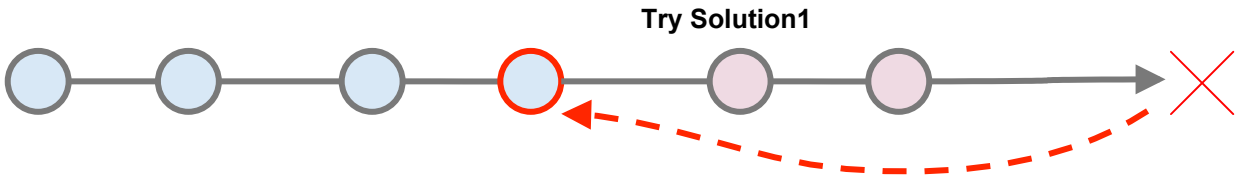
Branching – why branching

Scenario1: Suppose we dealing with a problem, which has two possible solutions:

- Solution1
- Solution2

But we are not sure which solution can work out/ work better

Option1: try one solution, if not working out, roll back, try another solution



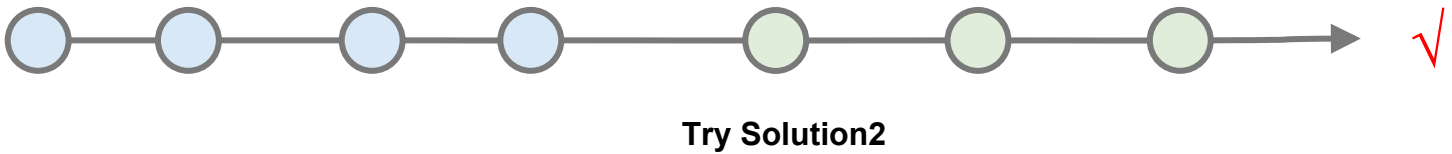
Branching – why branching

Scenario1: Suppose we dealing with a problem, which has two possible solutions:

- Solution1
- Solution2

But we are not sure which solution can work out/ work better

Option1: try one solution, if not working out, roll back, try another solution



Try Solution2

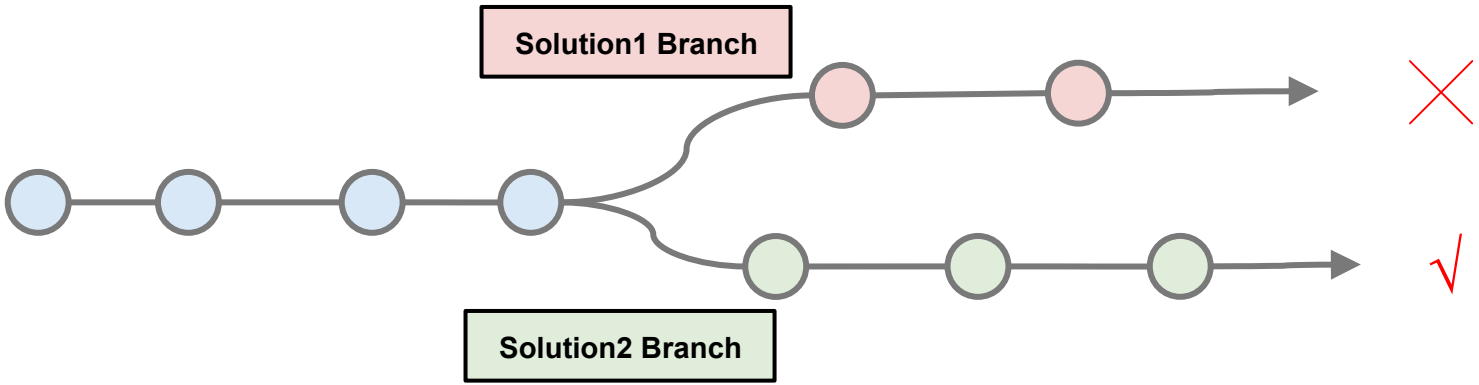
Branching – why branching

Scenario1: Suppose we dealing with a problem, which has two possible solutions:

- Solution1
- Solution2

But we are not sure which solution can work out/ work better

Better option: create 2 branches to test each solution in parallel

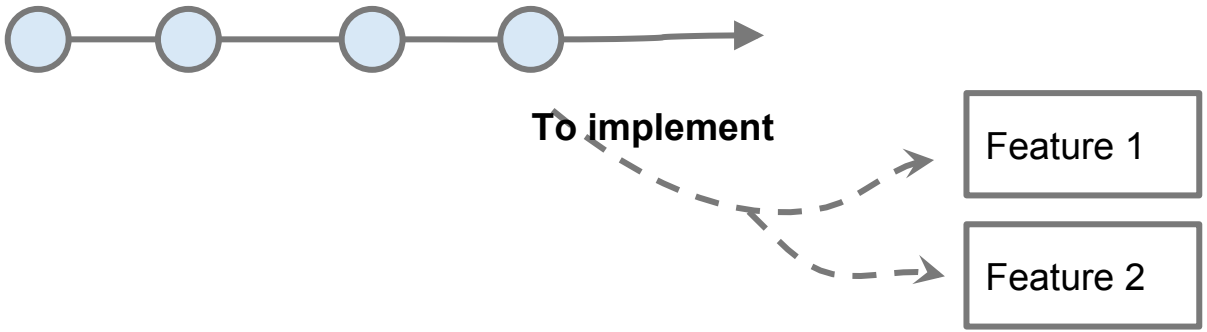


Branching – why branching

Scenario2: Suppose we have two features to implement for one system:

- Feature1
- Feature2

Feature1 and feature2 are totally isolated, how can development be efficient?



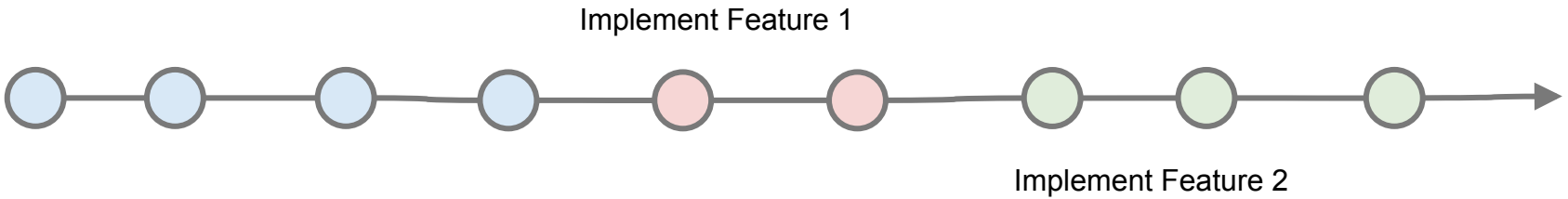
Branching – why branching

Scenario2: Suppose we have two features to implement for one system:

- Feature1
- Feature2

Feature1 and feature2 are totally isolated, how can development be efficient?

Option1: implement one by one → not efficient



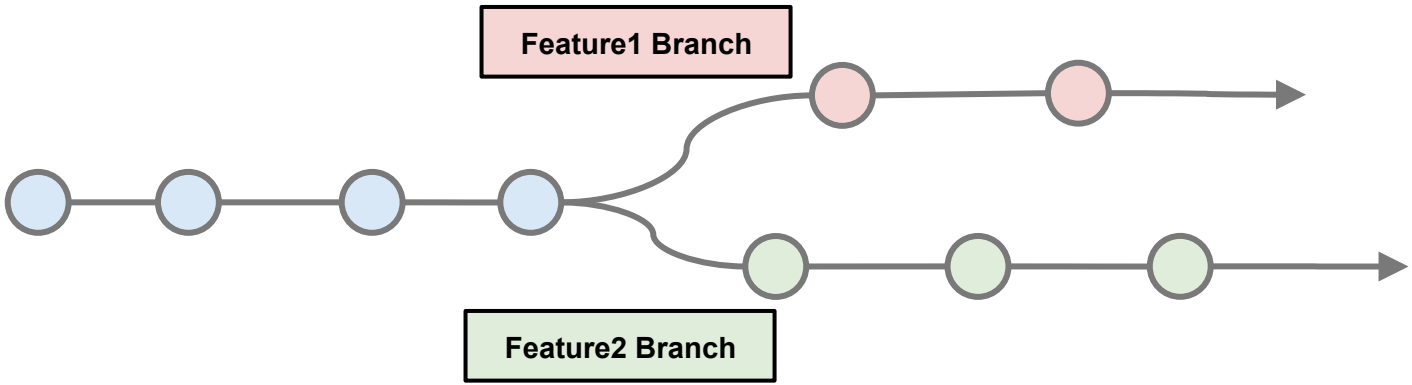
Branching – why branching

Scenario2: Suppose we have two features to implement for one system:

- Feature1
- Feature2

Feature1 and feature2 are totally isolated, how can development be efficient?

Better option: create one branch for each feature, multiple developers can work in parallel

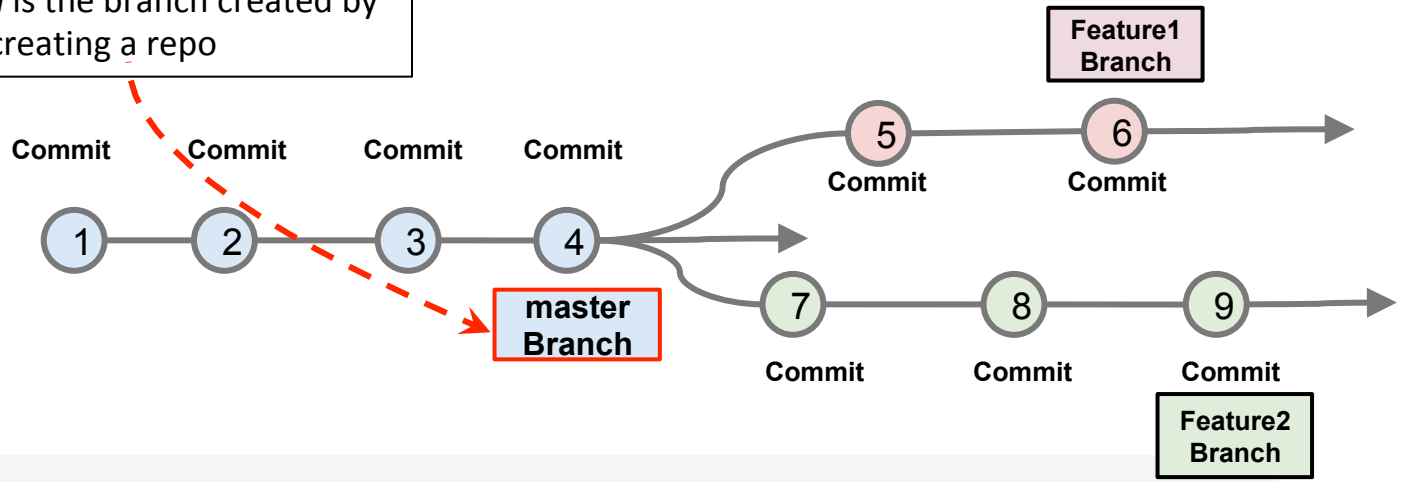


Branching – why branching

- Branching is used to achieve code **isolation**
- You should branch whenever you cannot pursue and record two development efforts in one branch
- Will talk about specific branching strategies later!

Branching – master branch

master Branch is the branch created by default when creating a repo



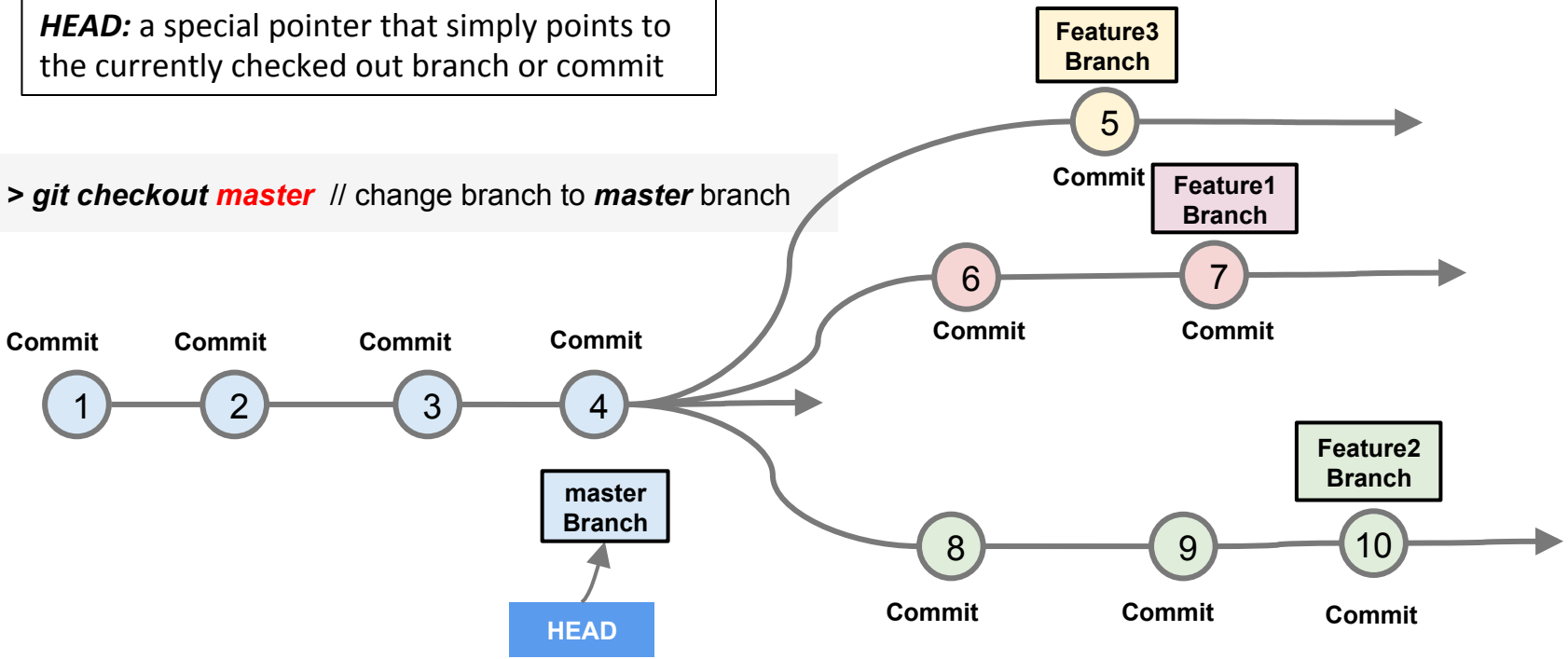
Common commands about branches

> git branch // list all local branches, current checked-out branch has a "*"
> git branch <branch_name> // create a new branch with <branch_name>
> git checkout <branch_name> // change working branch to <branch_name>
> git checkout -b <branch_name> // create and checkout to a new branch with <branch_name>

Branching – HEAD pointer

HEAD: a special pointer that simply points to the currently checked out branch or commit

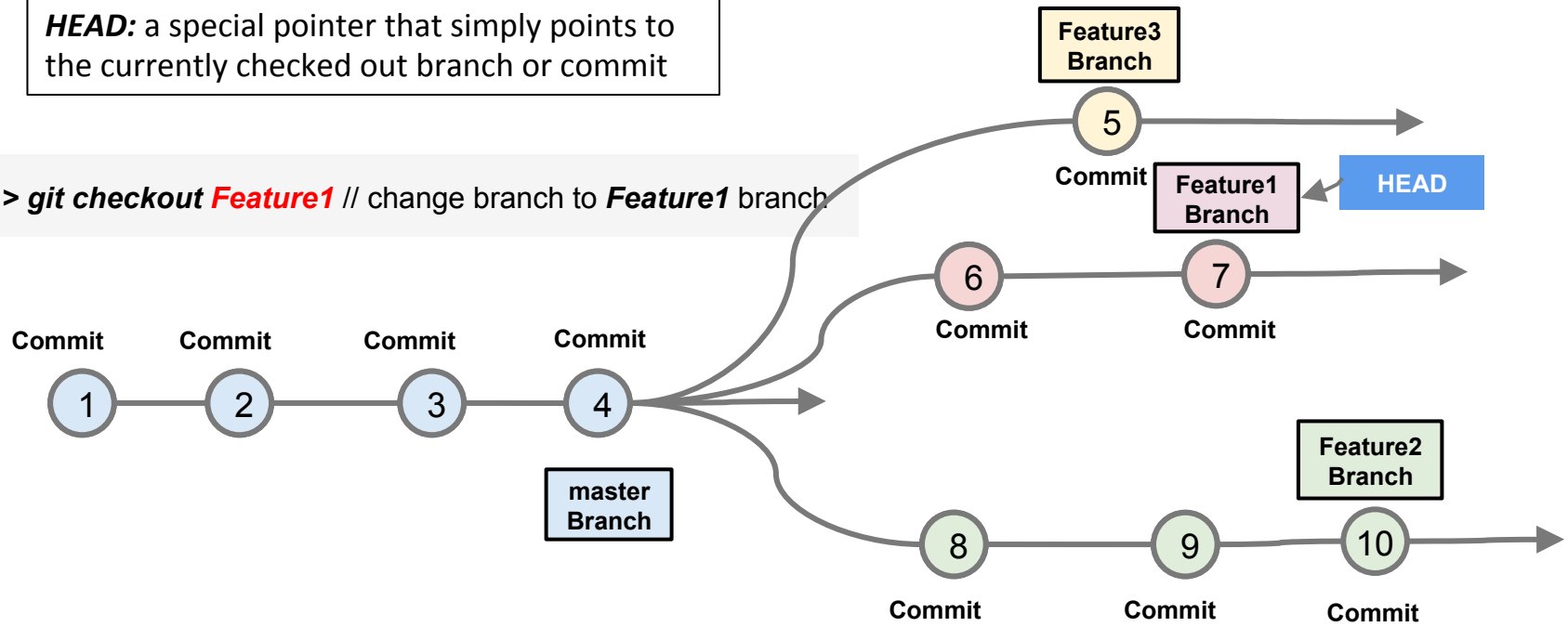
> *git checkout master* // change branch to ***master*** branch



Branching – HEAD pointer

HEAD: a special pointer that simply points to the currently checked out branch or commit

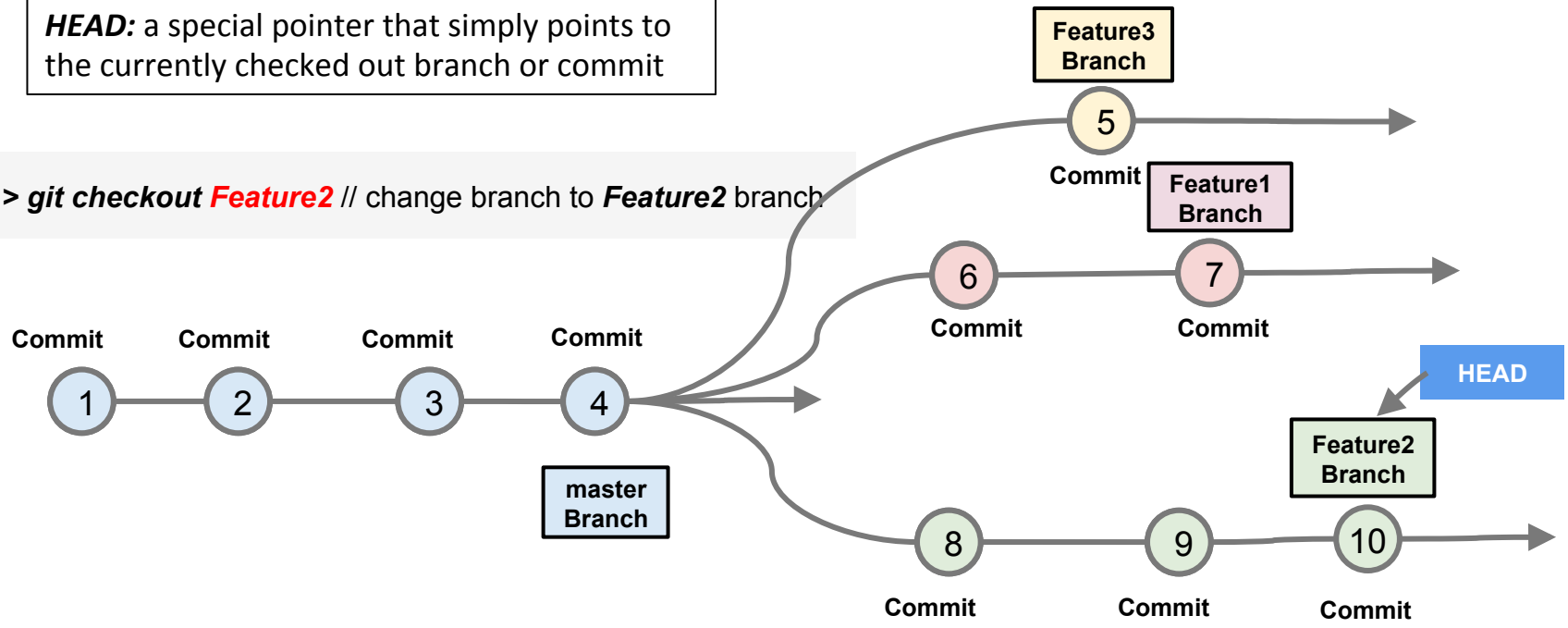
> ***git checkout Feature1*** // change branch to ***Feature1*** branch



Branching – HEAD pointer

HEAD: a special pointer that simply points to the currently checked out branch or commit

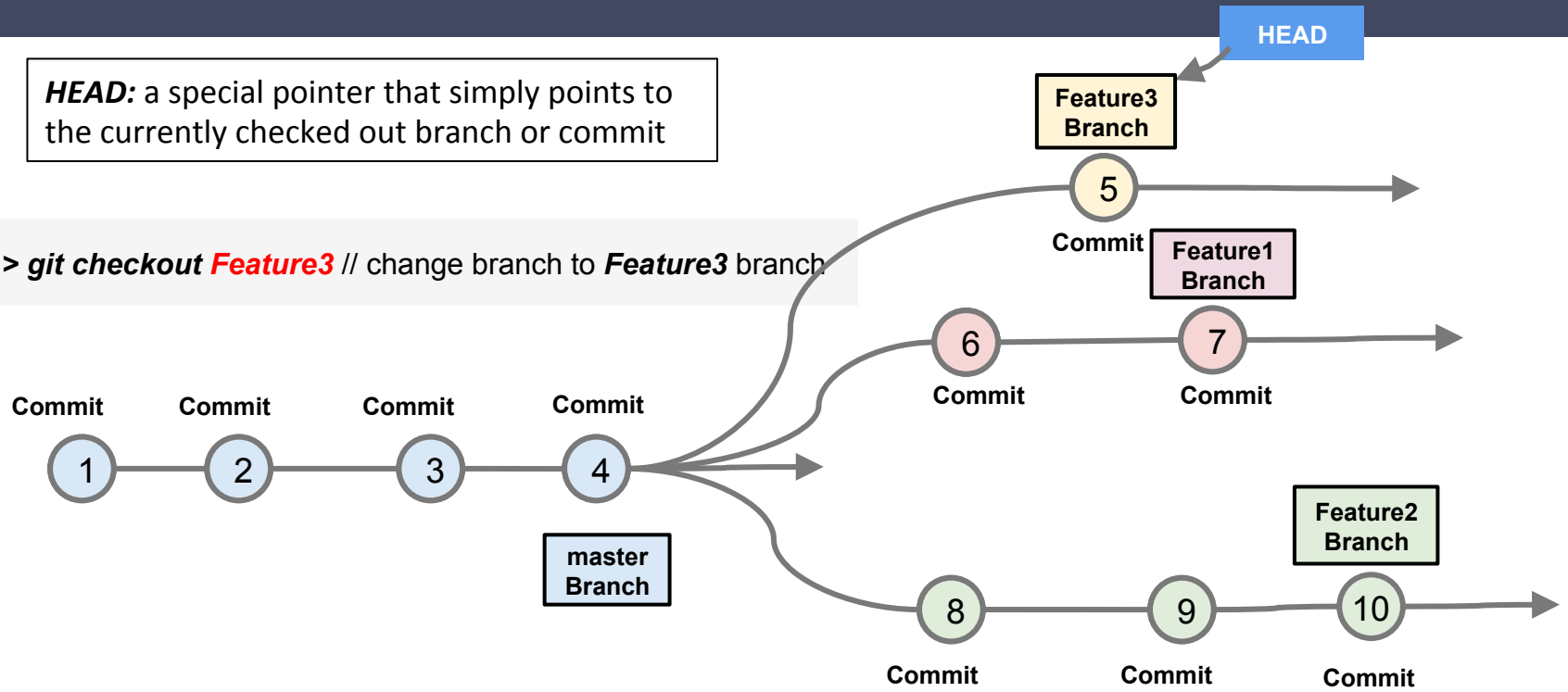
> ***git checkout Feature2*** // change branch to ***Feature2*** branch



Branching – HEAD pointer

HEAD: a special pointer that simply points to the currently checked out branch or commit

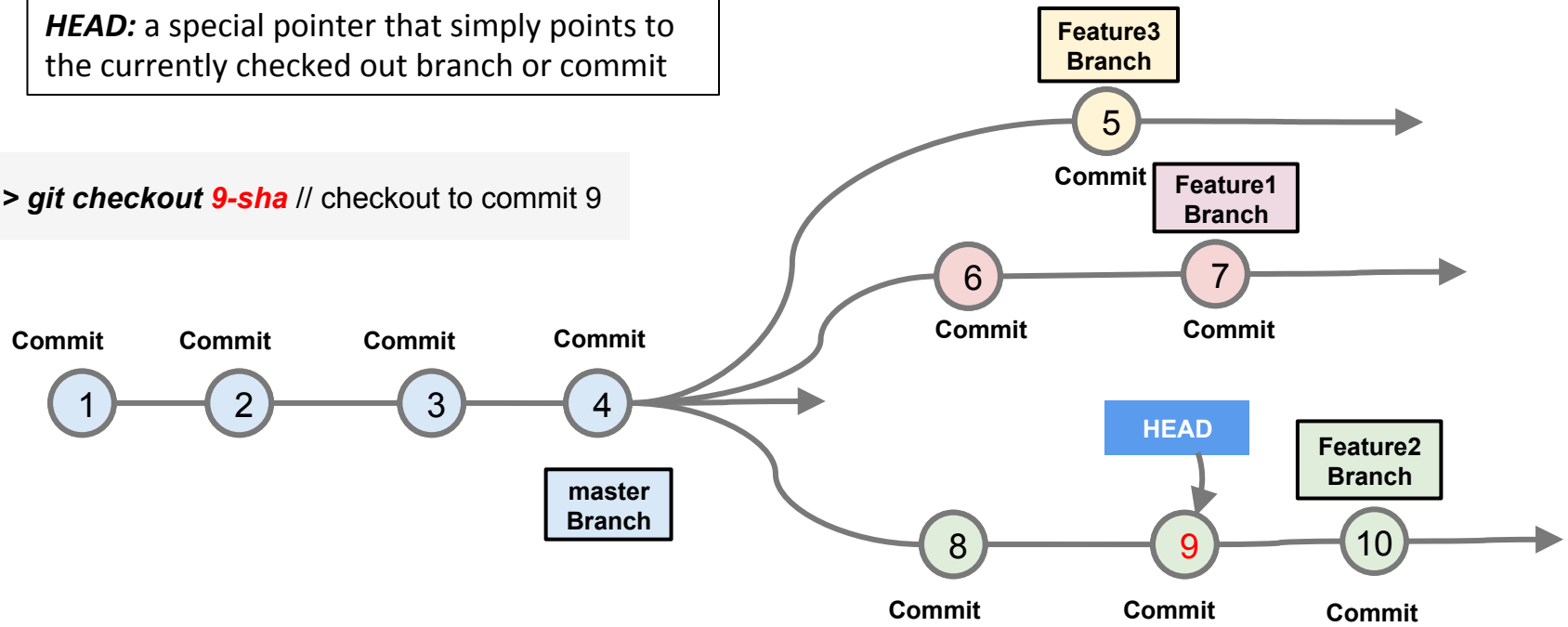
> ***git checkout Feature3*** // change branch to ***Feature3*** branch



Branching – HEAD pointer

HEAD: a special pointer that simply points to the currently checked out branch or commit

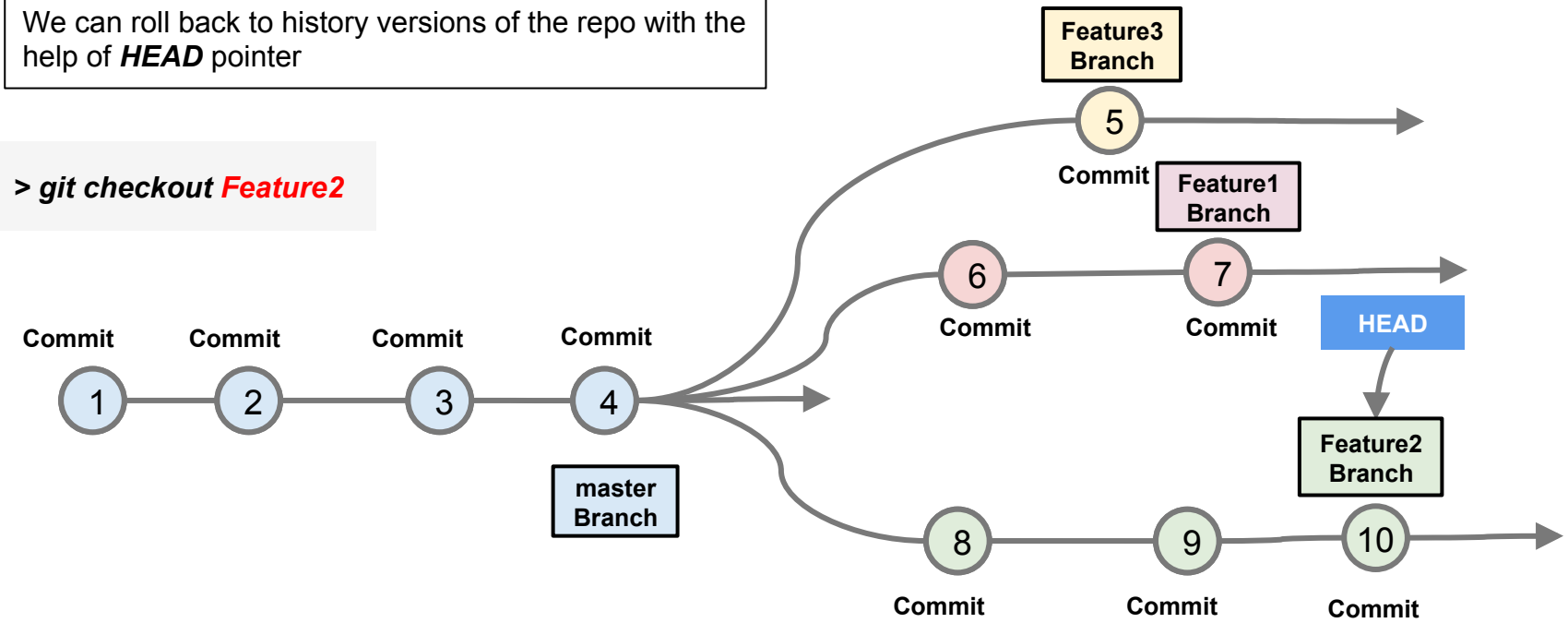
> *git checkout 9-sha* // checkout to commit 9



Branching – roll back to history version

We can roll back to history versions of the repo with the help of ***HEAD*** pointer

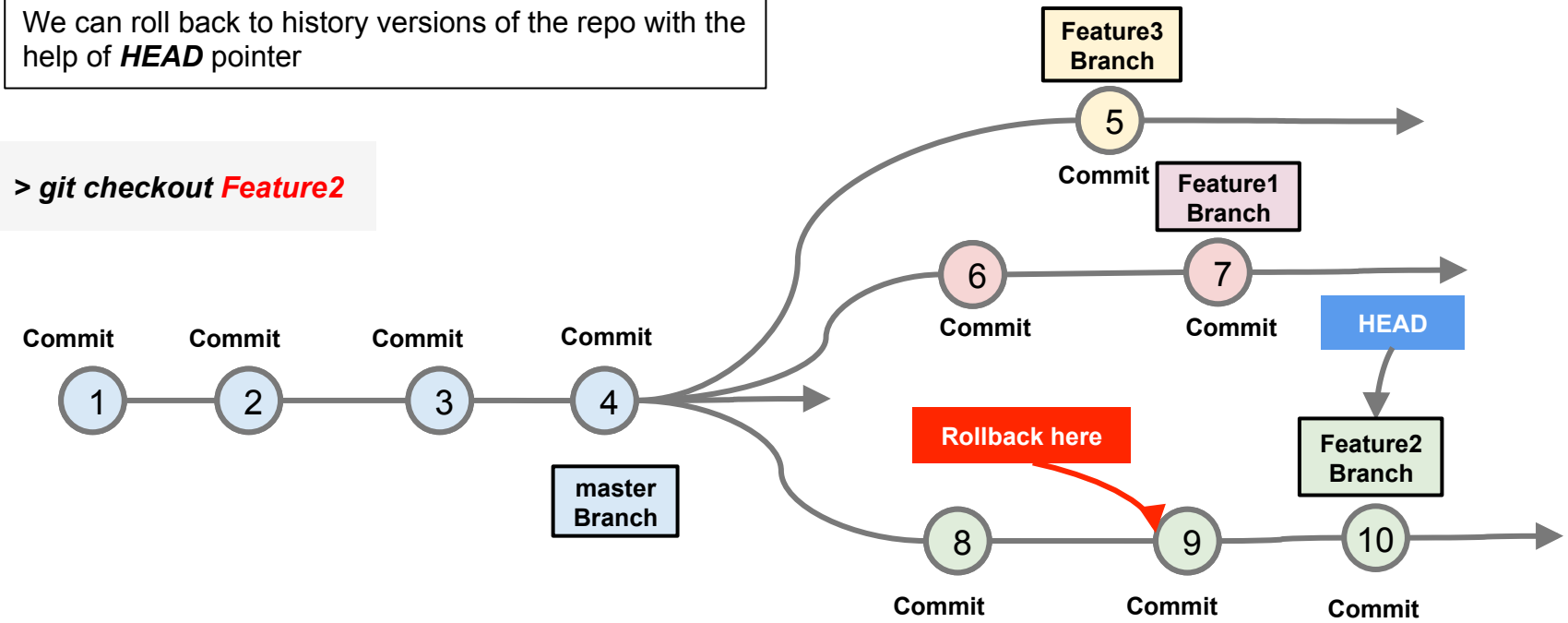
> git checkout Feature2



Branching – roll back to history version

We can roll back to history versions of the repo with the help of ***HEAD*** pointer

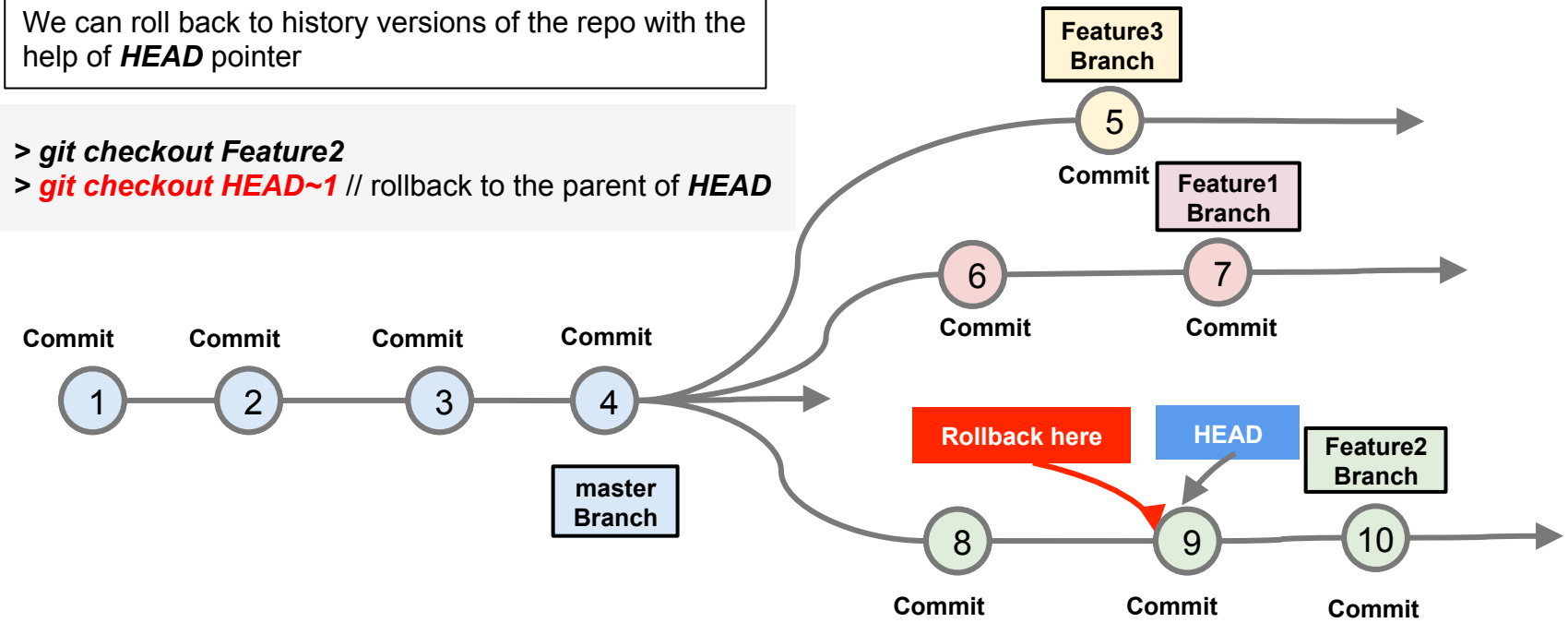
> git checkout Feature2



Branching – roll back to history version

We can roll back to history versions of the repo with the help of ***HEAD*** pointer

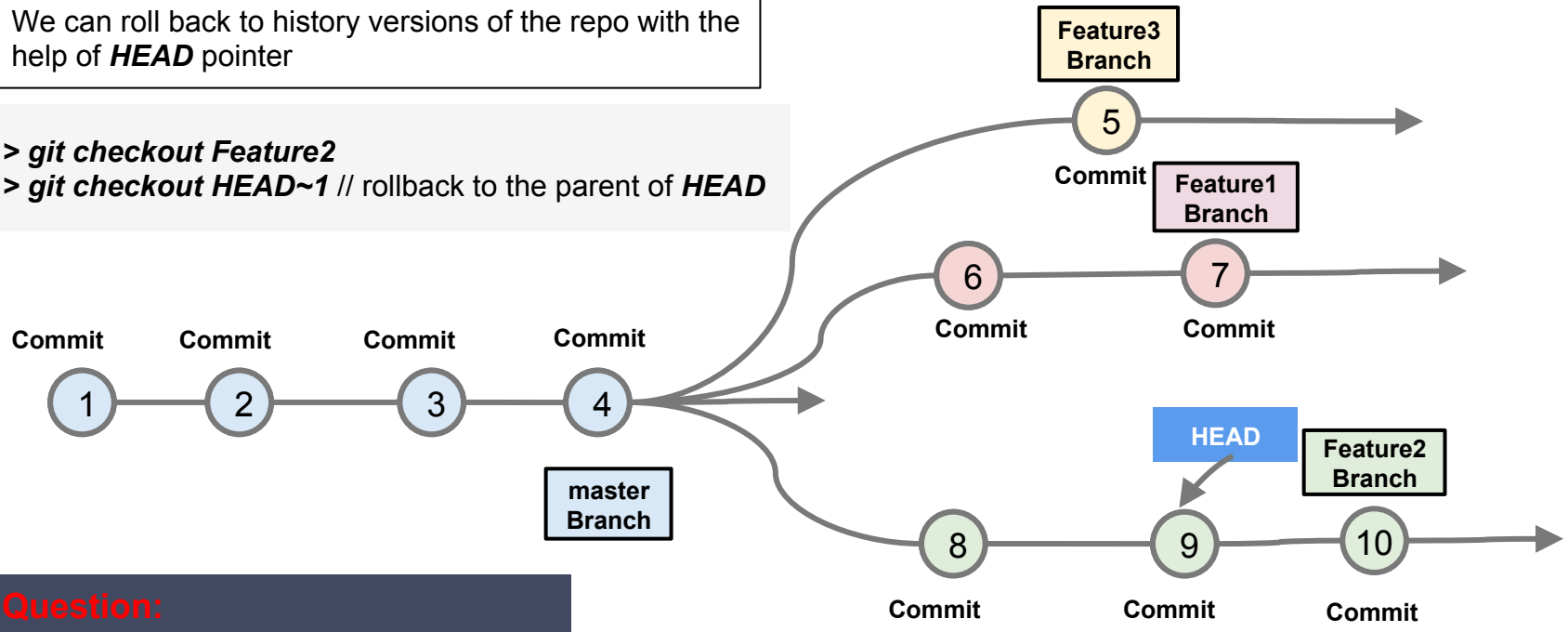
> git checkout Feature2
> git checkout HEAD~1 // rollback to the parent of ***HEAD***



Branching – roll back to history version

We can roll back to history versions of the repo with the help of ***HEAD*** pointer

> git checkout Feature2
> git checkout HEAD~1 // rollback to the parent of ***HEAD***

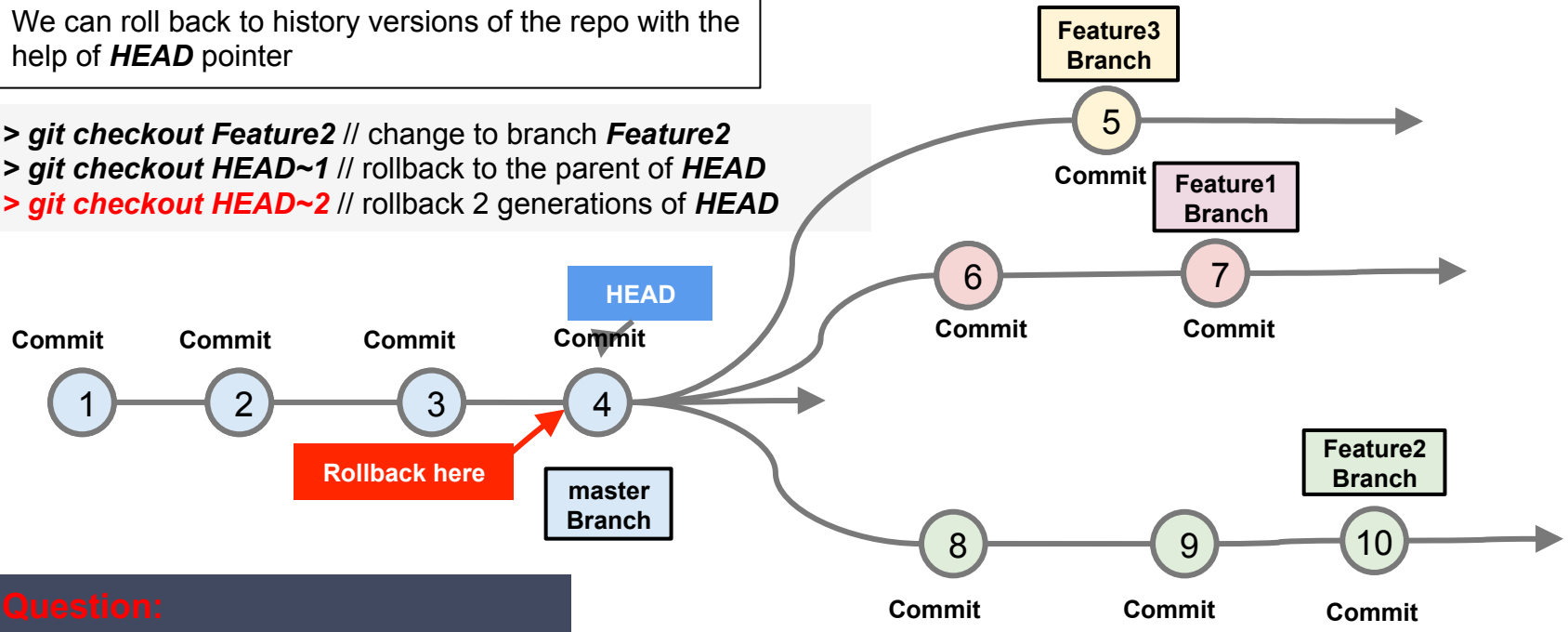


Question:
> git checkout HEAD~2
Where is the HEAD pointer now?

Branching – roll back to history version

We can roll back to history versions of the repo with the help of ***HEAD*** pointer

> ***git checkout Feature2*** // change to branch ***Feature2***
> ***git checkout HEAD~1*** // rollback to the parent of ***HEAD***
> ***git checkout HEAD~2*** // rollback 2 generations of ***HEAD***



Question:
> git checkout HEAD~2
Where is the HEAD pointer now?

Version Control Systems: Outline

- Types of Version Control Systems
- Git
  - Branching
  - Merging
  - Rebasing
  - Squashing
  - Cherry-pick
  - Remote repositories
- GitHub
  - Cloning vs. Forking
  - Making pull request
- Branching strategies

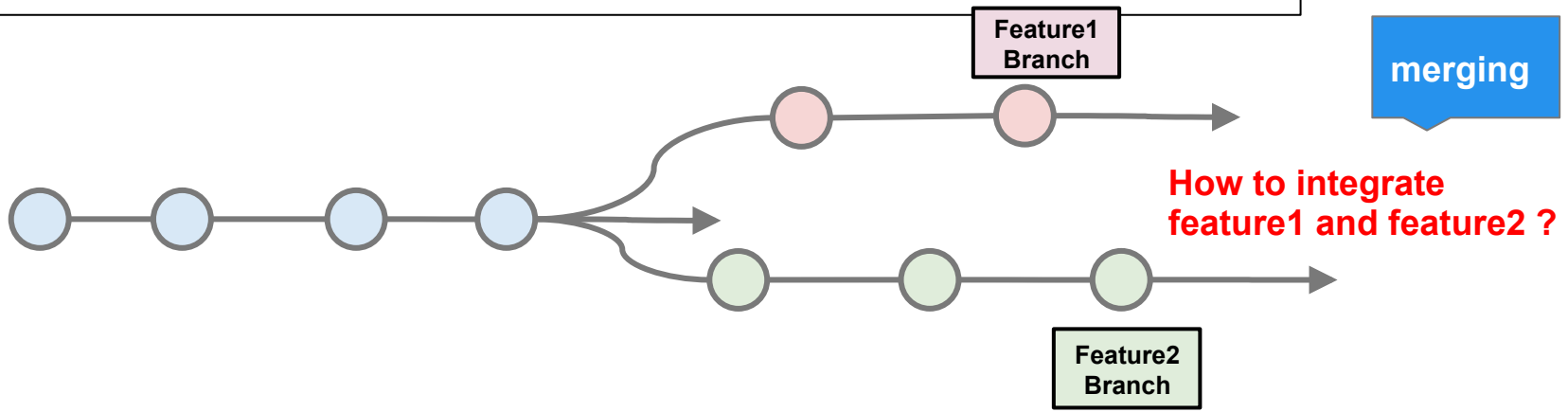
Collaborative development needs merging

Scenario2: Suppose we have two features to implement for one system:

- Feature1
- Feature2

Feature1 and feature2 are totally isolated, how can development be efficient?

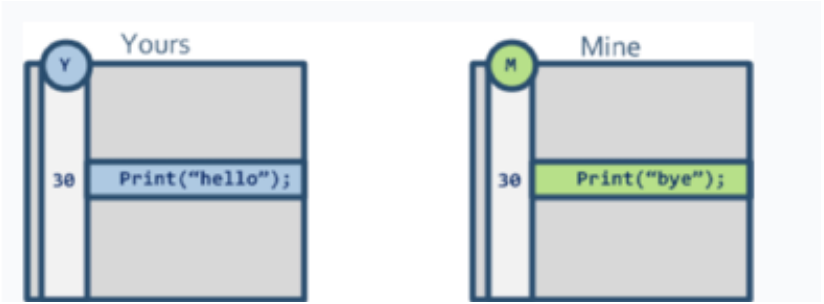
Better option: create one branch for each feature, work in parallel



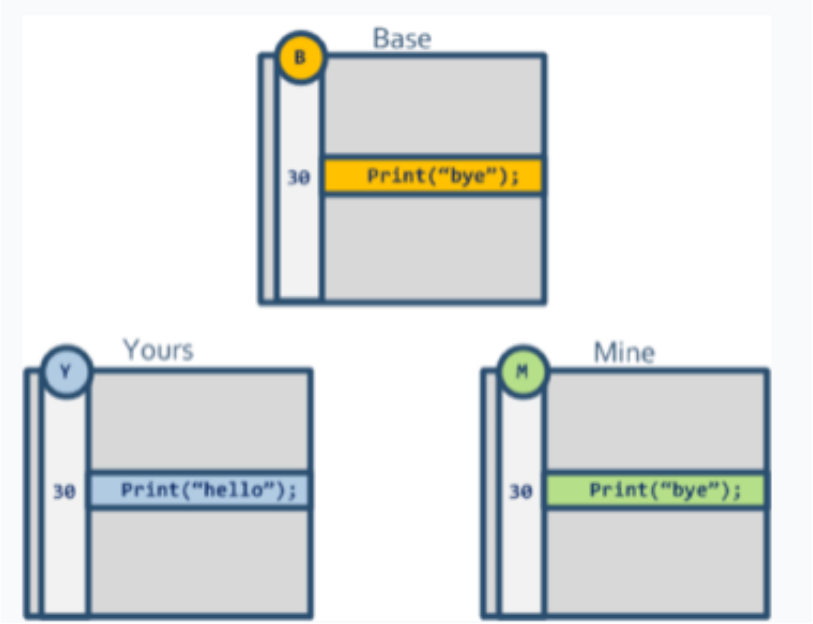
Two-way merging vs. Three-way merging

Two-way merging:

Cannot tell whether you modified line 30 or if I modified it or if we both modified it



Three-way merging:

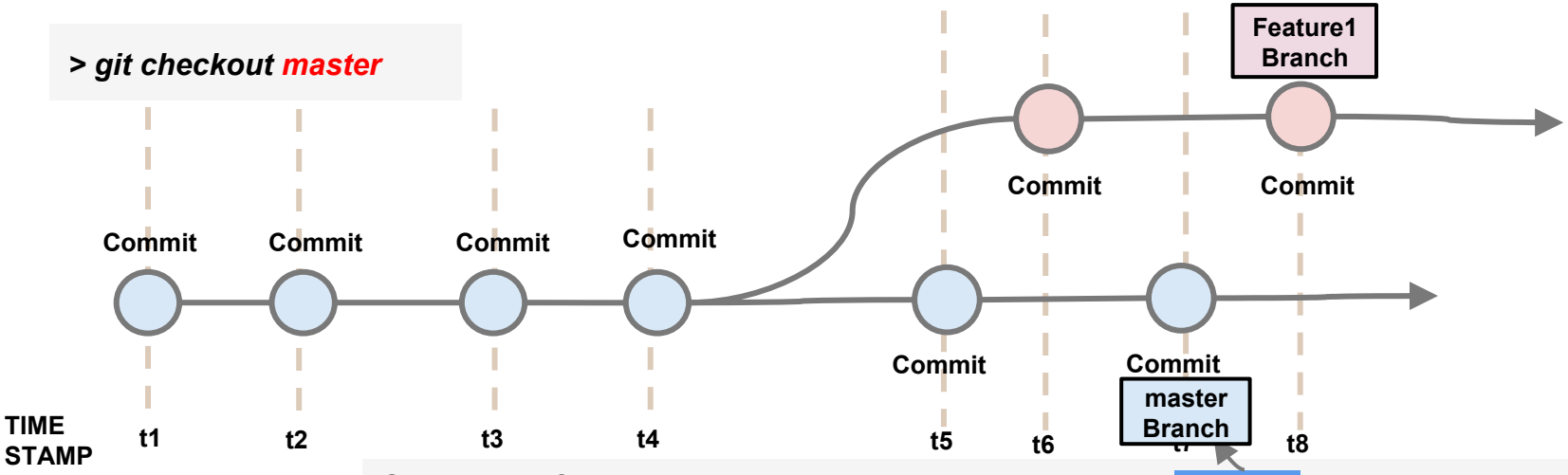


http://www.drdobbs.com/tools/three-way-merging-a-look-under-the-hood/240164902

Merging – git use three-way merging

Merging steps:

- Merge divergence
- move the ***branch*** pointer, create a merge commit



Commands for *Merge*

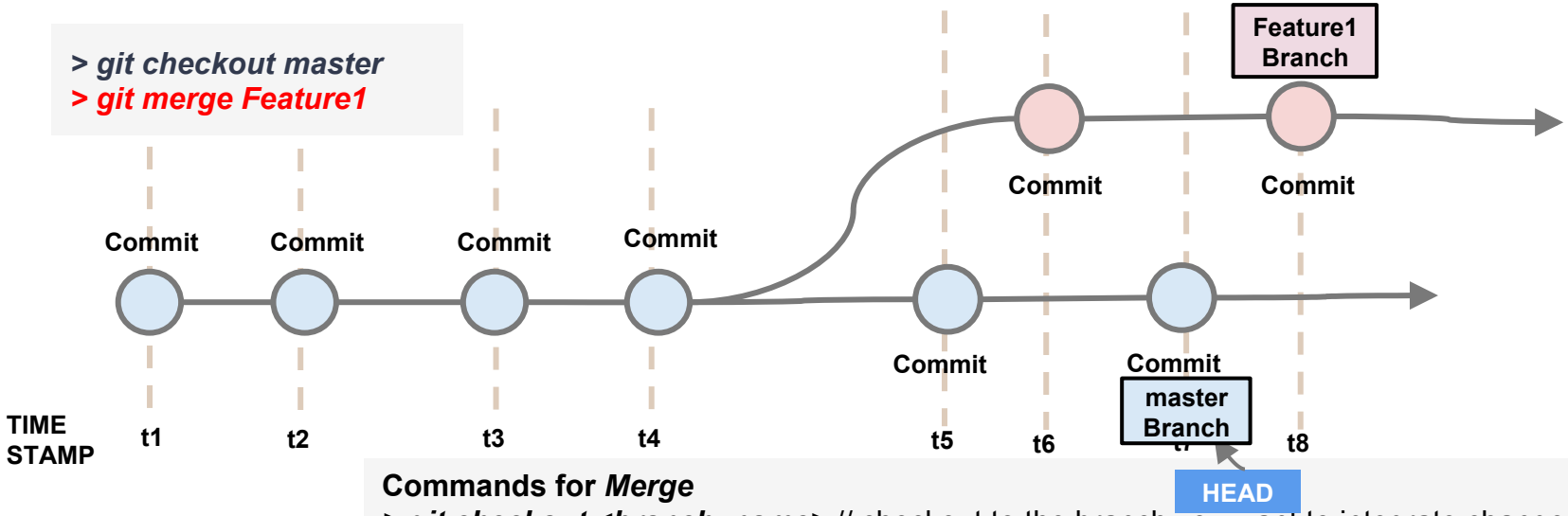
> git checkout <branch_name> // checkout to the branch you want to integrate changes to

> git merge <branch_to_merge> // merge current branch with <branch_to_merge>

Merging – git use three-way merging

Merging steps:

- Merge divergence
- move the ***branch*** pointer, create a merge commit



Commands for *Merge*

> git checkout <branch_name> // checkout to the branch you want to integrate changes to

> git merge <branch_to_merge> // merge current branch with <branch_to_merge>

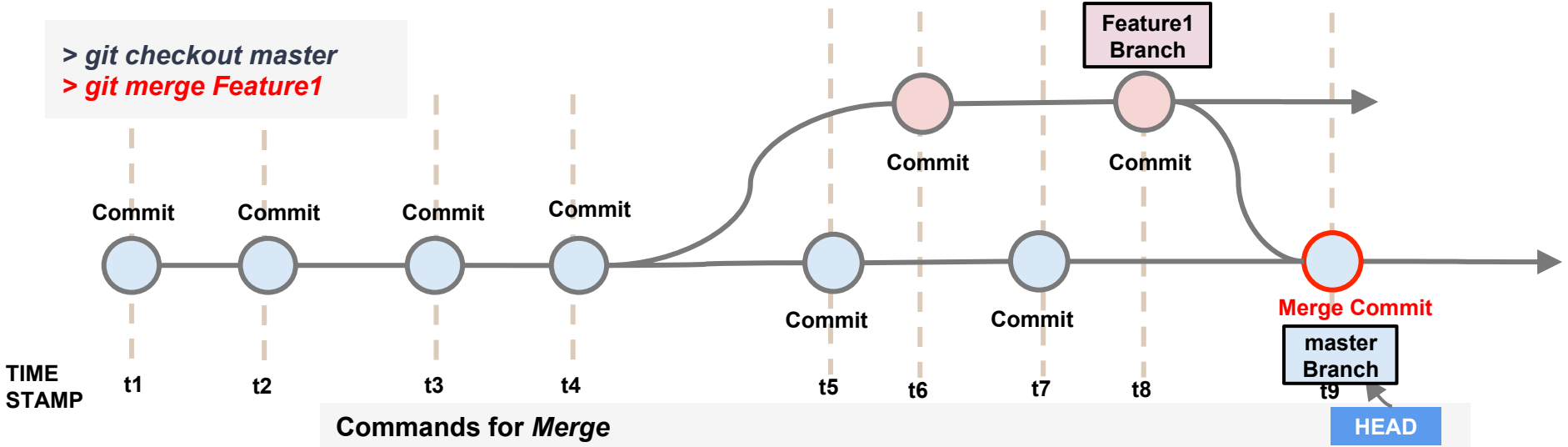
Merging – git use three-way merging

Merging steps:

- Merge divergence
- move the ***branch*** pointer, create a merge commit

Create a merge commit

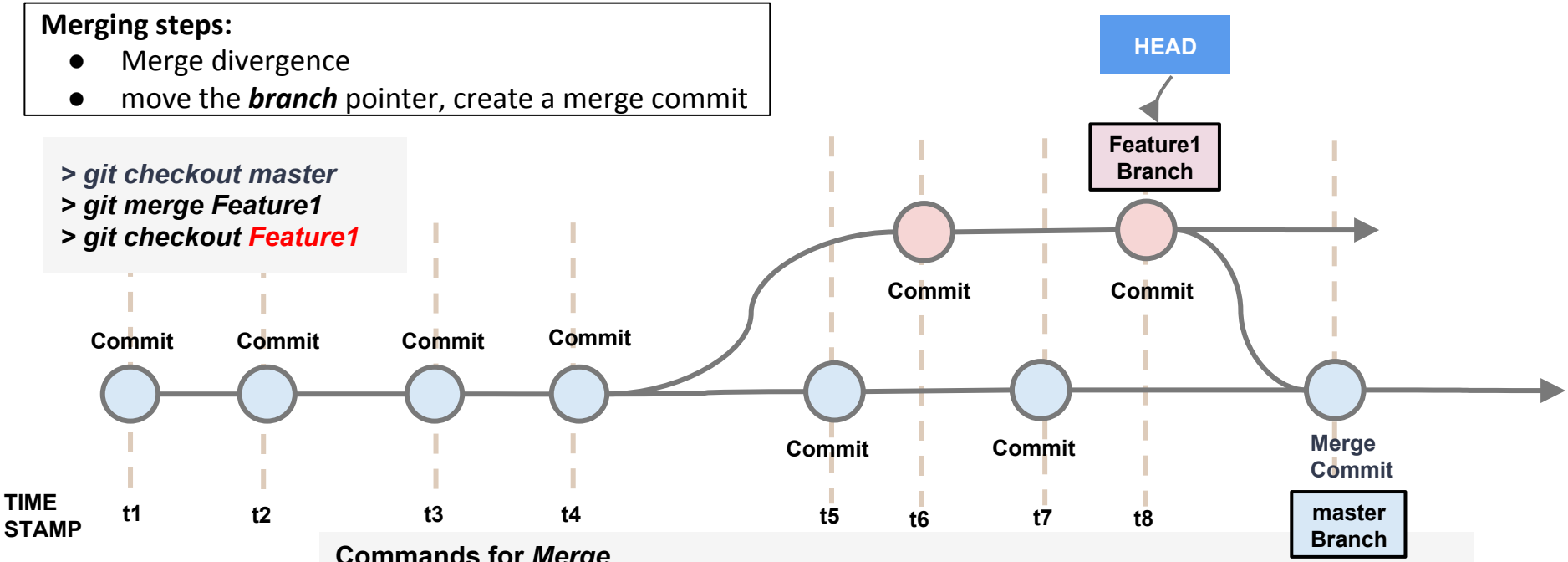
> git checkout master
> git merge Feature1



Commands for *Merge*
> git checkout <branch_name> // checkout to the branch you want to integrate changes to
> git merge <branch_to_merge> // merge current branch with <branch_to_merge>

Merging – git use three-way merging

Merging steps:
- Merge divergence
- move the ***branch*** pointer, create a merge commit



Commands for *Merge*

> git checkout <branch_name> // checkout to the branch you want to integrate changes to

> git merge <branch_to_merge> // merge current branch with <branch_to_merge>

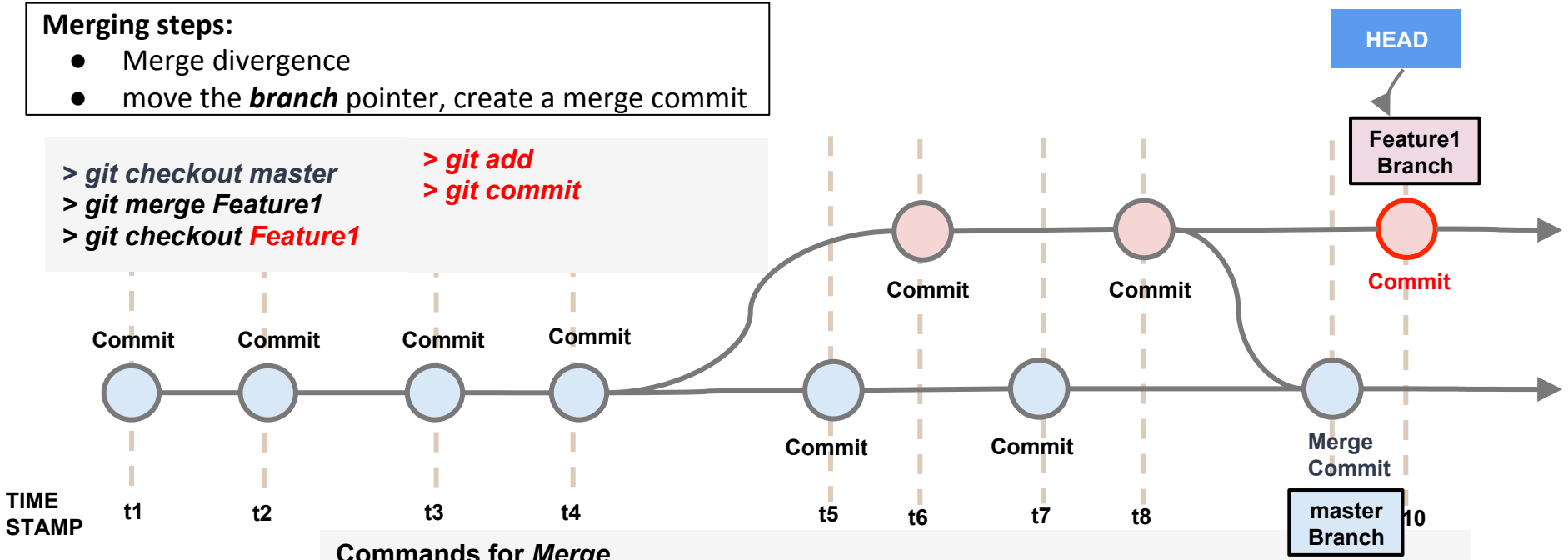
Merging – git use three-way merging

Merging steps:

- Merge divergence
- move the ***branch*** pointer, create a merge commit

> git checkout master
> git merge Feature1
> git checkout Feature1

> git add
> git commit



Commands for *Merge*

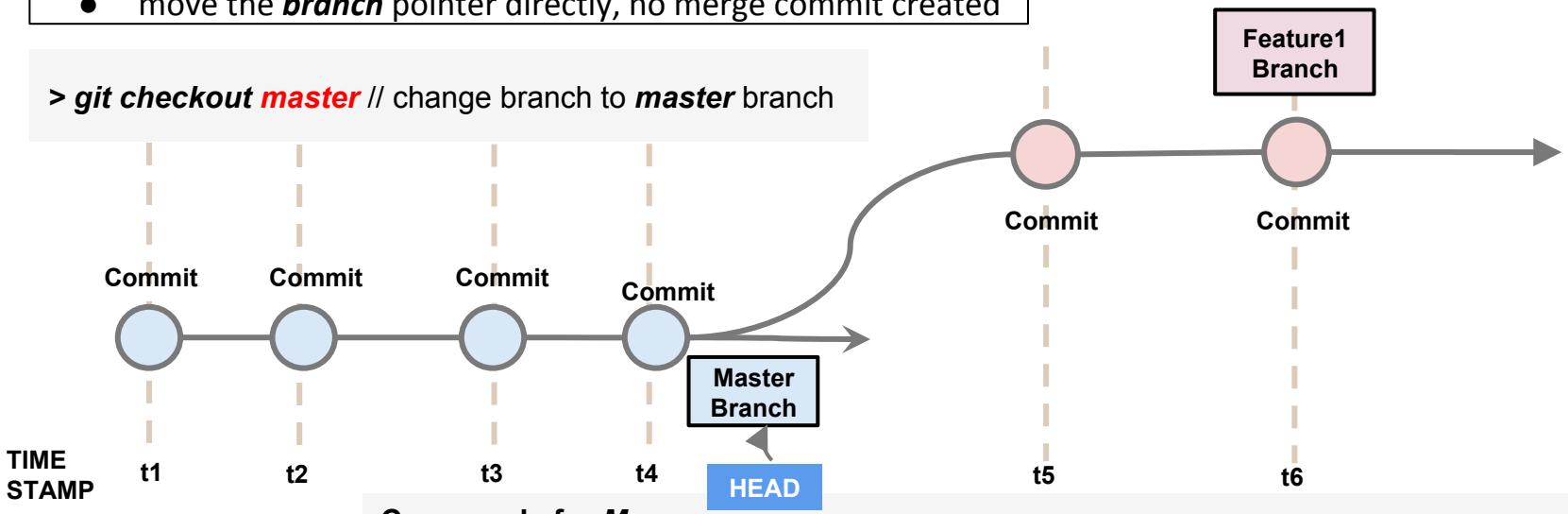
> git checkout <branch_name> // checkout to the branch you want to integrate changes to
> git merge <branch_to_merge> // merge current branch with <branch_to_merge>

Merging - fast-forward merging

Fast-forward merging steps:

- merge sequential commits
- move the ***branch*** pointer directly, no merge commit created

> ***git checkout master*** // change branch to ***master*** branch



Commands for *Merge*

> git checkout <branch_name> // checkout to the branch you want to integrate changes to

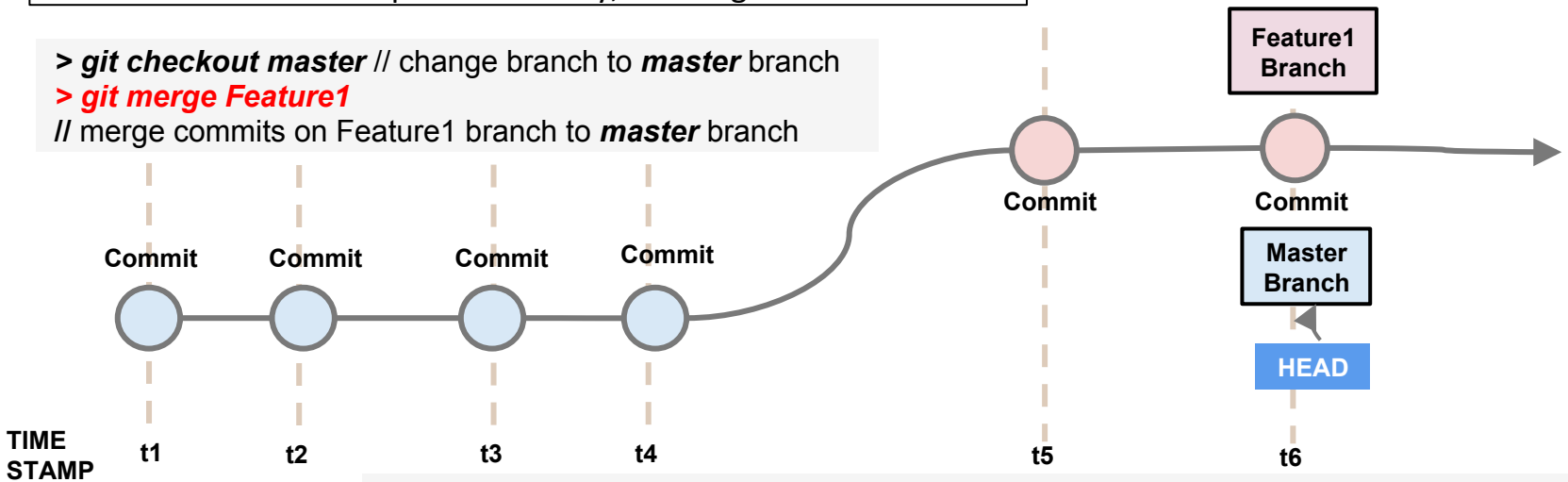
> git merge <branch_to_merge> // merge current branch with <branch_to_merge>

Merging - fast-forward merging

Fast-forward merging steps:

- merge sequential commits
- move the ***branch*** pointer directly, no merge commit created

> git checkout master // change branch to ***master*** branch
> git merge Feature1
// merge commits on Feature1 branch to ***master*** branch



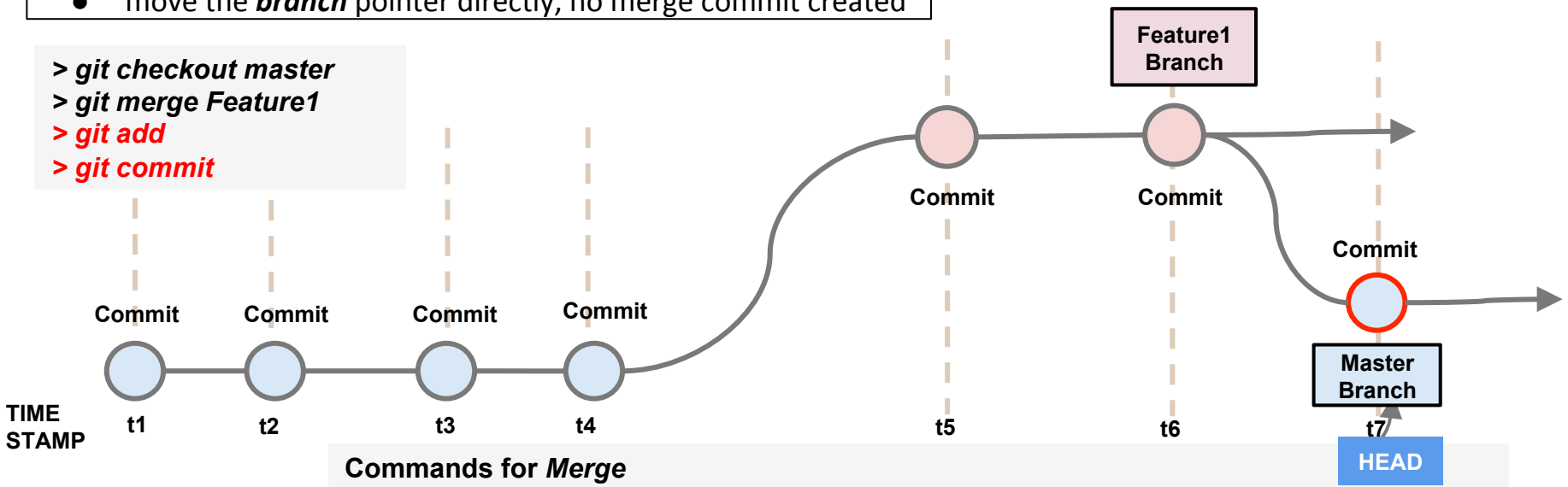
Commands for *Merge*
> git checkout <branch_name> // checkout to the branch you want to integrate changes to
> git merge <branch_to_merge> // merge current branch with <branch_to_merge>

Merging - fast-forward merging

Fast-forward merging steps:

- merge sequential commits
- move the ***branch*** pointer directly, no merge commit created

> git checkout master
> git merge Feature1
> git add
> git commit



Commands for *Merge*
> git checkout <branch_name> // checkout to the branch you want to integrate changes to
> git merge <branch_to_merge> // merge current branch with <branch_to_merge>

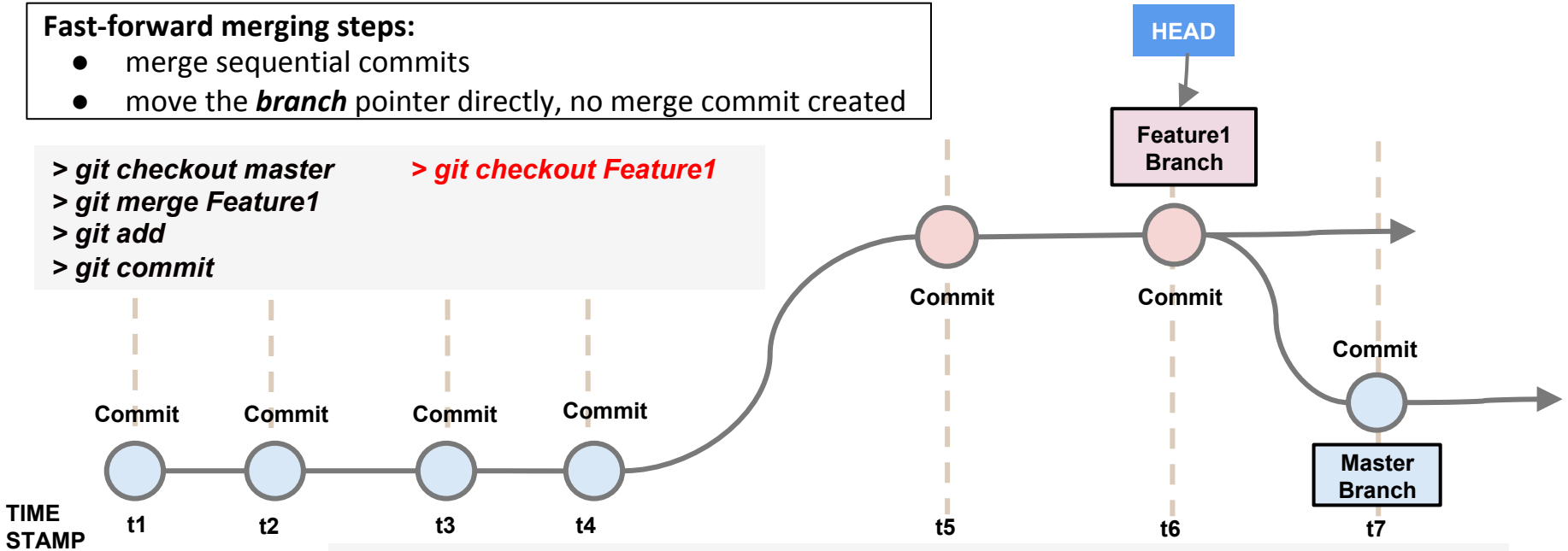
Merging - fast-forward merging

Fast-forward merging steps:

- merge sequential commits
- move the ***branch*** pointer directly, no merge commit created

> git checkout master
> git merge Feature1
> git add
> git commit

> git checkout Feature1



Commands for *Merge*

> git checkout <branch_name> // checkout to the branch you want to integrate changes to
> git merge <branch_to_merge> // merge current branch with <branch_to_merge>

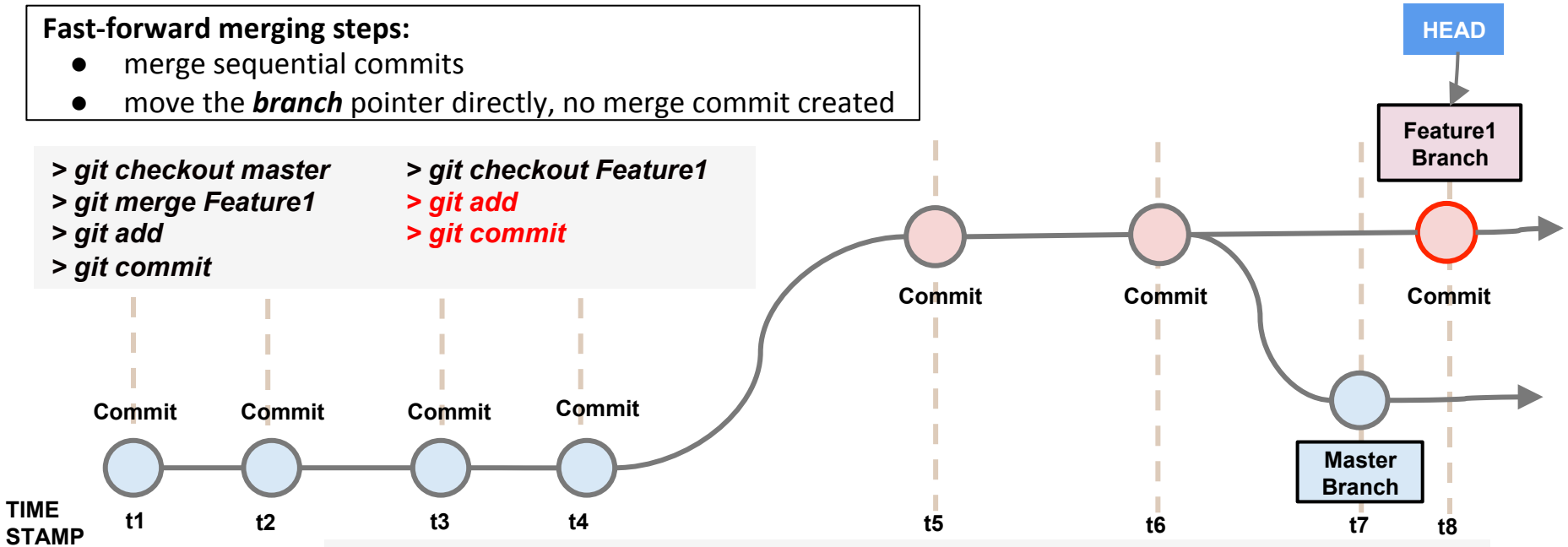
Merging - fast-forward merging

Fast-forward merging steps:

- merge sequential commits
- move the ***branch*** pointer directly, no merge commit created

```
> git checkout master
> git merge Feature1
> git add
> git commit
```

```
> git checkout Feature1
> git add
> git commit
```



Commands for *Merge*

> git checkout <branch_name> // checkout to the branch you want to integrate changes to

> git merge <branch_to_merge> // merge current branch with <branch_to_merge>

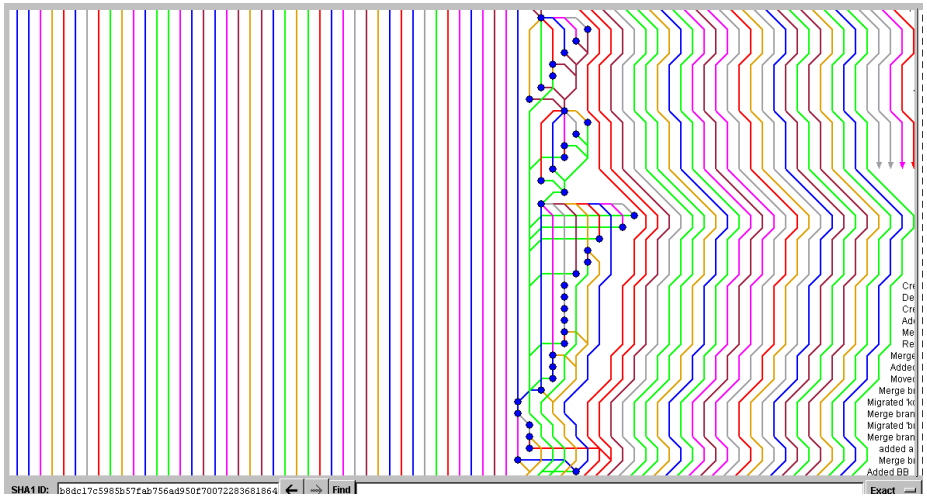
Merging

If used properly, merging is helpful:

- Is a non-destructive operation
- Keeps information in merge commit

If used improperly, merging could cause issues:

- Creates large amount of extraneous merge commits (for three-way merging)
- Might cause the project histories to be messy and less readable

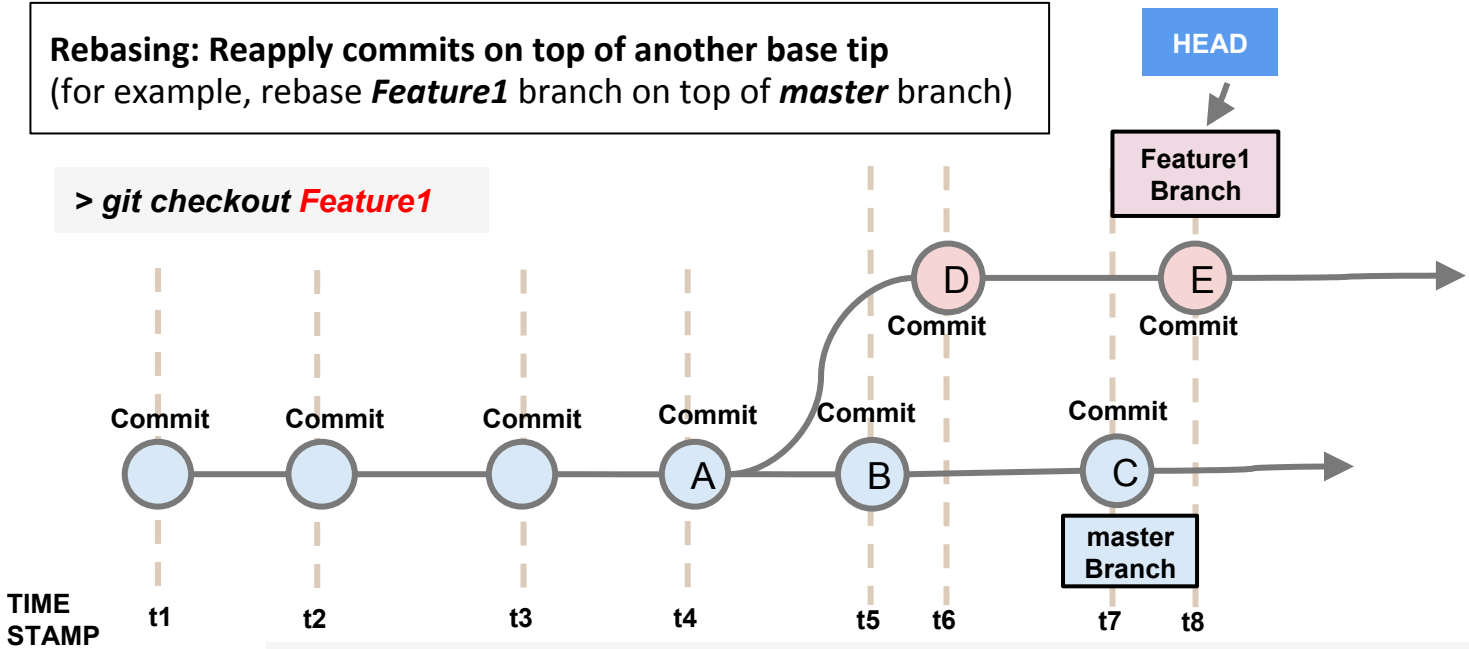


Version Control Systems: Outline

- Types of Version Control Systems
- Git
  - Branching
  - Merging
  - Rebasing
  - Squashing
  - Cherry-pick
  - Remote repositories
- GitHub
  - Cloning vs. Forking
  - Making pull request
- Branching strategies

Rebasing – before rebasing

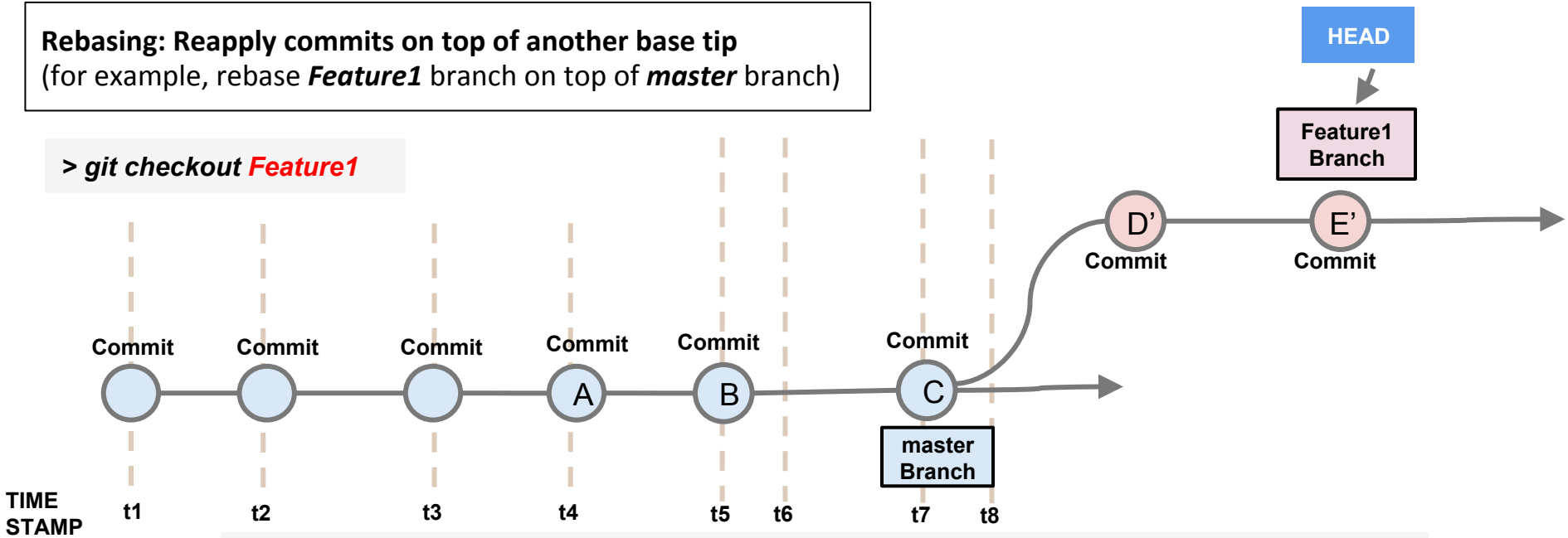
Rebasing: Reapply commits on top of another base tip
(for example, rebase ***Feature1*** branch on top of ***master*** branch)



Commands for *Rebase*
> git checkout <branch_name> // checkout to the branch you want to rebase
> git rebase <branch_rebase_to> // rebase current branch to the last commit of the given branch

Rebasing – after rebasing

Rebasing: Reapply commits on top of another base tip
(for example, rebase ***Feature1*** branch on top of ***master*** branch)



Commands for *Rebase*
> git checkout <branch_name> // checkout to the branch you want to rebase
> git rebase <branch_rebase_to> // rebase current branch to the last commit of the given branch

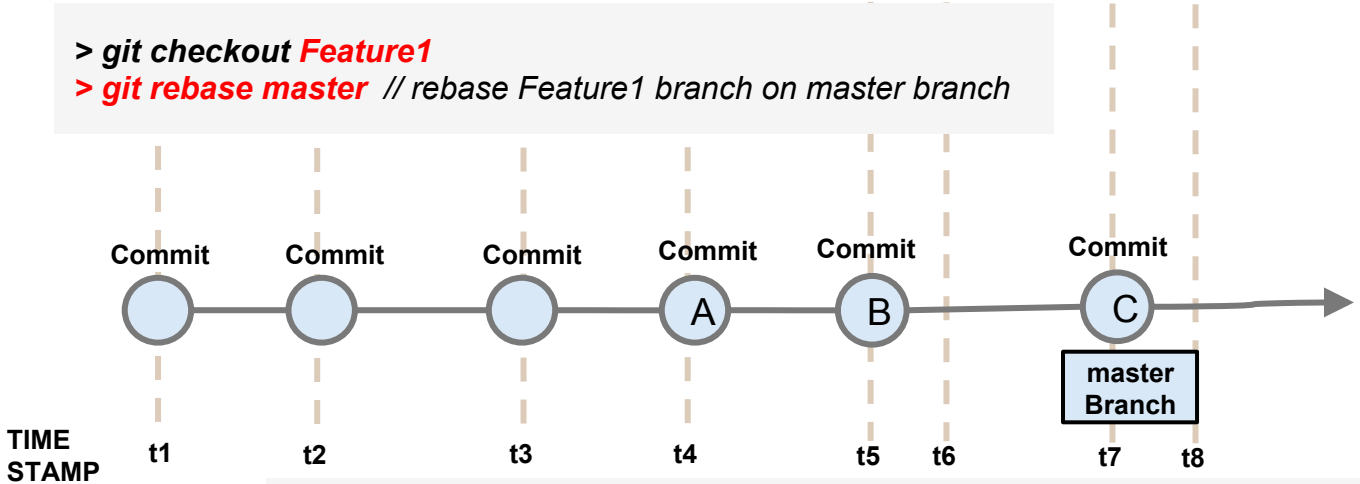
Rebasing – step1

Rebasing: Reapply commits on top of another base tip
(for example, rebase ***Feature1*** branch on top of ***master*** branch)

TEMP AREA

D Commit E Commit

> git checkout Feature1
> git rebase master *// rebase Feature1 branch on master branch*



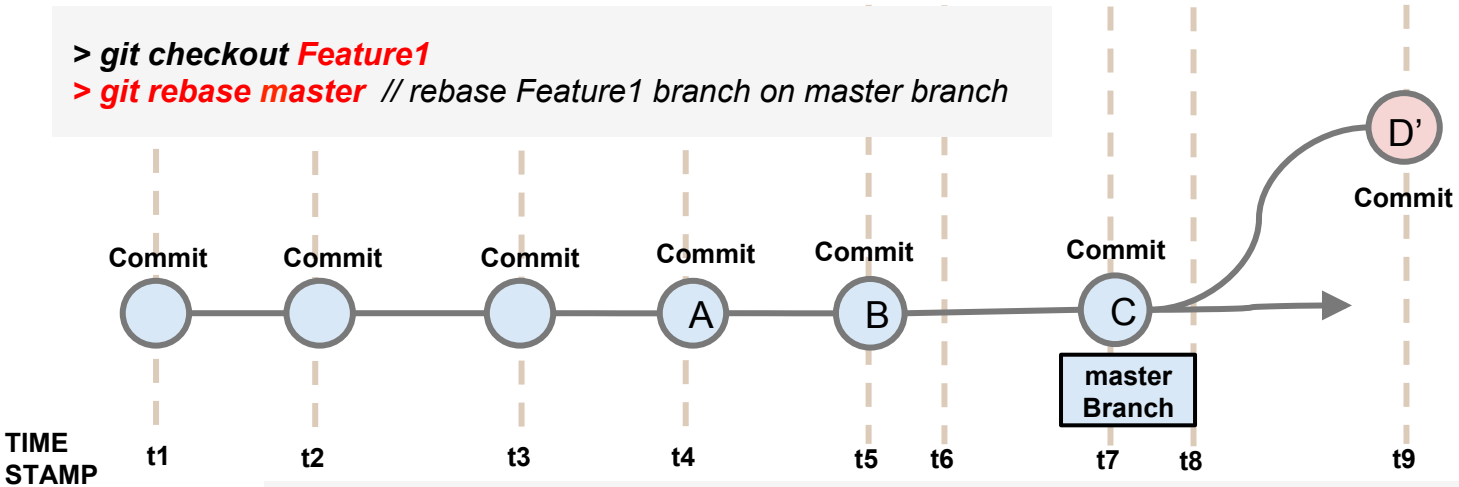
Commands for *Rebase*
> git checkout <branch_name> // checkout to the branch you want to rebase
> git rebase <branch_rebase_to> // rebase current branch to the last commit of the given branch

Rebasing – step2

Rebasing: Reapply commits on top of another base tip
(for example, rebase ***Feature1*** branch on top of ***master*** branch)

TEMP AREA

> git checkout Feature1
> git rebase master *// rebase Feature1 branch on master branch*

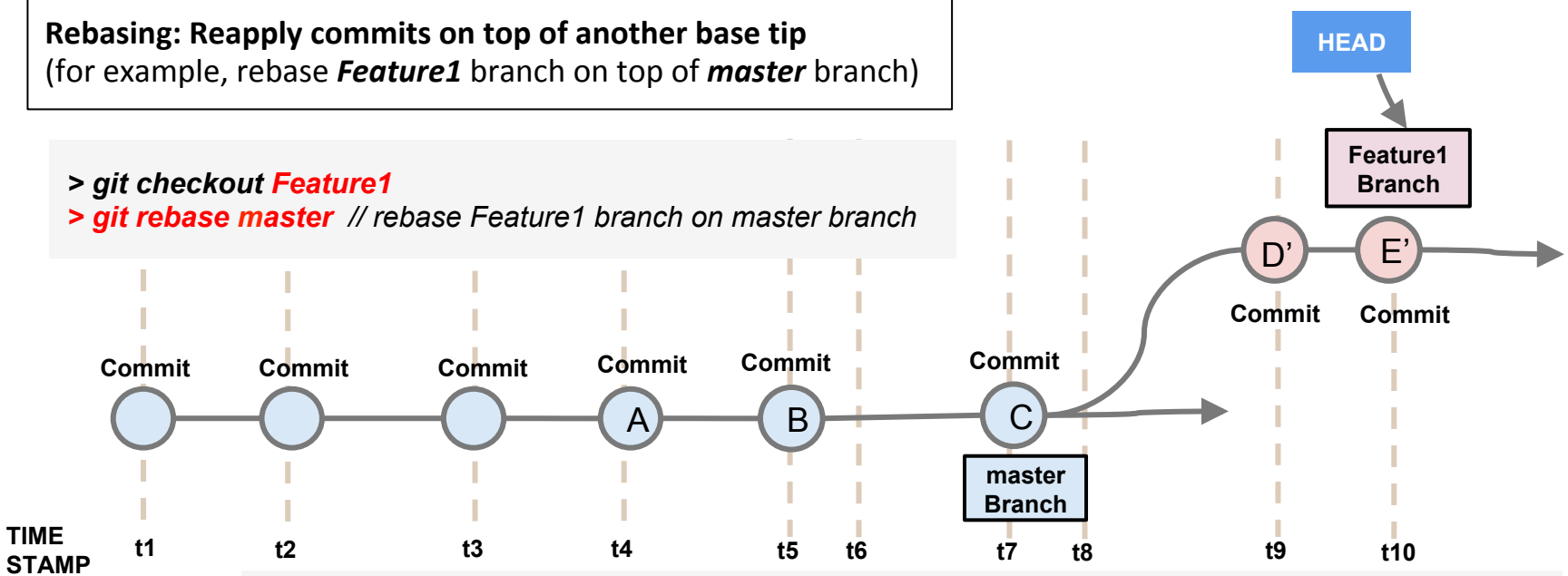


Commands for *Rebase*
> git checkout <branch_name> // checkout to the branch you want to rebase
> git rebase <branch_rebase_to> // rebase current branch to the last commit of the given branch

Rebasing result

Rebasing: Reapply commits on top of another base tip
(for example, rebase ***Feature1*** branch on top of ***master*** branch)

> git checkout Feature1
> git rebase master *// rebase Feature1 branch on master branch*

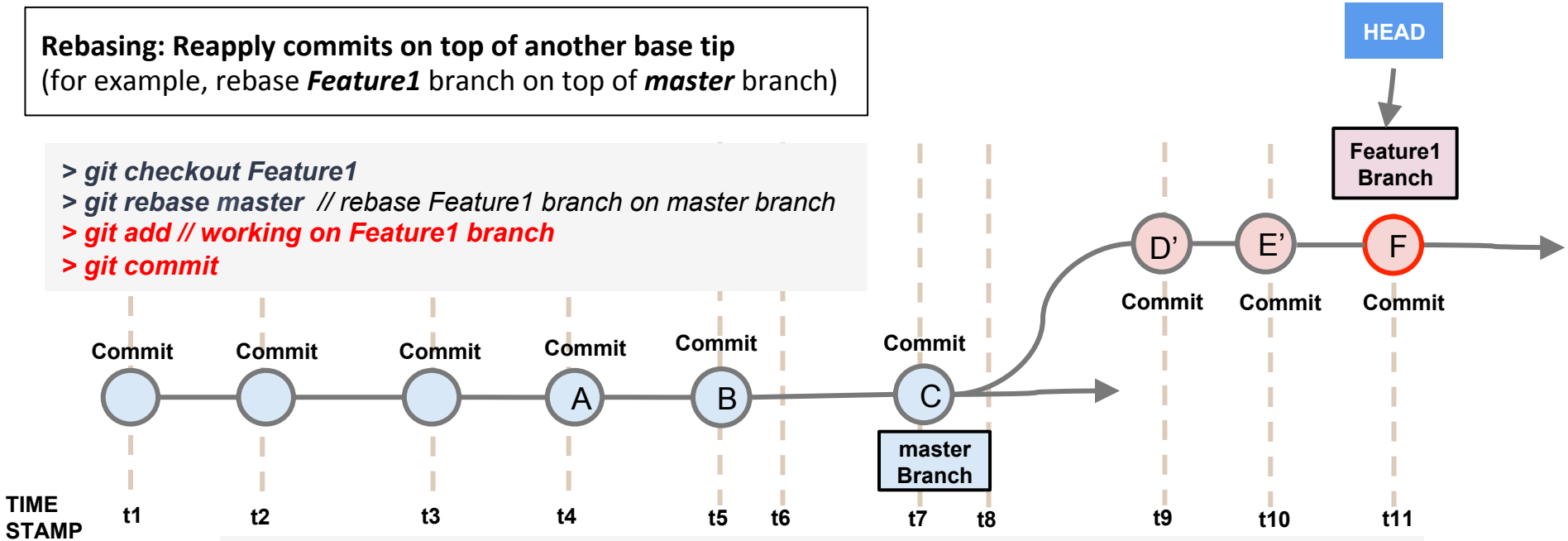


Commands for *Rebase*
> git checkout <branch_name> // checkout to the branch you want to rebase
> git rebase <branch_rebase_to> // rebase current branch to the last commit of the given branch

Rebasing result

Rebasing: Reapply commits on top of another base tip
(for example, rebase ***Feature1*** branch on top of ***master*** branch)

```
> git checkout Feature1
> git rebase master  // rebase Feature1 branch on master branch
> git add // working on Feature1 branch
> git commit
```



Commands for *Rebase*

> git checkout <branch_name> // checkout to the branch you want to rebase

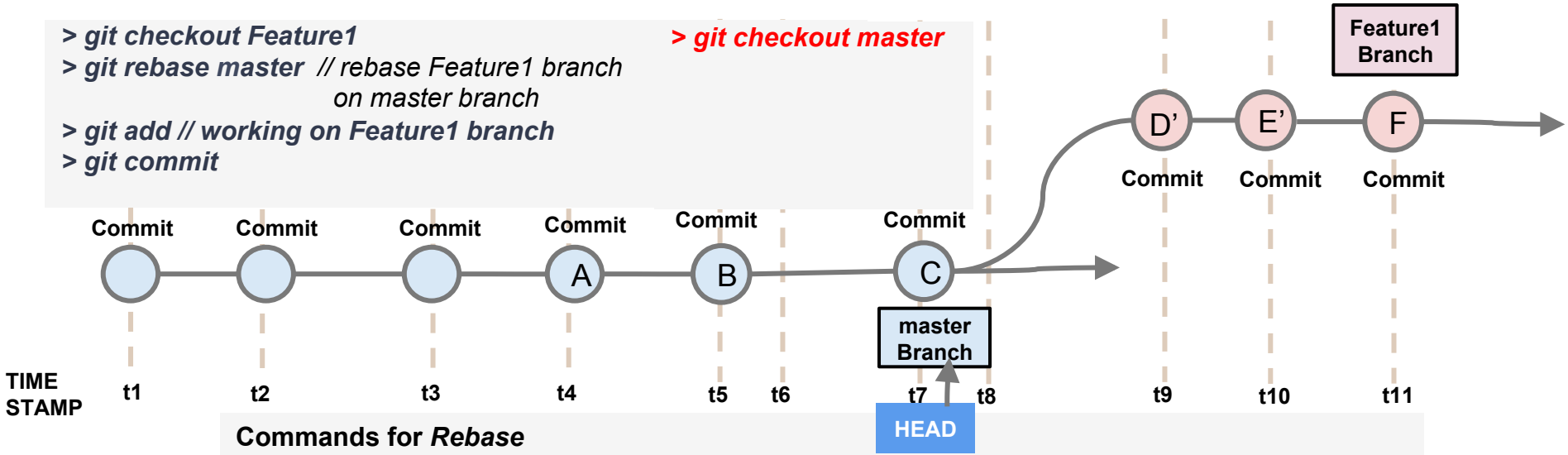
> git rebase <branch_rebase_to> // rebase current branch to the last commit of the given branch

Rebasing result

Rebasing: Reapply commits on top of another base tip
(for example, rebase ***Feature1*** branch on top of ***master*** branch)

> git checkout Feature1
> git rebase master *// rebase Feature1 branch on master branch*
> git add // working on Feature1 branch
> git commit

> git checkout master



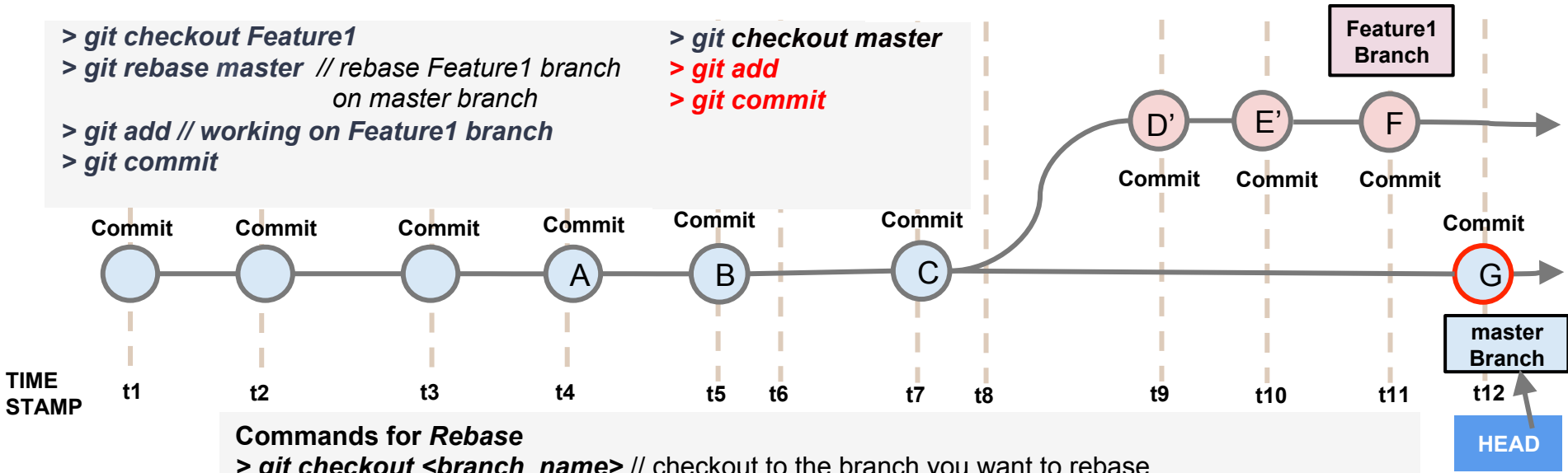
Commands for *Rebase*
> git checkout <branch_name> // checkout to the branch you want to rebase
> git rebase <branch_rebase_to> // rebase current branch to the last commit of the given branch

Rebasing result

Rebasing: Reapply commits on top of another base tip
(for example, rebase ***Feature1*** branch on top of ***master*** branch)

```
> git checkout Feature1
> git rebase master  // rebase Feature1 branch
                        on master branch
> git add // working on Feature1 branch
> git commit
```

```
> git checkout master
> git add
> git commit
```



Commands for *Rebase*

> git checkout <branch_name> // checkout to the branch you want to rebase

> git rebase <branch_rebase_to> // rebase current branch to the last commit of the given branch

Rebasing

Pros:

- Keep a clean project history
  - Linear
  - No “merge commit”

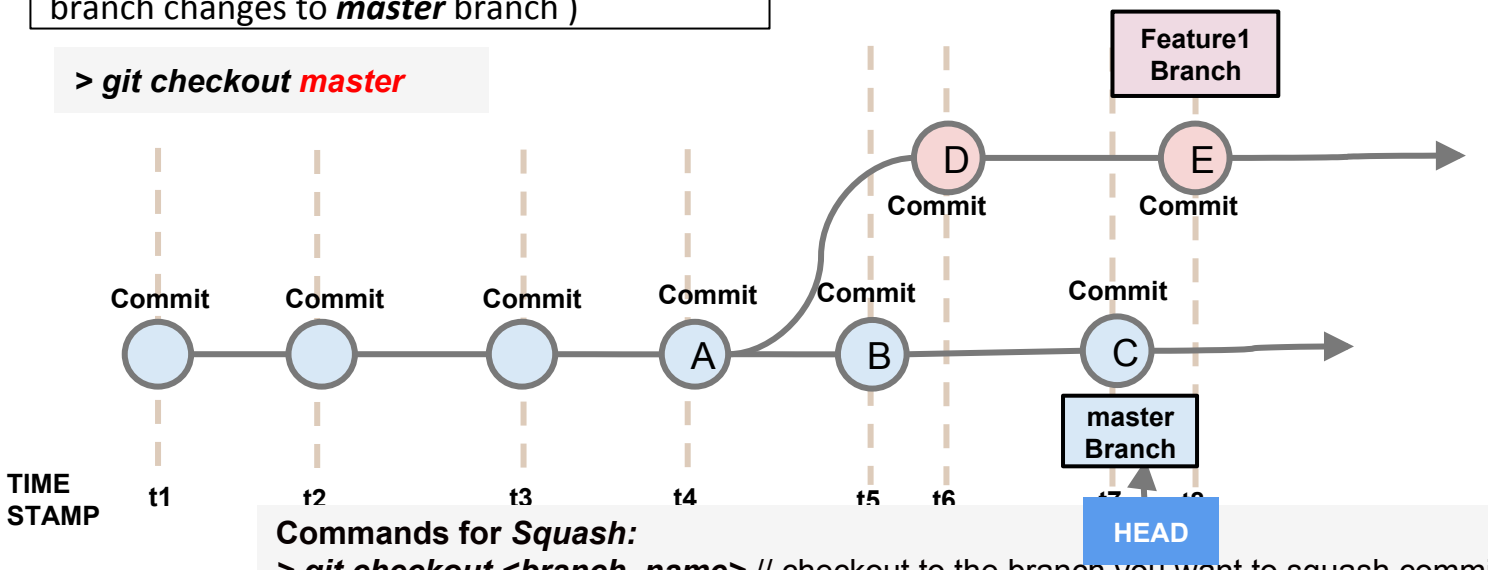
Cons:

- Re-write project histories, can be potentially **catastrophic** for your collaboration workflow
- Lose information that developers attached in the merge commits

Squashing – make history even more clean

Squashing*:** meld a series of commits down into a single commit (e.g., integrate ***Feature1 branch changes to ***master*** branch)

> git checkout master



Commands for *Squash:*

> git checkout <branch_name> // checkout to the branch you want to squash commits to

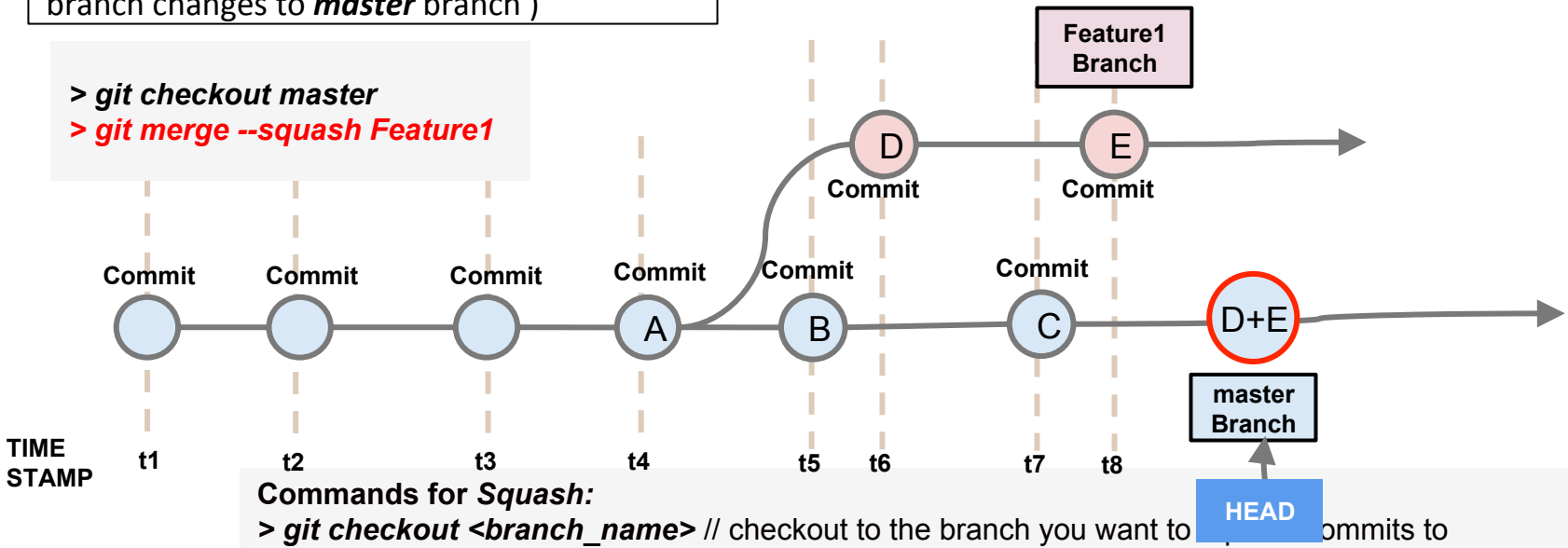
Way1: ***> git commit --squash <branch_to_squash>*** // squash the branch where new changes are

Way2: ***> git rebase <branch_to_squash> -i*** , then choose 's' option // using rebasing interactive mode

Squashing – make history even more clean

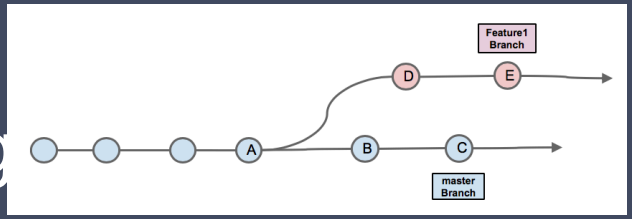
Squashing*:** meld a series of commits down into a single commit (e.g., integrate ***Feature1 branch changes to ***master*** branch)

> git checkout master
> git merge --squash Feature1

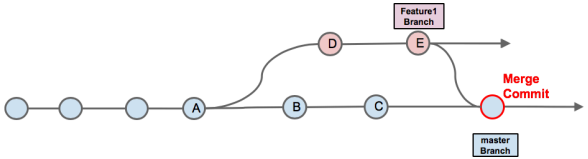


Commands for *Squash:*
> git checkout <branch_name> // checkout to the branch you want to [illegible]ommits to
Way1: ***> git commit --squash <branch_to_squash>*** // squash the branch where new changes are
Way2: ***> git rebase <branch_to_squash> -i*** , then choose 's' option // using rebasing interactive mode

Merging vs. Rebasing vs. Squashing

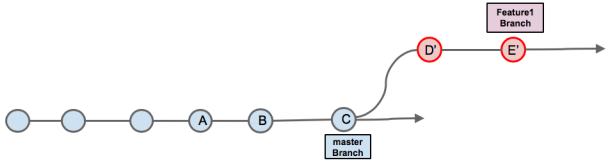


Merging:



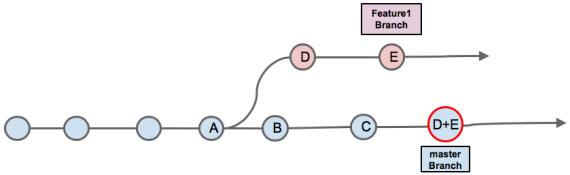
- Creates a new “merge commit” in the master branch that ties together the histories of both branches
- Is a *non-destructive* operation: the existing branches are not changed in any way

Rebasing:



- Moves the entire feature branch to begin on the tip of the master branch
- Re-write the history by creating brand new commits for each commit in the original branch

Squashing:



- Meld all changes on feature branch to one commit and apply on the tip of the master branch
- Keep history clean by creating a single commit containing all changes from the original branch

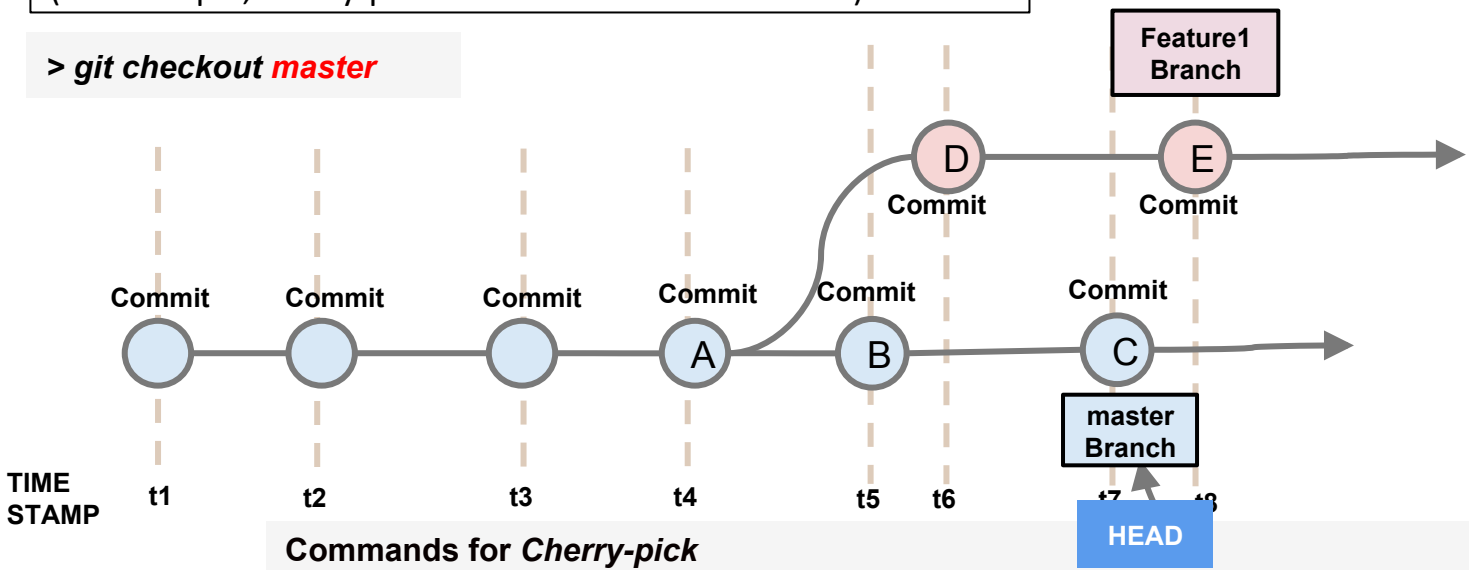
Version Control Systems: Outline

- Types of Version Control Systems
- Git
  - Branching
  - Merging
  - Rebasing
  - Squashing
  - Cherry-pick
  - Remote repositories
- GitHub
  - Cloning vs. Forking
  - Making pull request
- Branching strategies

Cherry-pick

Cherry-pick: Choose a commit from one branch and apply it to another by creating a new commit
(for example, cherry-pick commit ***E*** to ***master*** branch)

> git checkout master

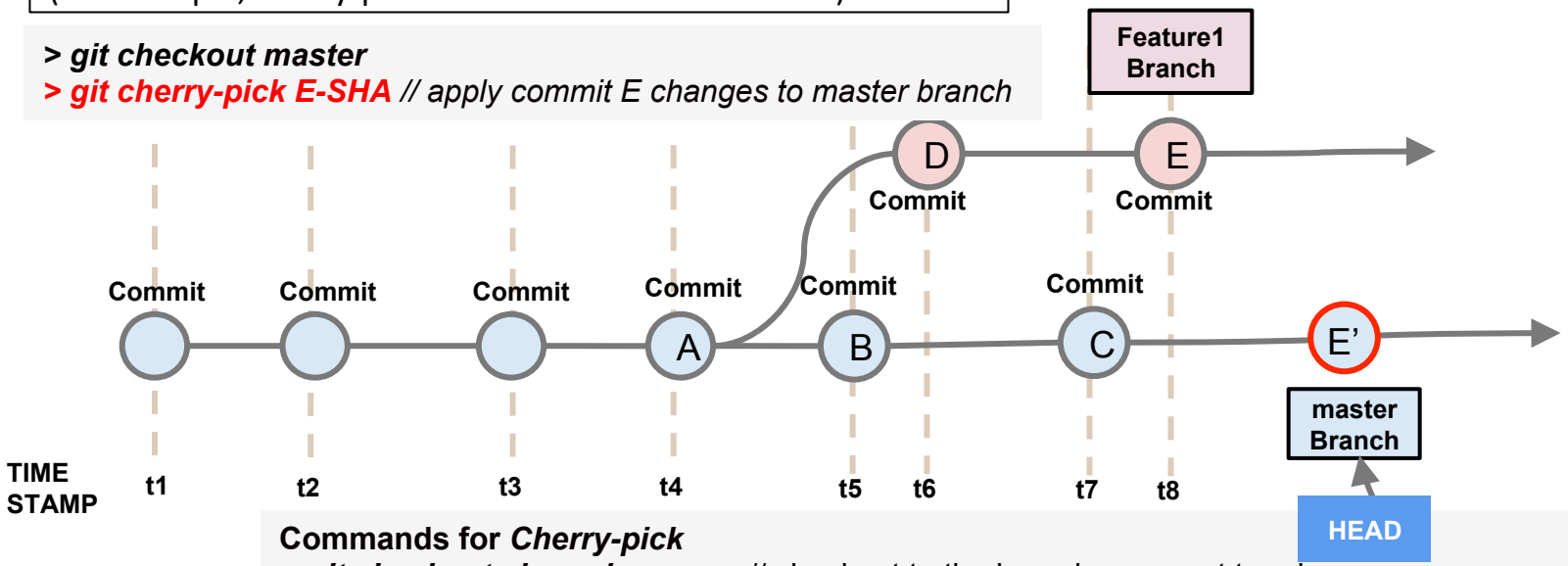


Commands for *Cherry-pick*
> git checkout <branch_name> // checkout to the branch you want to rebase
> git rebase <branch_rebase_to> // rebase current branch to the last commit of the given branch

Cherry-pick

Cherry-pick: Choose a commit from one branch and apply it to another by creating a new commit
(for example, cherry-pick commit ***E*** to ***master*** branch)

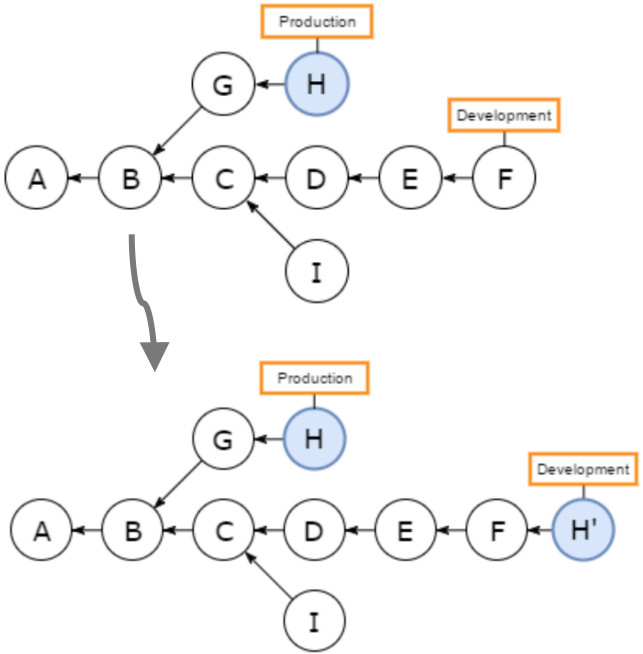
> git checkout master
> git cherry-pick E-SHA *// apply commit E changes to master branch*



Commands for *Cherry-pick*
> git checkout <branch_name> // checkout to the branch you want to rebase
> git rebase <branch_rebase_to> // rebase current branch to the last commit of the given branch

Why Cherry-pick?

- Useful when developers need a specific commit applied to some branches, but not commits prior to this one, e.g., apply the security patch to both Prod and Dev branch
- Cherry-pick creates a duplicate commit with the same changes and developers lose the ability to track the history of the original commit



https://www.farreachinc.com/blog/far-reach/2017/03/16/git-cherry-pick-why-and-how

Conflicts in integration (merging, rebasing, squashing, cherry-pick, etc.)

- Conflicts occur when
  - Two commits modified the same line in the same file, and Git doesn't know which change to apply
  - A file is deleted that another person is attempting to edit

- You must resolve such merge conflicts with a new commit before you can merge these branches / rebase/ squash / cherry-pick

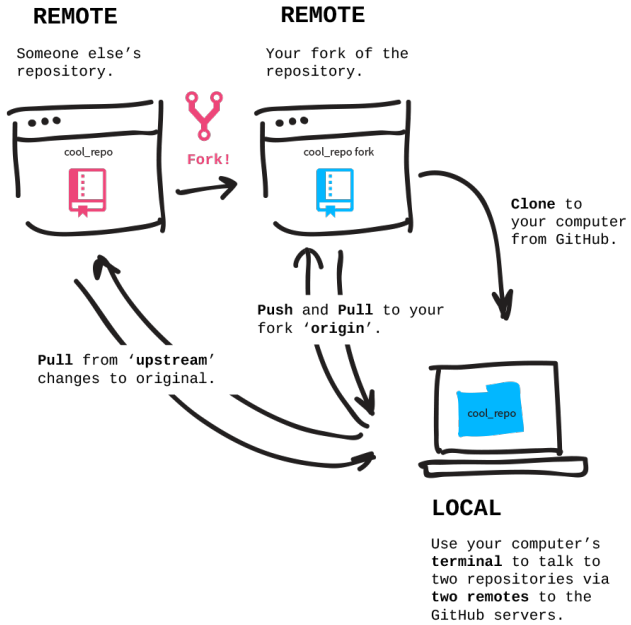
- Try to pull frequently to avoid merge conflicts

Version Control Systems: Outline

- Types of Version Control Systems
- Git
  - Branching
  - Merging
  - Rebasing
  - Squashing
  - Cherry-pick
  - Remote repositories
- GitHub
  - Cloning vs. Forking
  - Making pull request
- Branching strategies

Remote repositories

- Remote repositories are versions of your project that are hosted on the Internet or network somewhere, e.g., GitHub, your own laptop, etc.
- One repo can has multiple remote repositories
- Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them

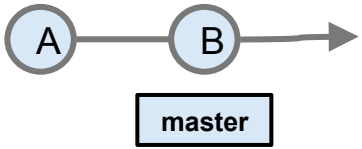


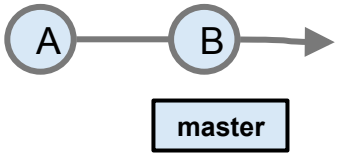
Remote repositories

Local repository:

Remote repository:

git push



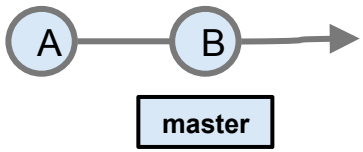


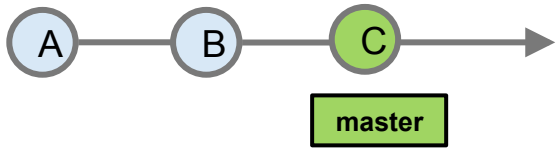
Remote repositories

Local repository:

Remote repository:

git push



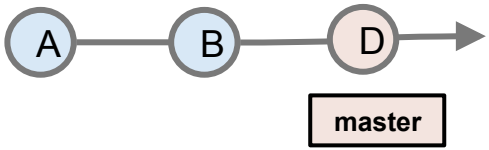


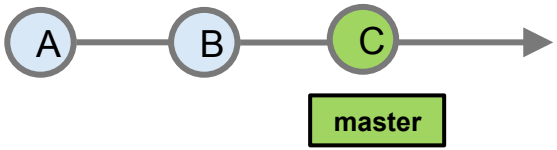
Remote repositories

Local repository:

Remote repository:

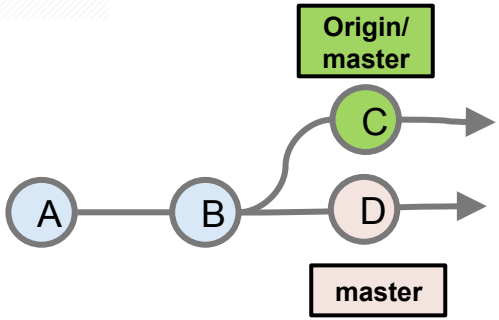
git push



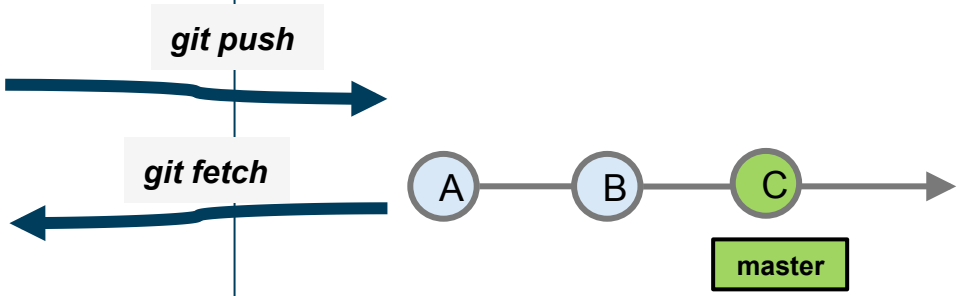


Remote repositories

Local repository:



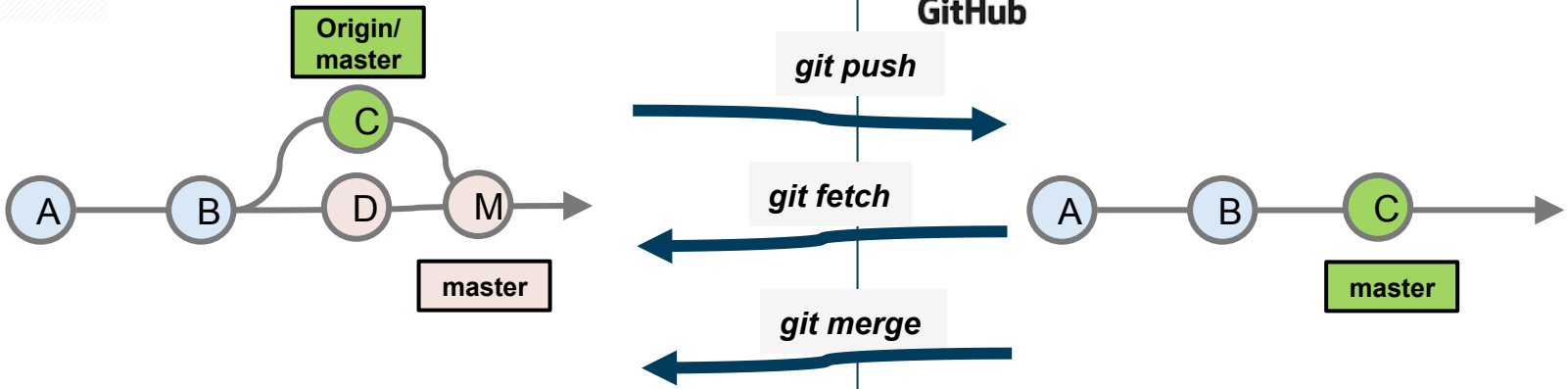
Remote repository:



Remote repositories

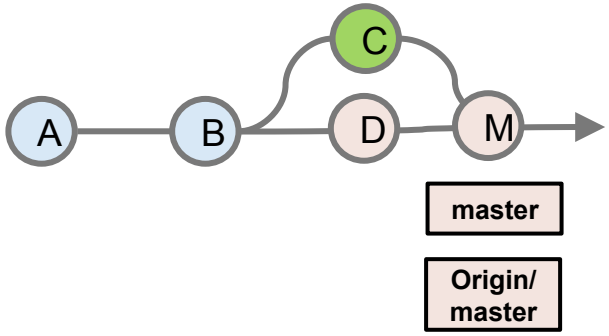
Local repository:

Remote repository:



Remote repositories

Local repository:



GitHub

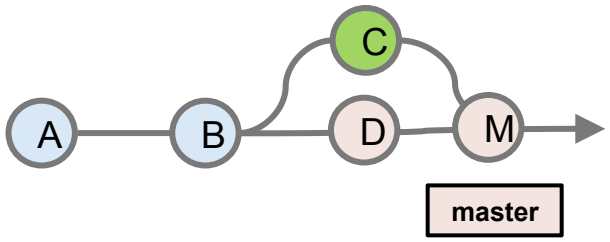
git push

git fetch

git merge

git push

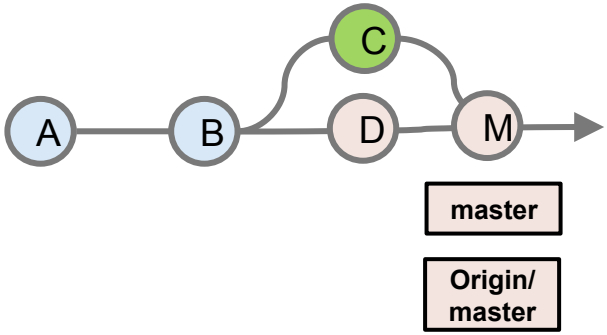
Remote repository:

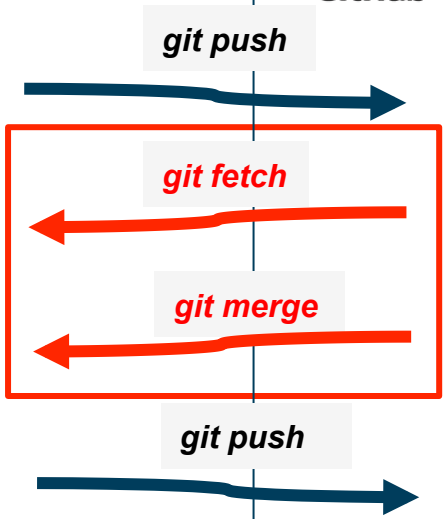


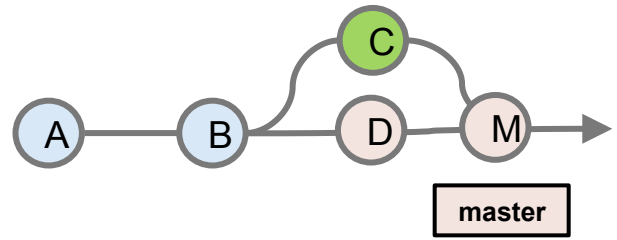
Remote repositories

Local repository:

Remote repository:





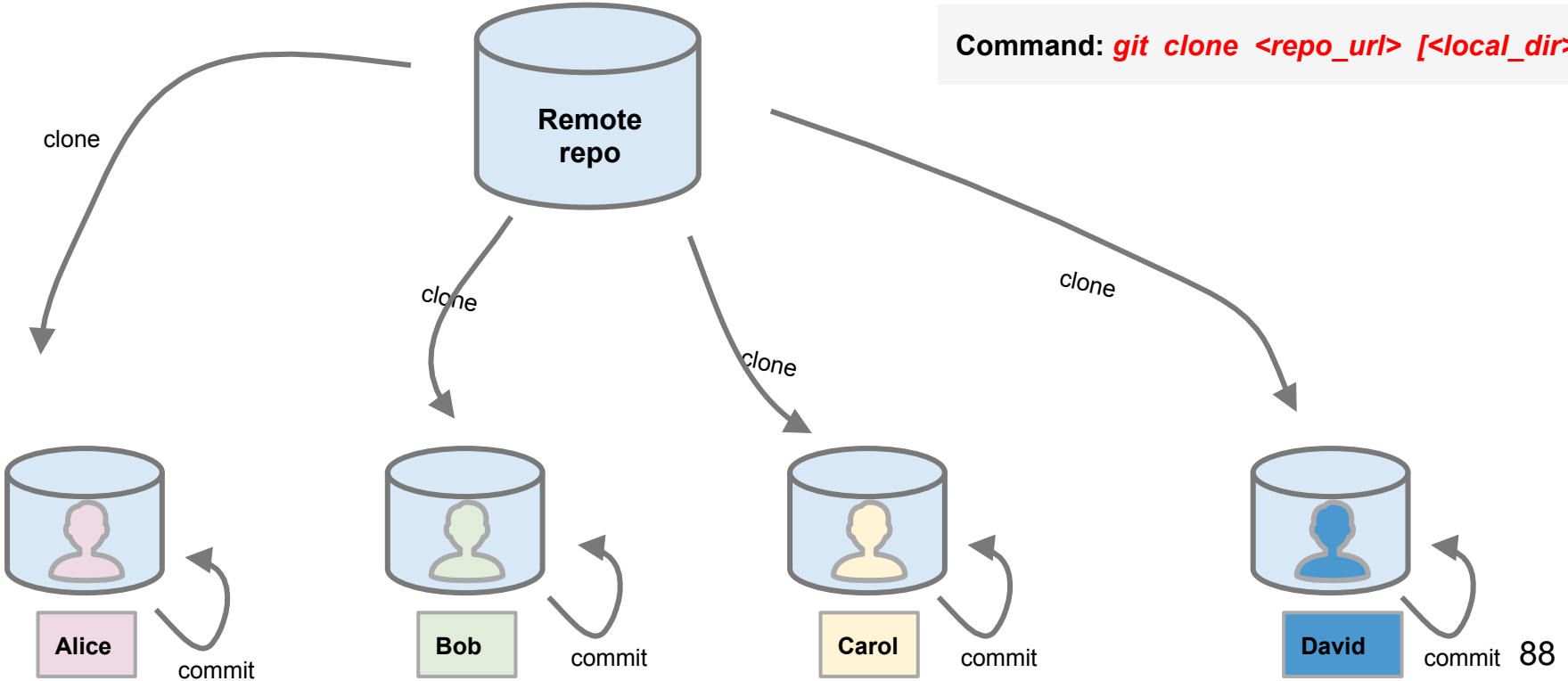


Version Control Systems: Outline

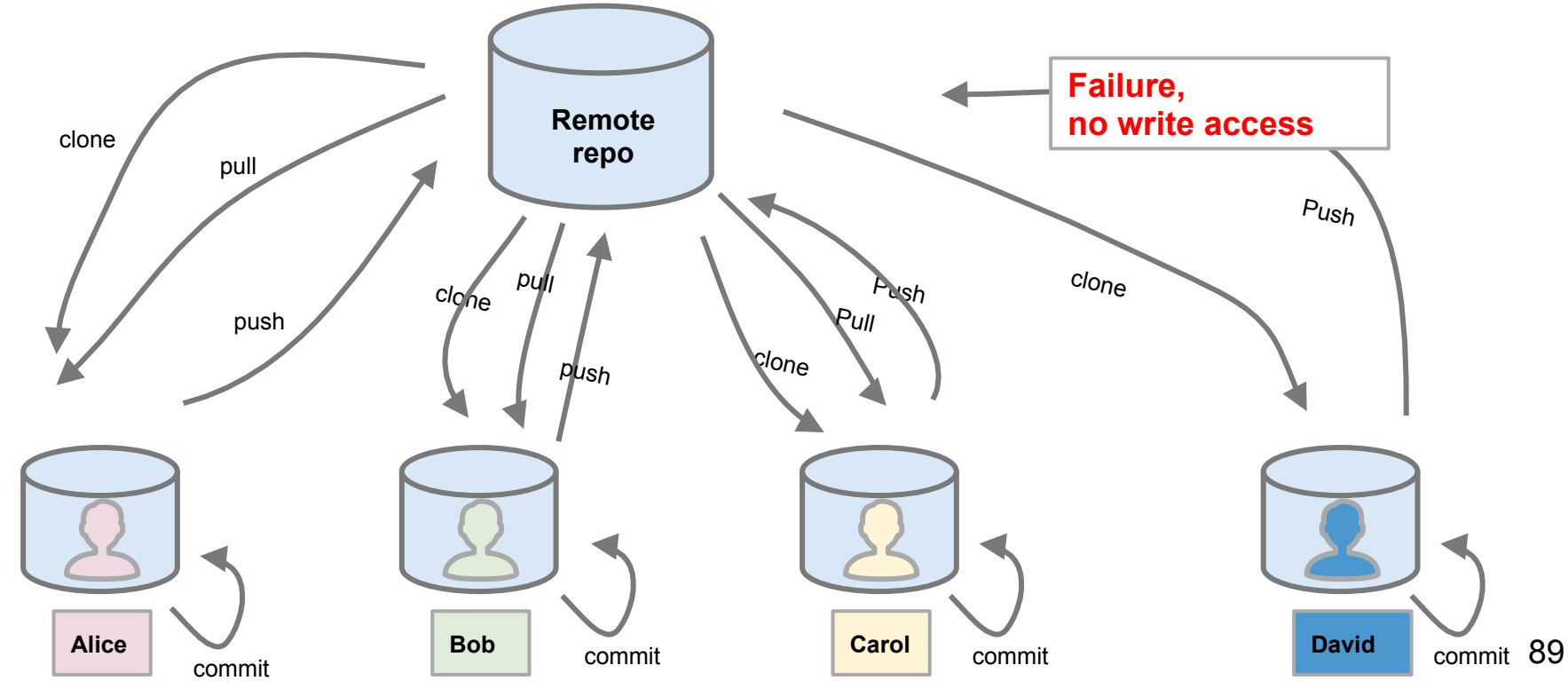
- Types of Version Control Systems
- Git
  - Branching
  - Merging
  - Rebasing
  - Squashing
  - Cherry-pick
  - Remote repositories
- GitHub
  - Cloning vs. Forking
  - Making pull request
- Branching strategies

Cloning vs. Forking a repo

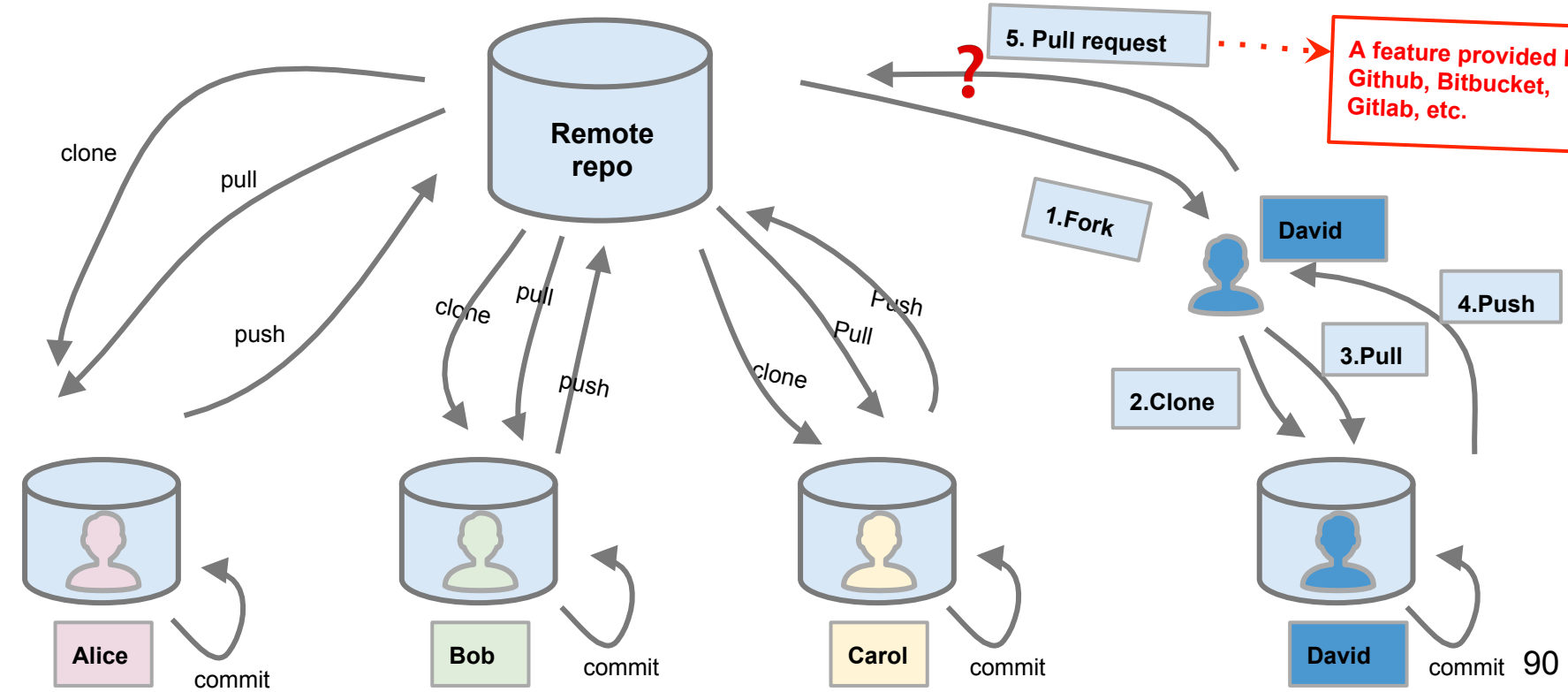
Command: ***git clone <repo_url> [<local_dir>]***



Cloning vs. Forking a repo



Cloning vs. Forking a repo



Forking

- A fork is a copy of a repository.
- Forking allows you to freely experiment with changes without affecting the original project.
- Most commonly, forks are used to either
  - propose changes to someone else's project or
  - use someone else's project as a starting point for your own idea.

Forking - Propose changes to someone else's project

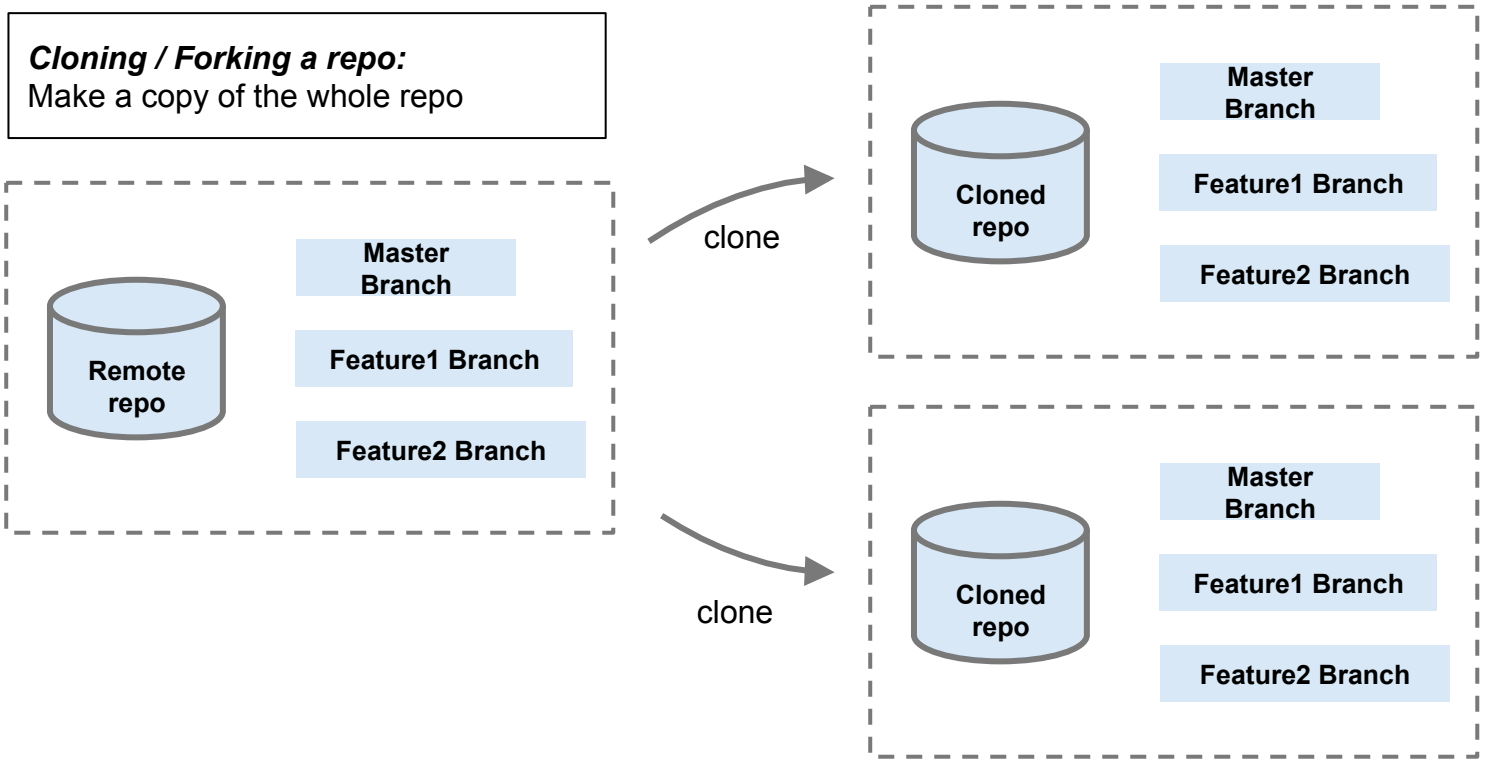
- Process:
  - Fork the repository.
  - Make the fix.
  - Submit a pull request to the original project repository.
- If the project owner likes your work, they can pull your fix into the original repository!

Forking - Use someone else's project as a starting point for your own idea

- Make sure to respect the license file of the original repository and to include a license file that determines how you want your project to be shared with others!

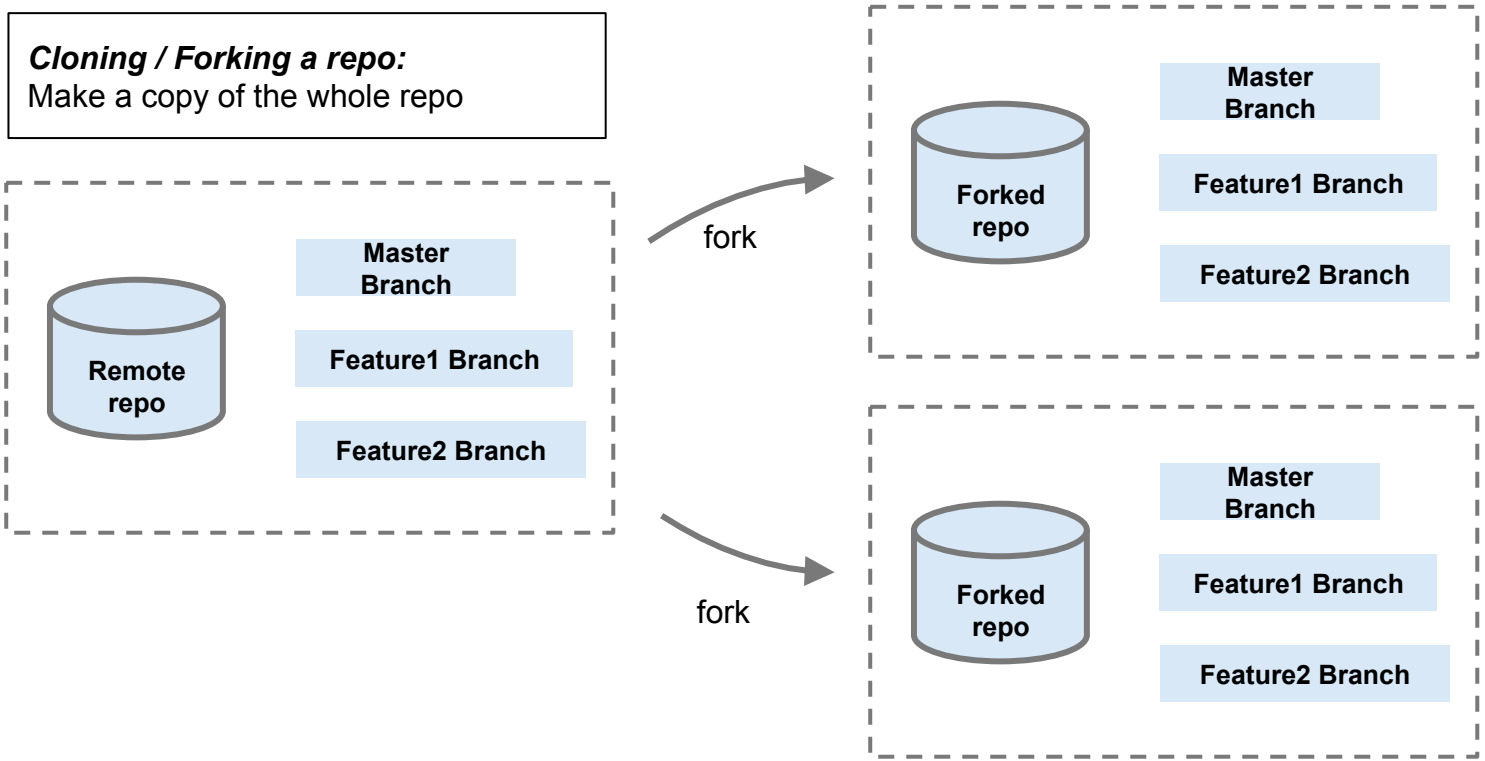
Cloning/forking a repo vs. branching

Cloning / Forking a repo:
Make a copy of the whole repo



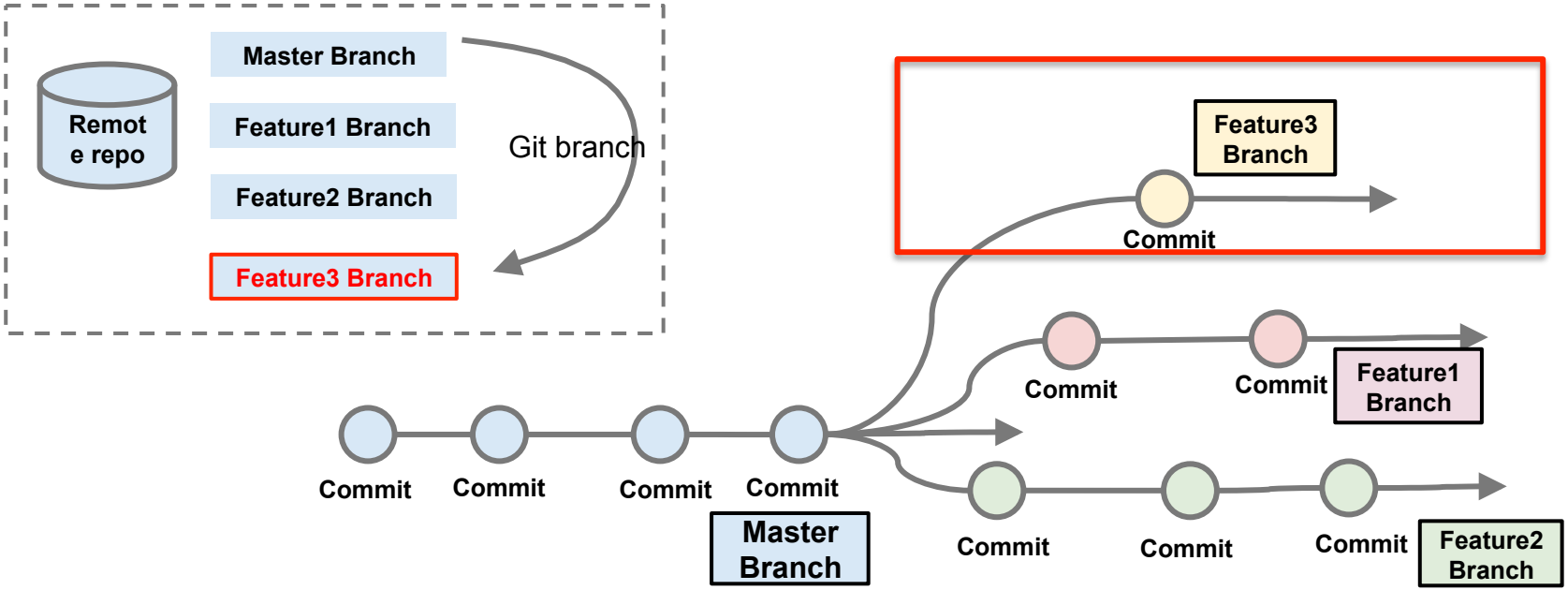
Cloning/forking a repo vs. branching

Cloning / Forking a repo:
Make a copy of the whole repo



Cloning/forking a repo vs. branching

Branching:
Add another working thread inside the repo

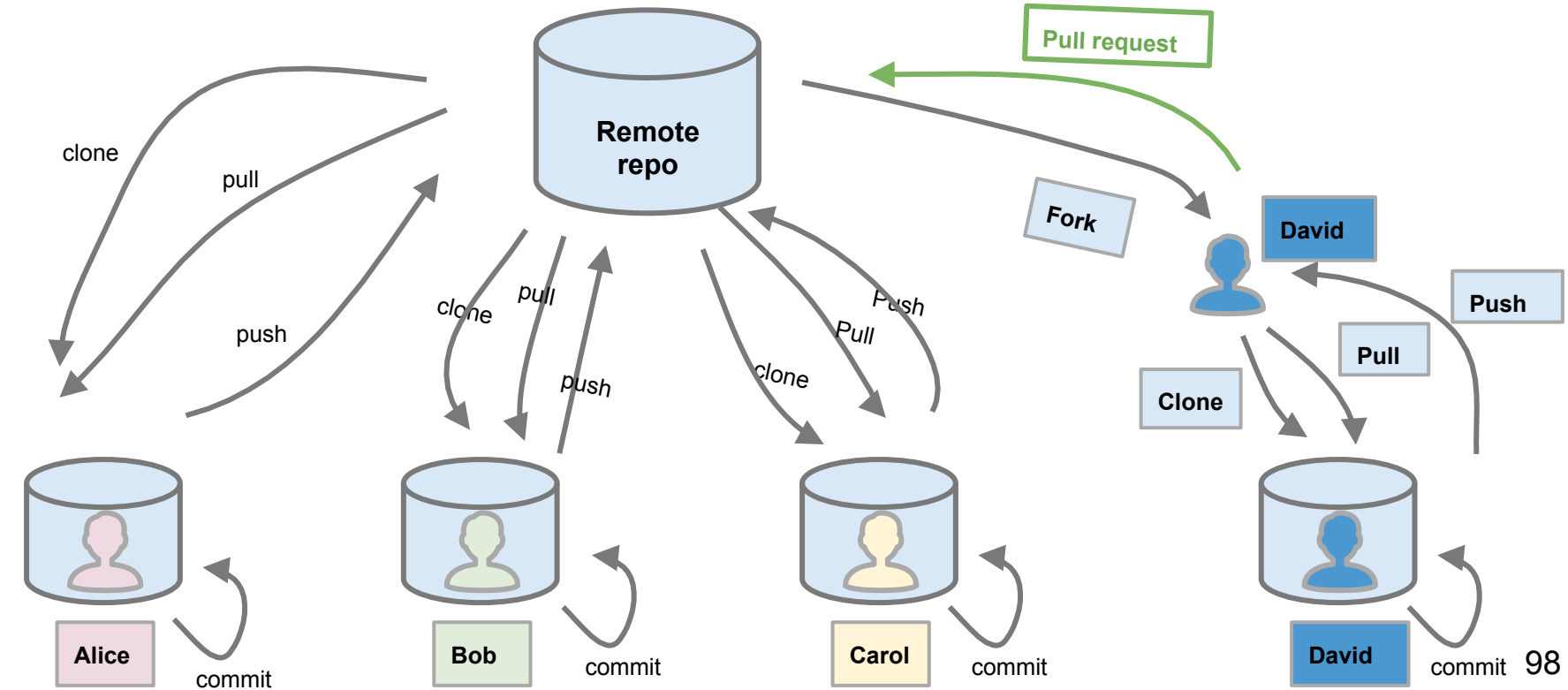


Making a Pull Request

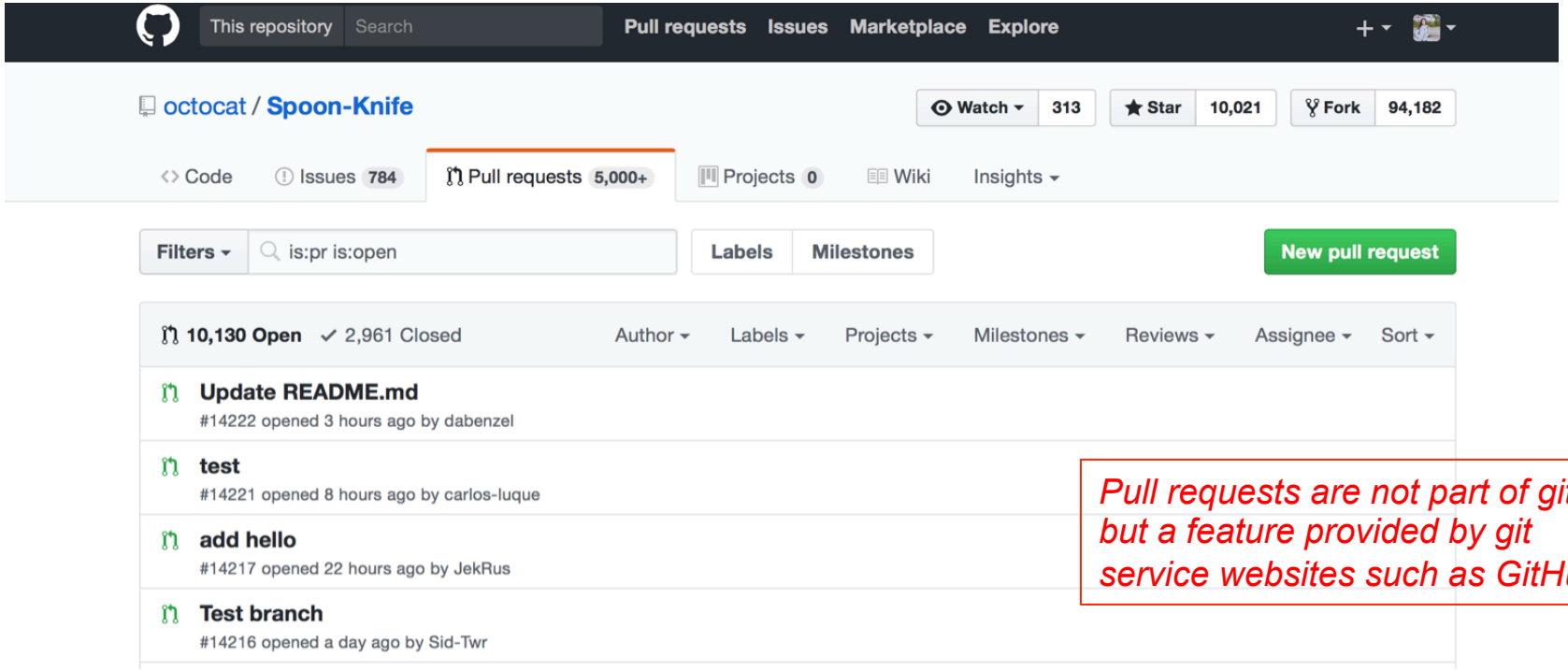
- If you want to share changes, which you made locally, you have two options:
  - If you have ***write*** access to the repo, you can push back to the remote repo directly
  - If not – you need to make a pull request

- Pull requests are also used between collaborators of a repo - for discussion and notification

Making a Pull Request - push back to the forked repo



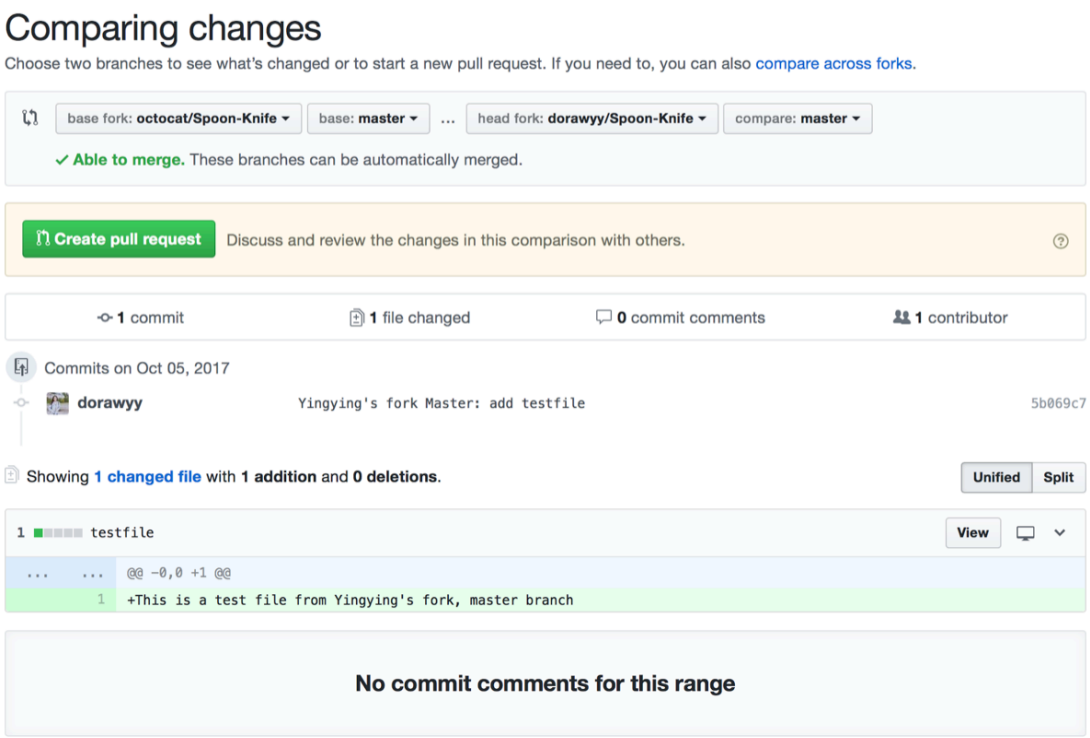
Making a Pull Request



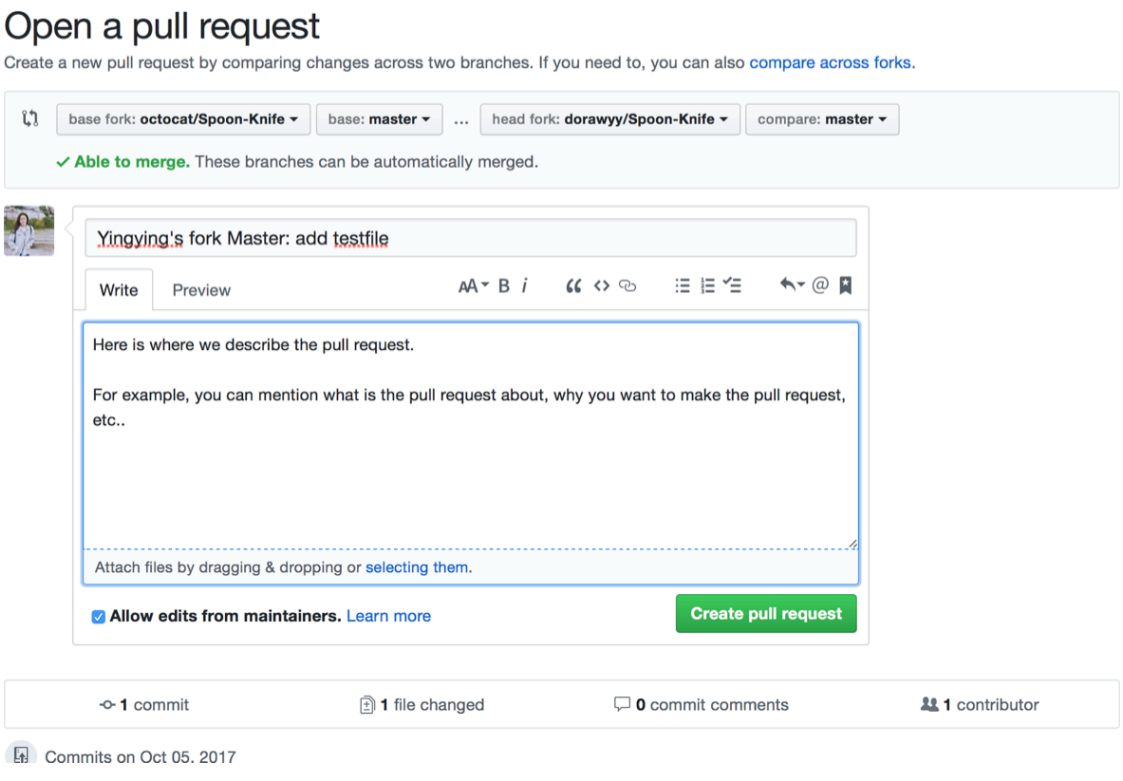
Pull requests are not part of git, but a feature provided by git service websites such as GitHub

https://github.com/octocat/Spoon-Knife/pulls

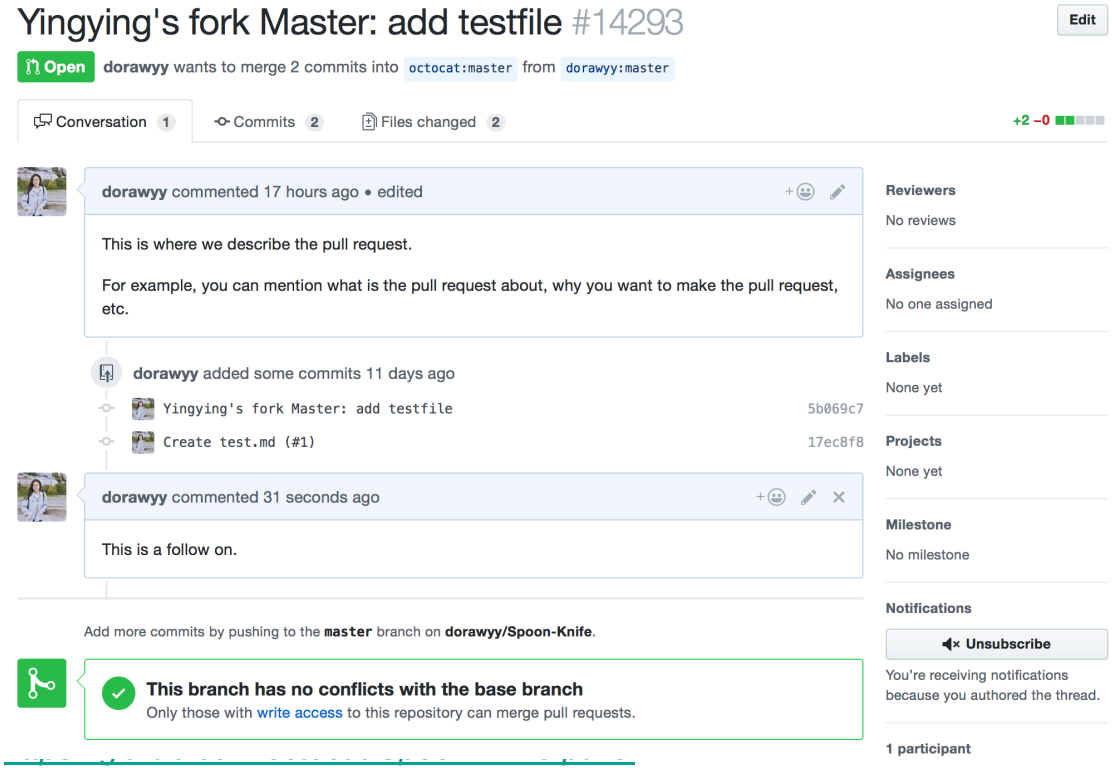
Making a Pull Request



Making a Pull Request

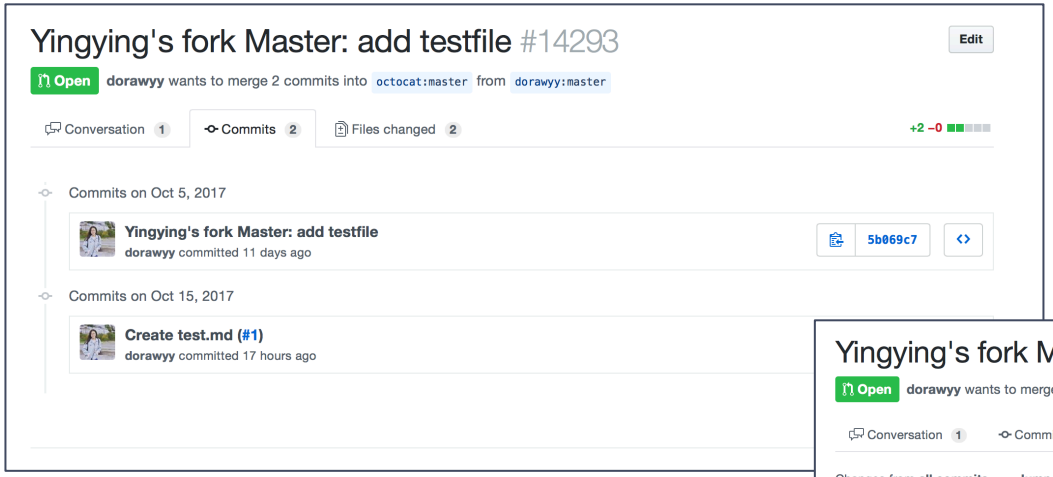


Making a Pull Request



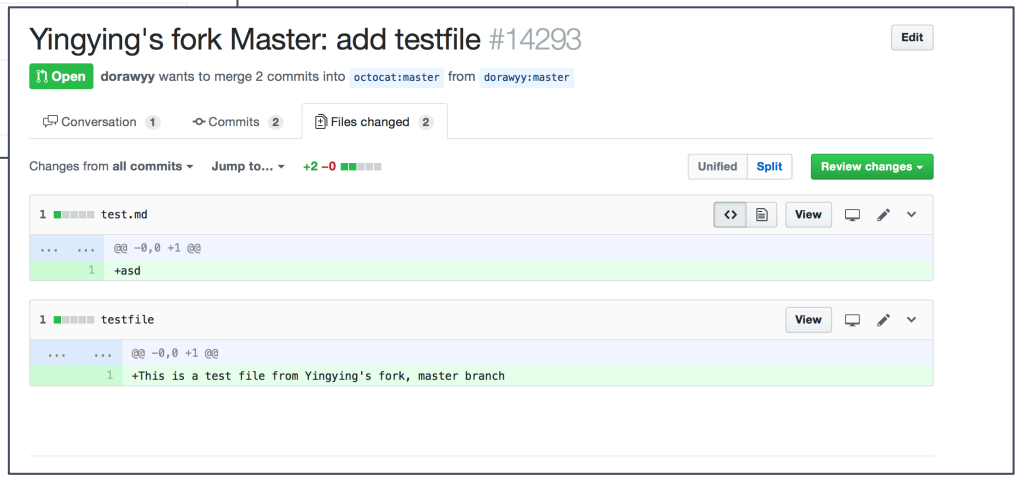
Pull request is submitted successfully

Making a Pull Request



Information listed inside the pull request:

- *All conversation*
- *All commits in the pull request*
- *All files changed in the pull request*
- *…*



https://github.com/octocat/Spoon-Knife/pulls

Version Control Systems: Outline

- Types of Version Control Systems
- Git
  - Branching
  - Merging
  - Rebasing
  - Squashing
  - Cherry-pick
  - Remote repositories
- GitHub
  - Cloning vs. Forking
  - Making pull request
- Branching strategies

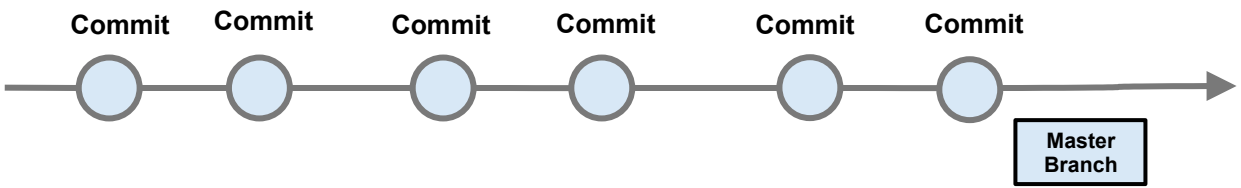
Branching strategies

- Master only workflow
- Master / develop workflow
- Feature branches
- Release branches
- Gitflow

Master only workflow

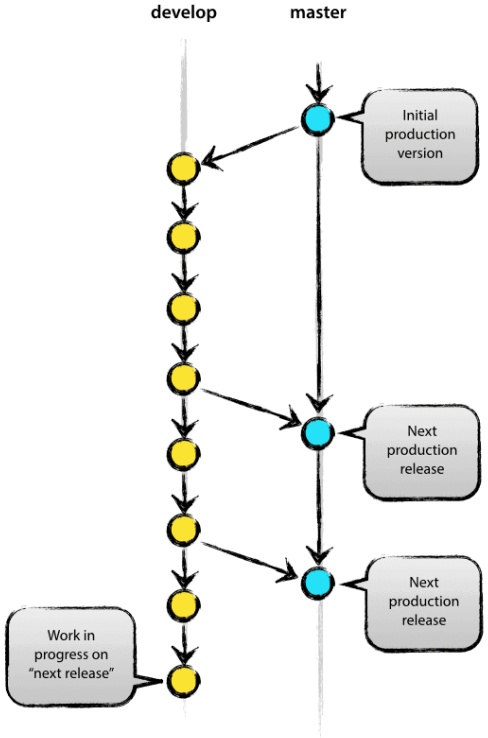
Mater only workflow:

- Everyone **works on the *master* branch** only without branching
- Always pull before push commits to the remote repo



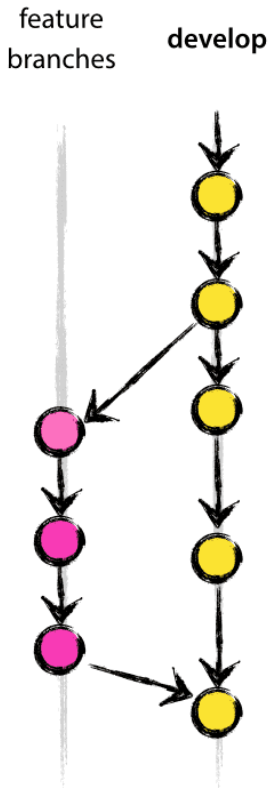
Master / Develop workflow

- The central repo holds two main branches with an infinite lifetime: master and develop
- Master is the main branch where the source code of HEAD always reflects a *production-ready* state.
- Develop to be the main branch where the source code of HEAD always reflects a state with the latest delivered development changes for the next release.
  - Some would call this the “integration branch”.
  - This is where any automatic nightly builds are built from.



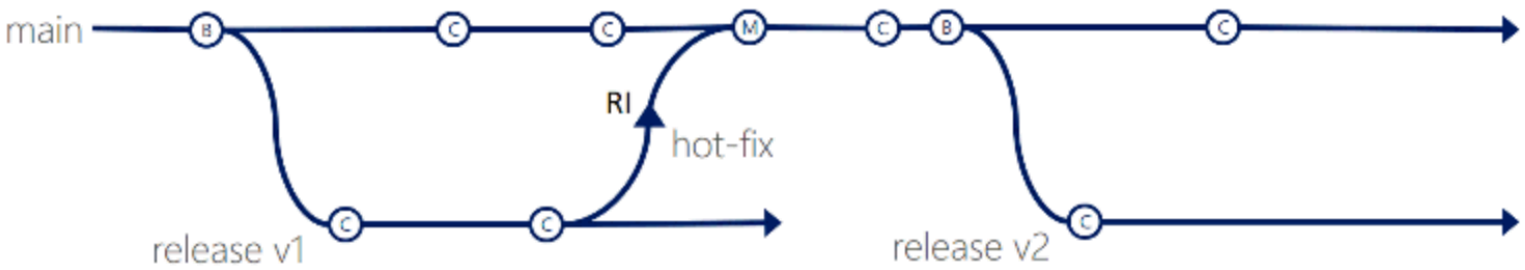
Feature branches

- Feature branches (or sometimes called topic branches) are used to develop new features for the upcoming or a distant future release.
- When starting development of a feature, the target release of the feature may be unknown
- The feature branch exists as long as the feature is in development, but will eventually be merged back into develop (to definitely add the new feature to the upcoming release) or discarded (in case of a disappointing experiment).
- Enables concurrent feature development



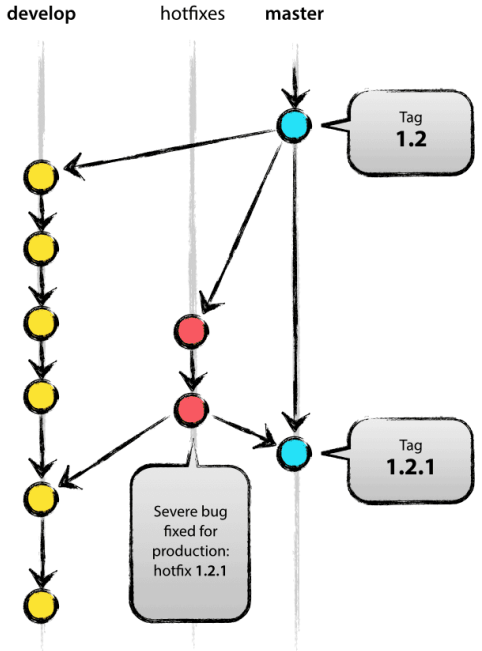
Release branches

- Create a branch for each upcoming release
- Enabling concurrent release management, multiple and parallel releases



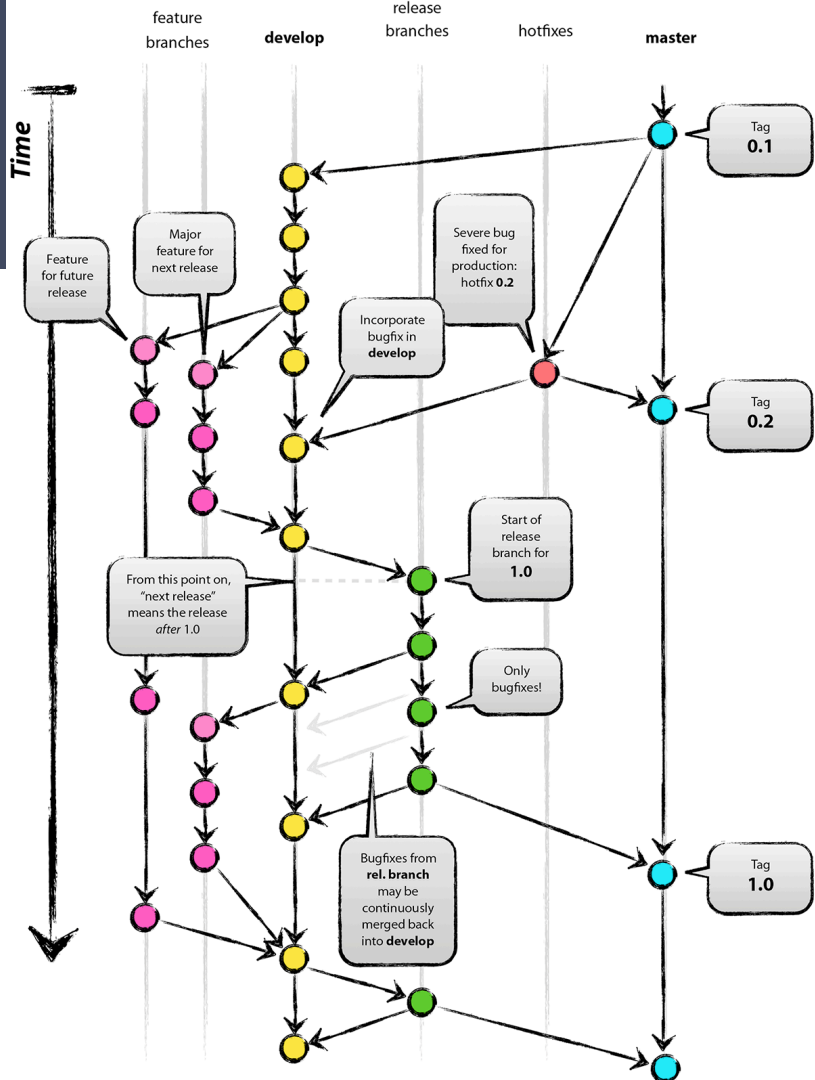
Other types of branches

- Hotfix
- QA
- …



Gitflow

- **Master** branch is always deployable.
- **Develop** branch is the center of development work
- **Release** branches keeps track of all releases
- **Hotfix** branches are similar to release branches but solve immediate “emergencies”
- **Feature** branches are used to develop individual features



Version Control Systems: Outline

- Types of Version Control Systems
- Git
  - Branching
  - Merging
  - Rebasing
  - Squashing
  - Cherry-pick
  - Remote repositories
- GitHub
  - Cloning vs. Forking
  - Making pull request
- Branching strategies

More Info

[1] A successful Git branching model

[2] Using Github inside a company

[3] Comparing workflows

[4] Understanding the Github flow

[5] Git workflows for pros: a good git guide

https://medium.com/@afeinman/knowing-git-ab8f60550e2b