Git for Developers

Introduction

Outline

- Version Control
- Centralized vs. Decentralized
- Parallel Development
- Deltified Storage vs. Snapshot Storage
- Git

Version Control

- The purpose of version control (VC):
  - History – who / did what / to what / when / why
- To enable parallel development, which could include:
  - **Multiple independent efforts** (e.g. maintenance on prior release while building the next). Branching handles this type of parallel development
  - **Multiple people on one line of development** (i.e., collaborative editing and sharing of data)
- Implementation flavors: centralized or distributed

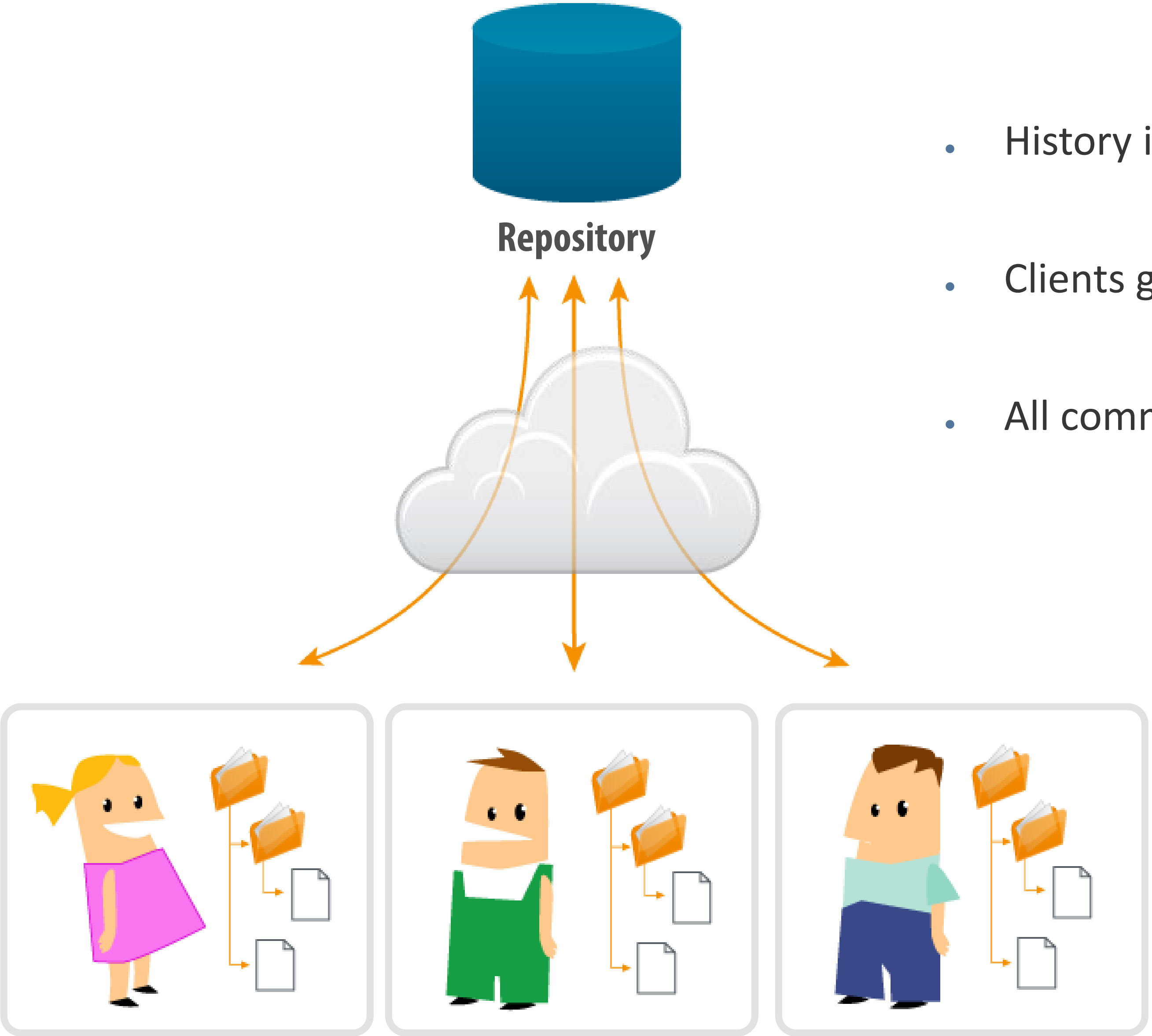
NOTE

Version control is a part of software configuration management.

* There are many different definitions of Software Configuration Management (SCM)

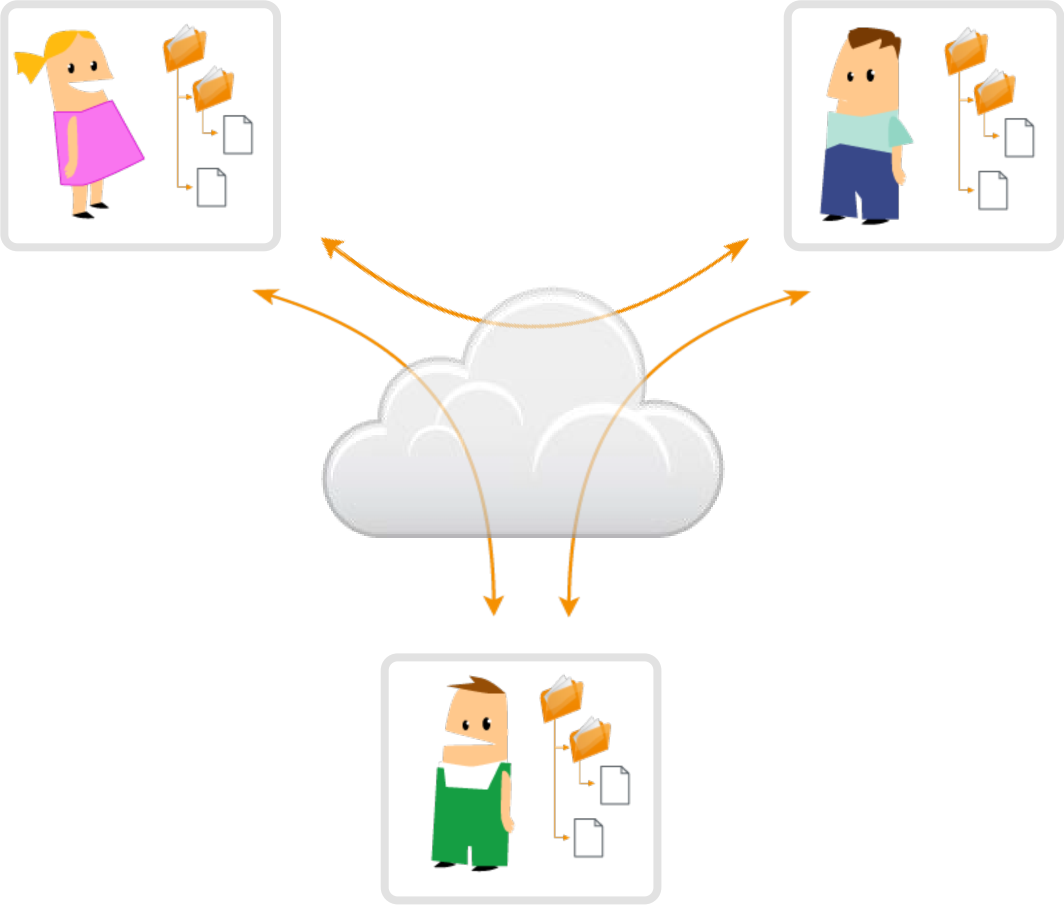
* Beyond version control, they typically include; build management, release management, defect tracking, configuration management and process automation

Centralized version control



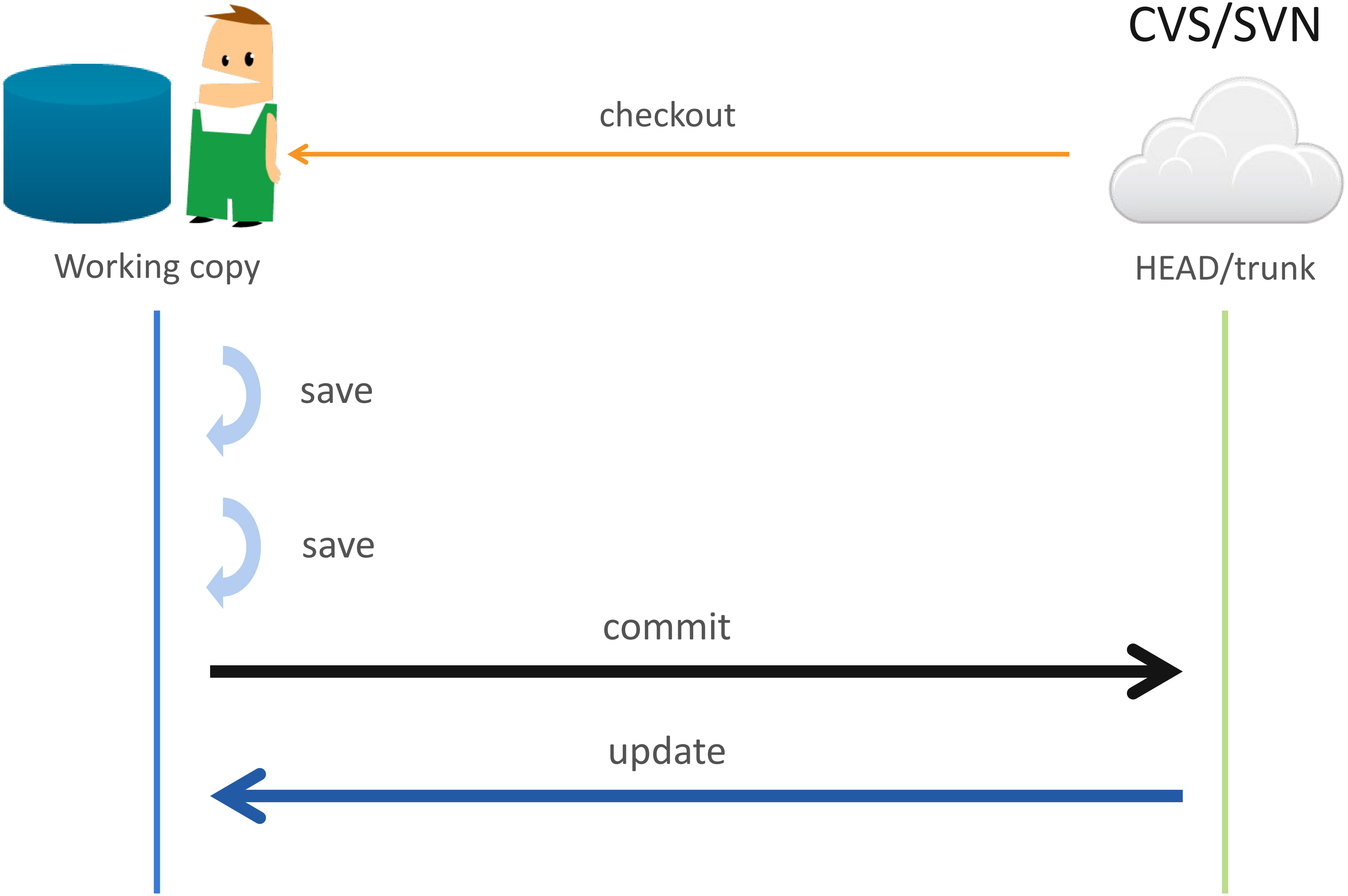
- History in one repository
- Clients get single revision
- All commits go into the one repository

Distributed version control



- Each user has at least one copy (clone) of the repository
- Each 'user' repository holds the full history
- There is a 'Main' repository only by convention

Centralized



CVS/SVN

checkout

Working copy

HEAD/trunk

save

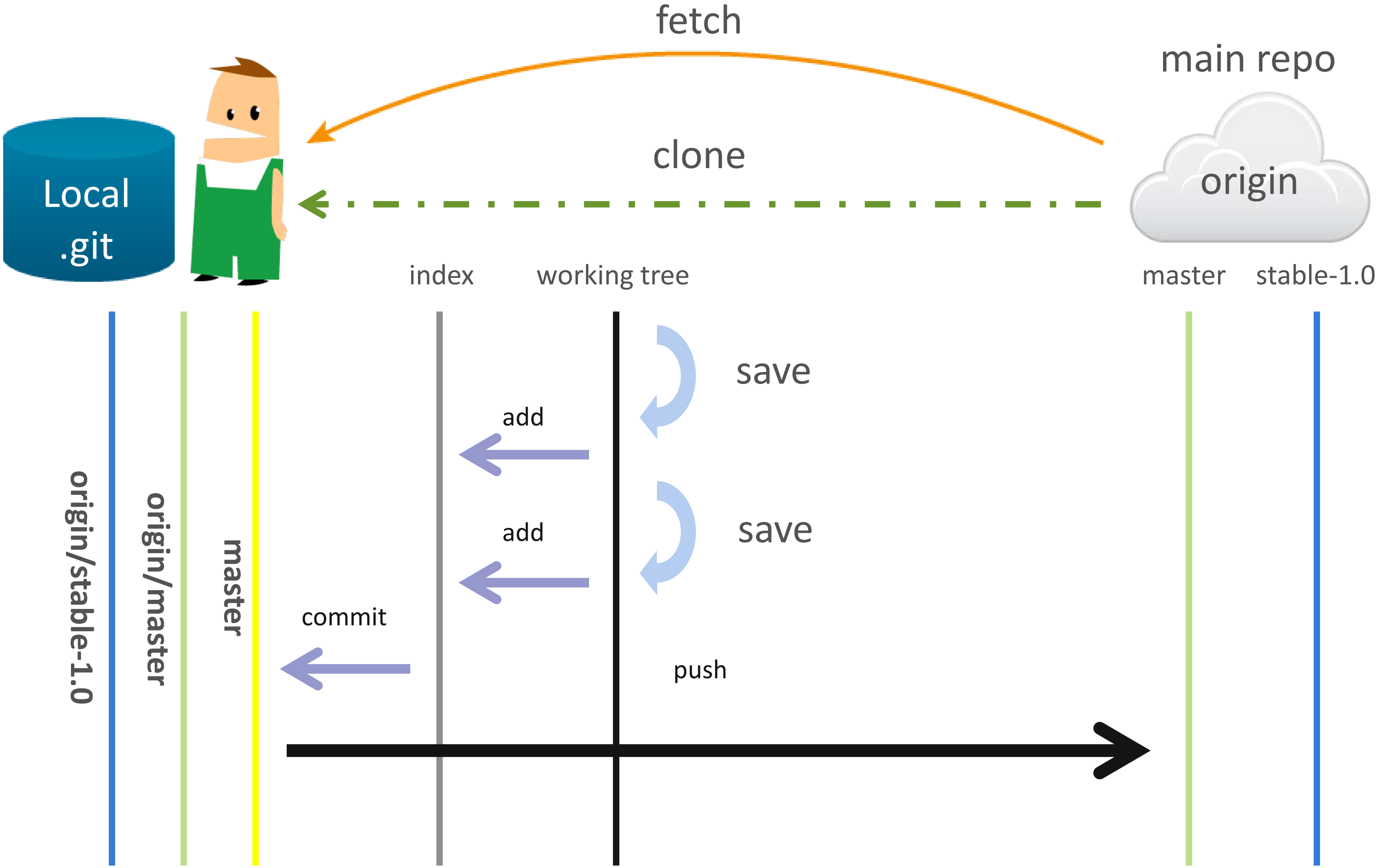
save

commit

update

Distributed

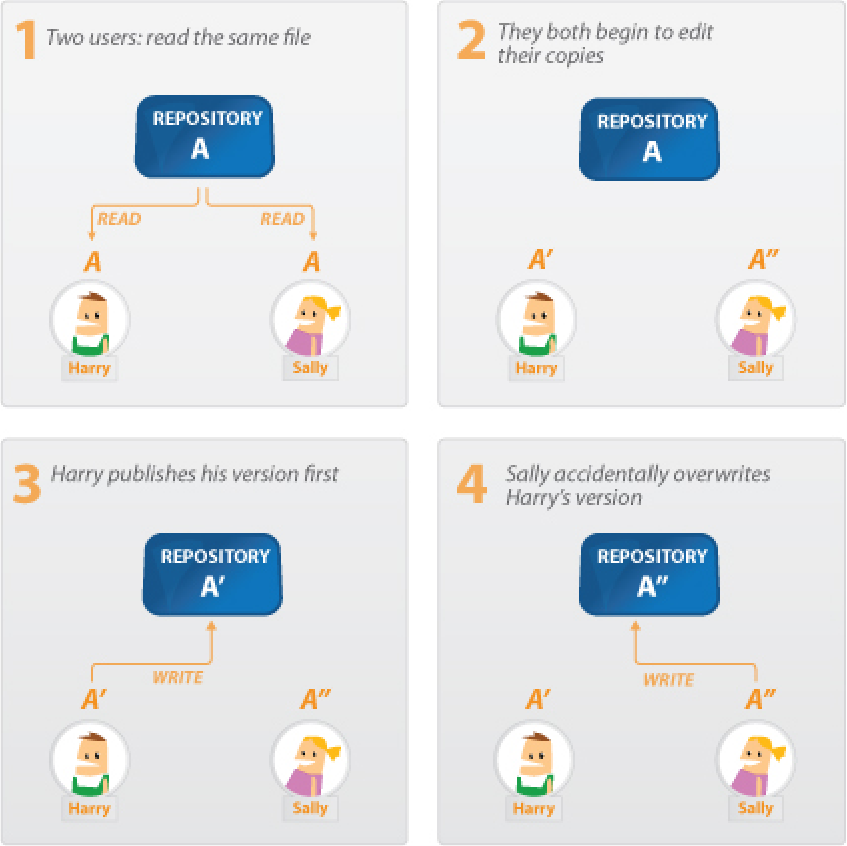
Git



Parallel development

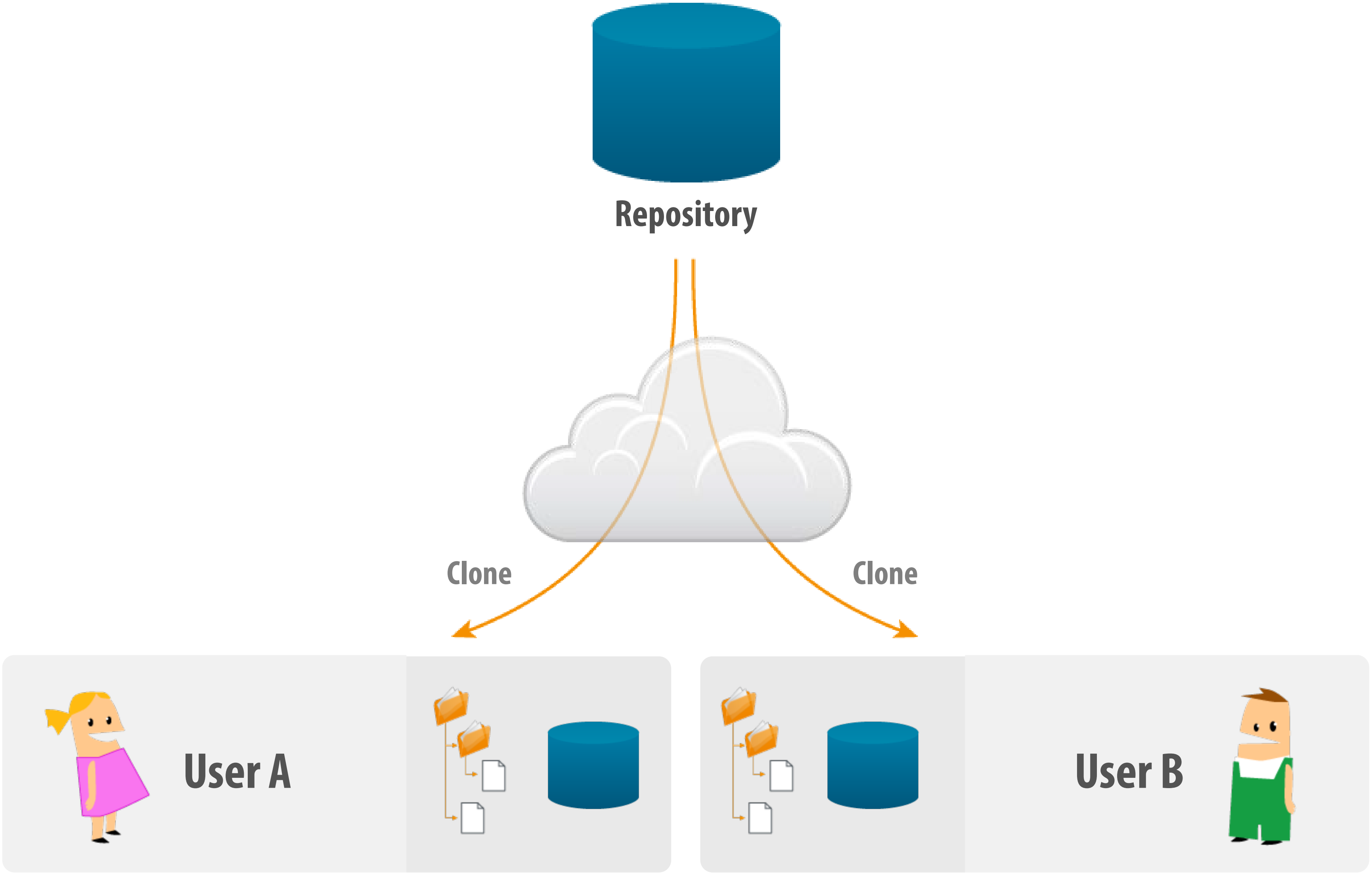
- A fundamental challenge version control systems have to solve is: how to work in parallel while preventing one user from overwriting the work of another.

- Two solution flavors:
  - copy-modify-merge (default)
  - lock-modify-unlock
    (not supported by distributed version control systems)


1 Two users: read the same file
REPOSITORY
A
READ
READ
A
A
Harry
Sally
2 They both begin to edit their copies
REPOSITORY
A
A′
A″
Harry
Sally
3 Harry publishes his version first
REPOSITORY
A′
WRITE
A′
A″
Harry
Sally
4 Sally accidentally overwrites Harry's version
REPOSITORY
A″
WRITE
A′
A″
Harry
Sally

Copy – Modify - Merge

Two users clone a main repository, each work in their local repository.

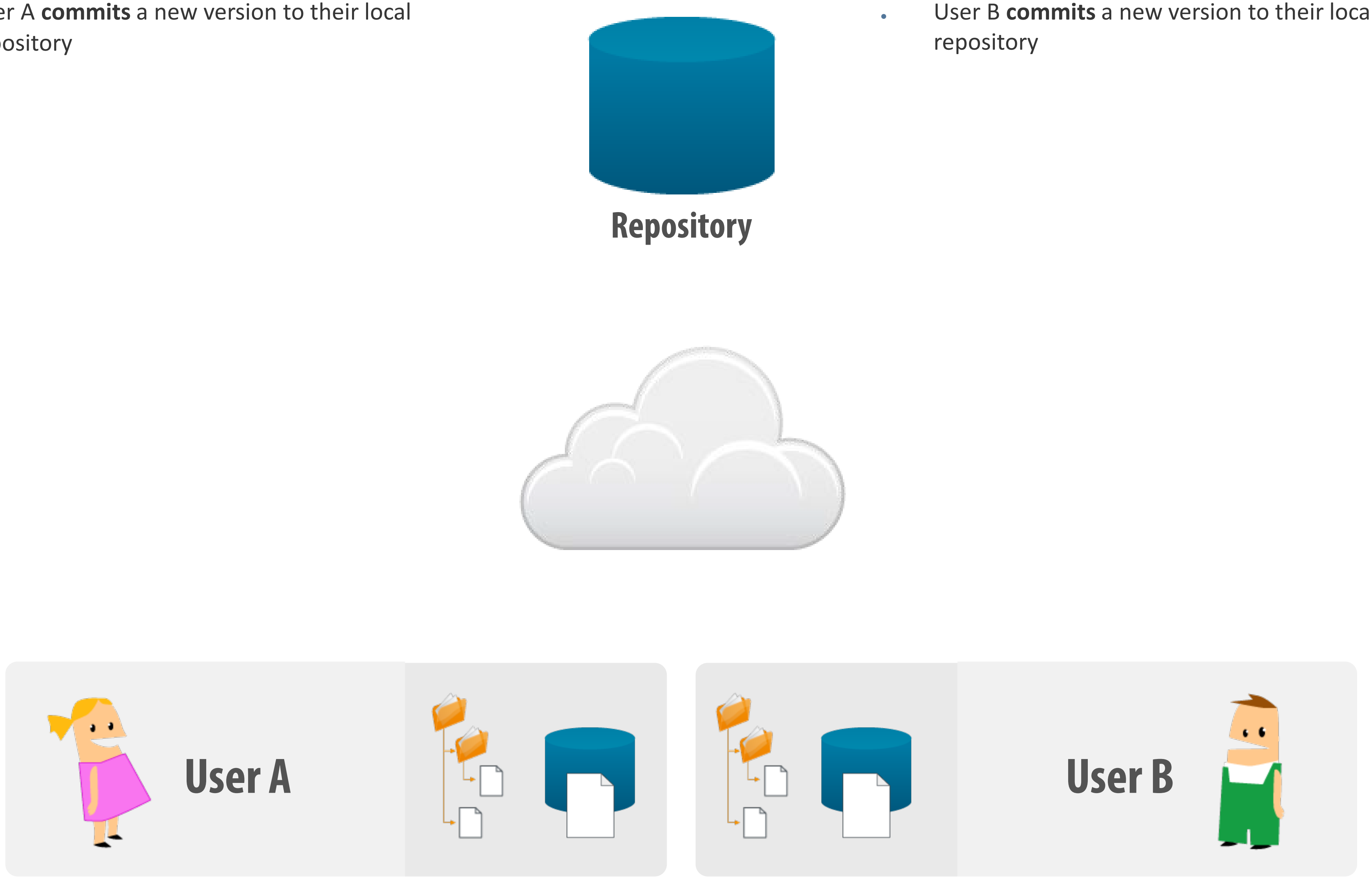


Copy – Modify - Merge

Two users clone a main repository, each work in their local repository.

- User A **commits** a new version to their local repository

- User B **commits** a new version to their local repository

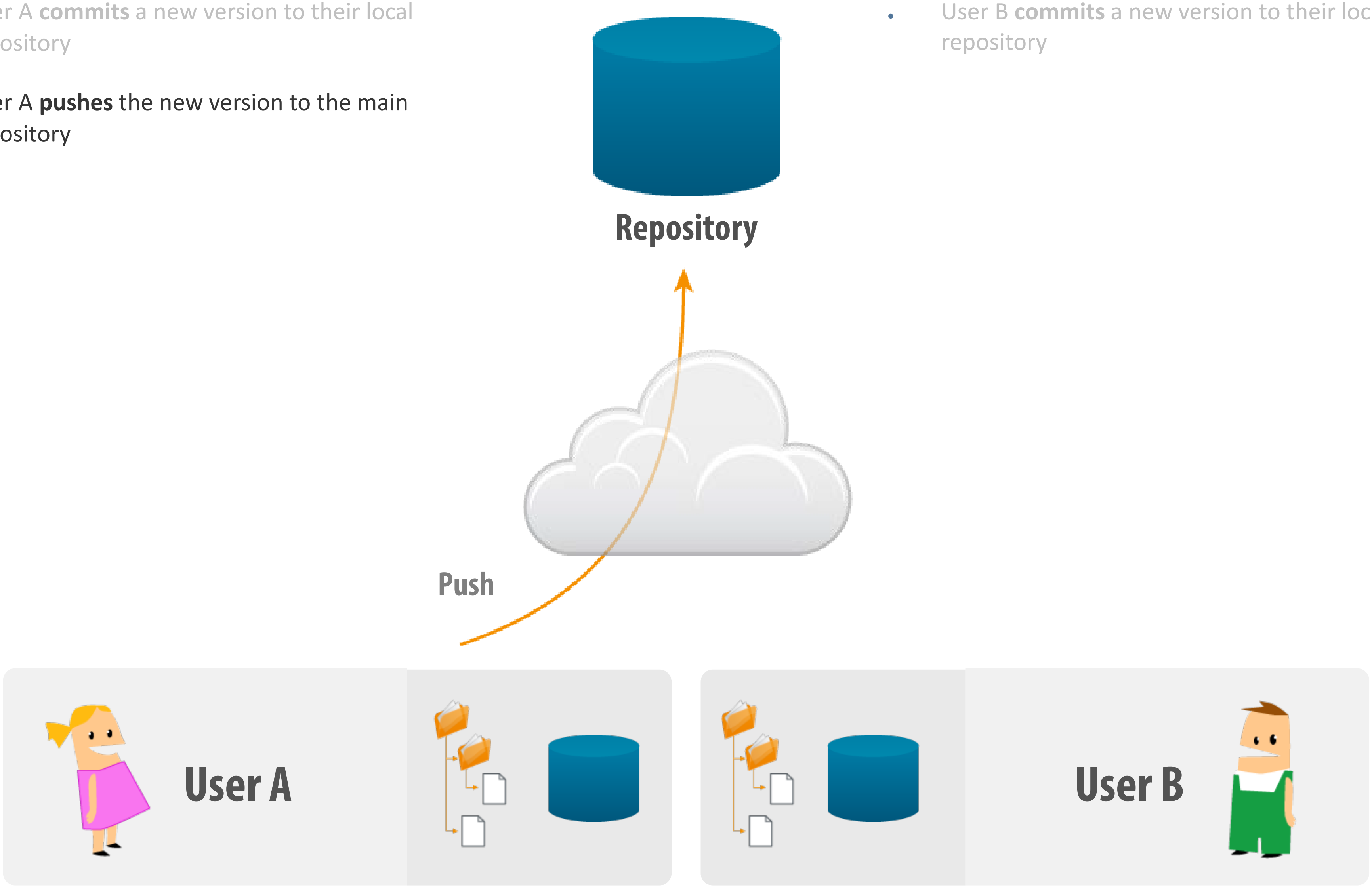


Copy – Modify - Merge

Two users clone a main repository, each work in their local repository.

- User A **commits** a new version to their local repository
- User A **pushes** the new version to the main repository

- User B **commits** a new version to their local repository

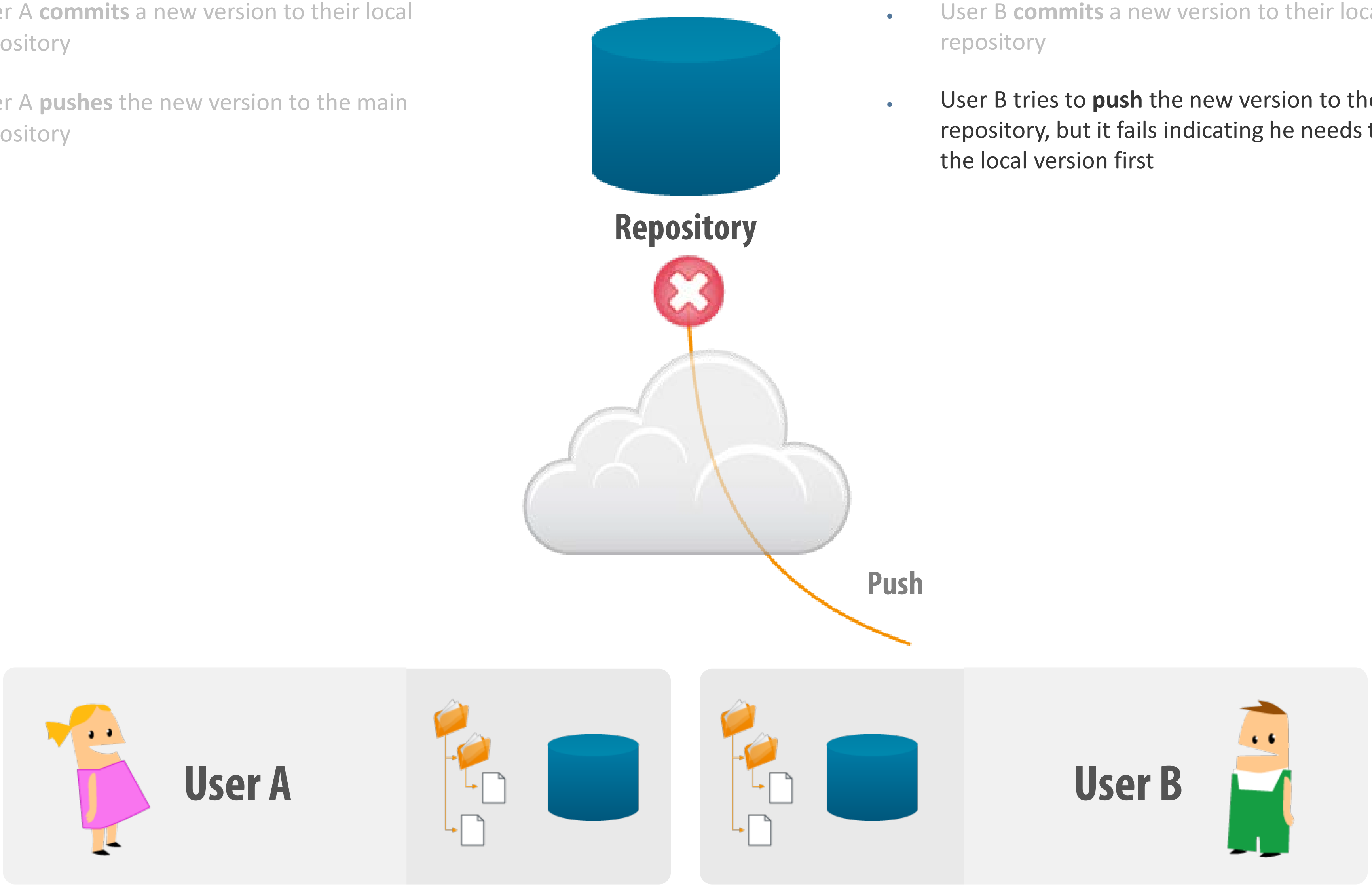


Copy – Modify - Merge

Two users clone a main repository, each work in their local repository.

- User A **commits** a new version to their local repository
- User A **pushes** the new version to the main repository

- User B **commits** a new version to their local repository
- User B tries to **push** the new version to the main repository, but it fails indicating he needs to update the local version first

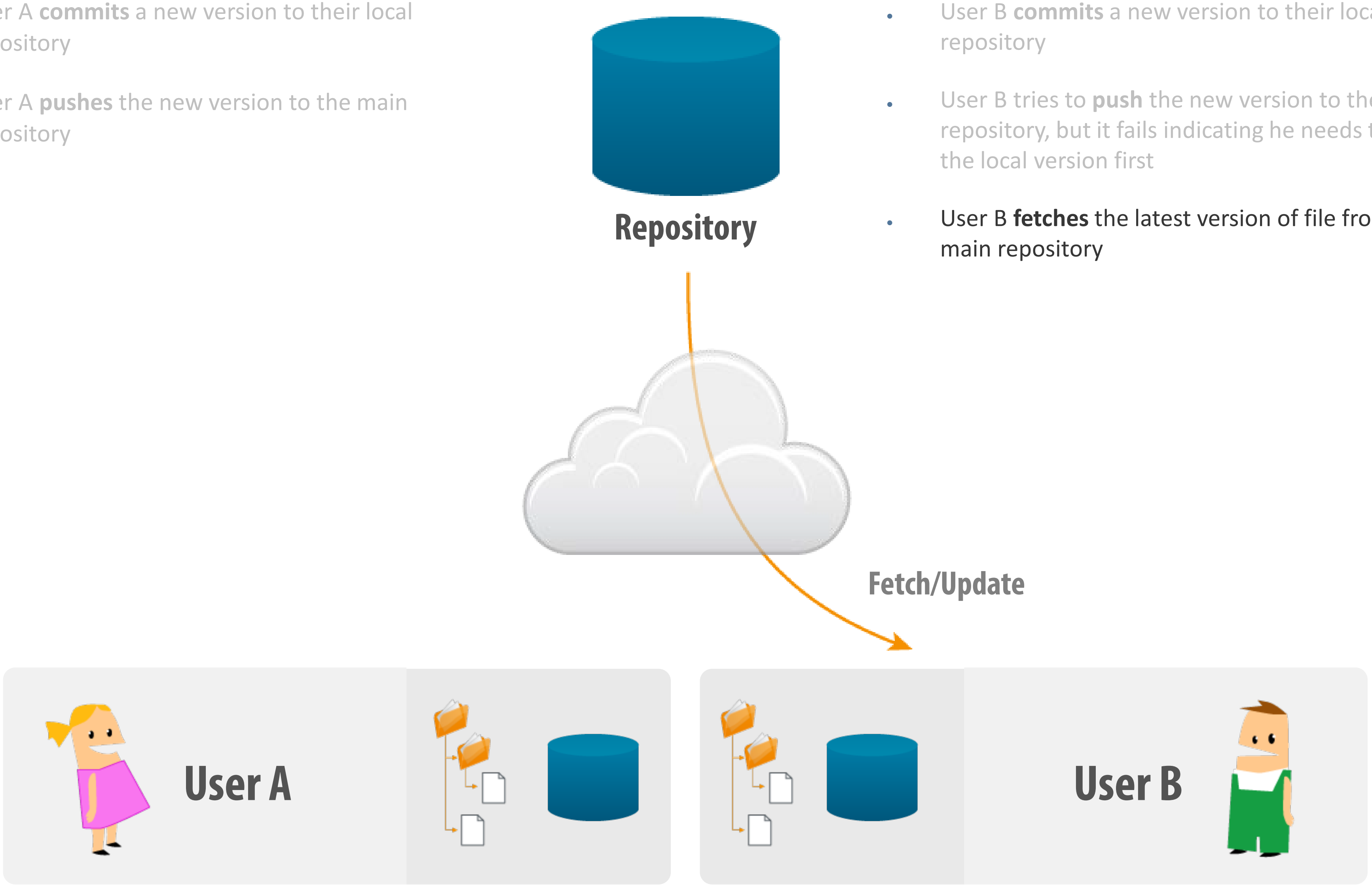


Copy – Modify - Merge

Two users clone a main repository, each work in their local repository.

- User A **commits** a new version to their local repository
- User A **pushes** the new version to the main repository

- User B **commits** a new version to their local repository
- User B tries to **push** the new version to the main repository, but it fails indicating he needs to update the local version first
- User B **fetches** the latest version of file from the main repository



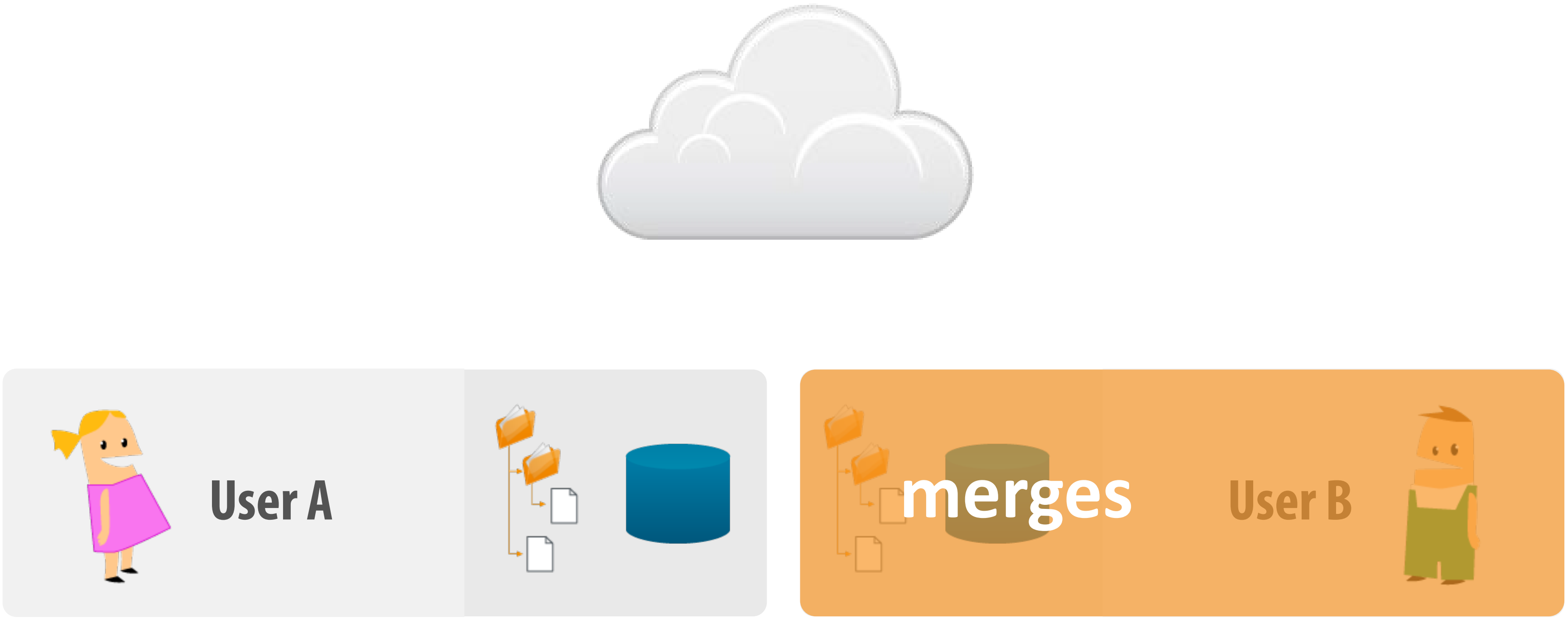
Copy – Modify - Merge

Two users clone a main repository, each work in their local repository.

- User A **commits** a new version to their local repository
- User A **pushes** the new version to the main repository

Repository

- User B **commits** a new version to their local repository
- User B tries to **push** the new version to the main repository, but it fails indicating he needs to update the local version first
- User B **fetches** the latest version of file from the main repository
- User B **merges** changes into his local version of file

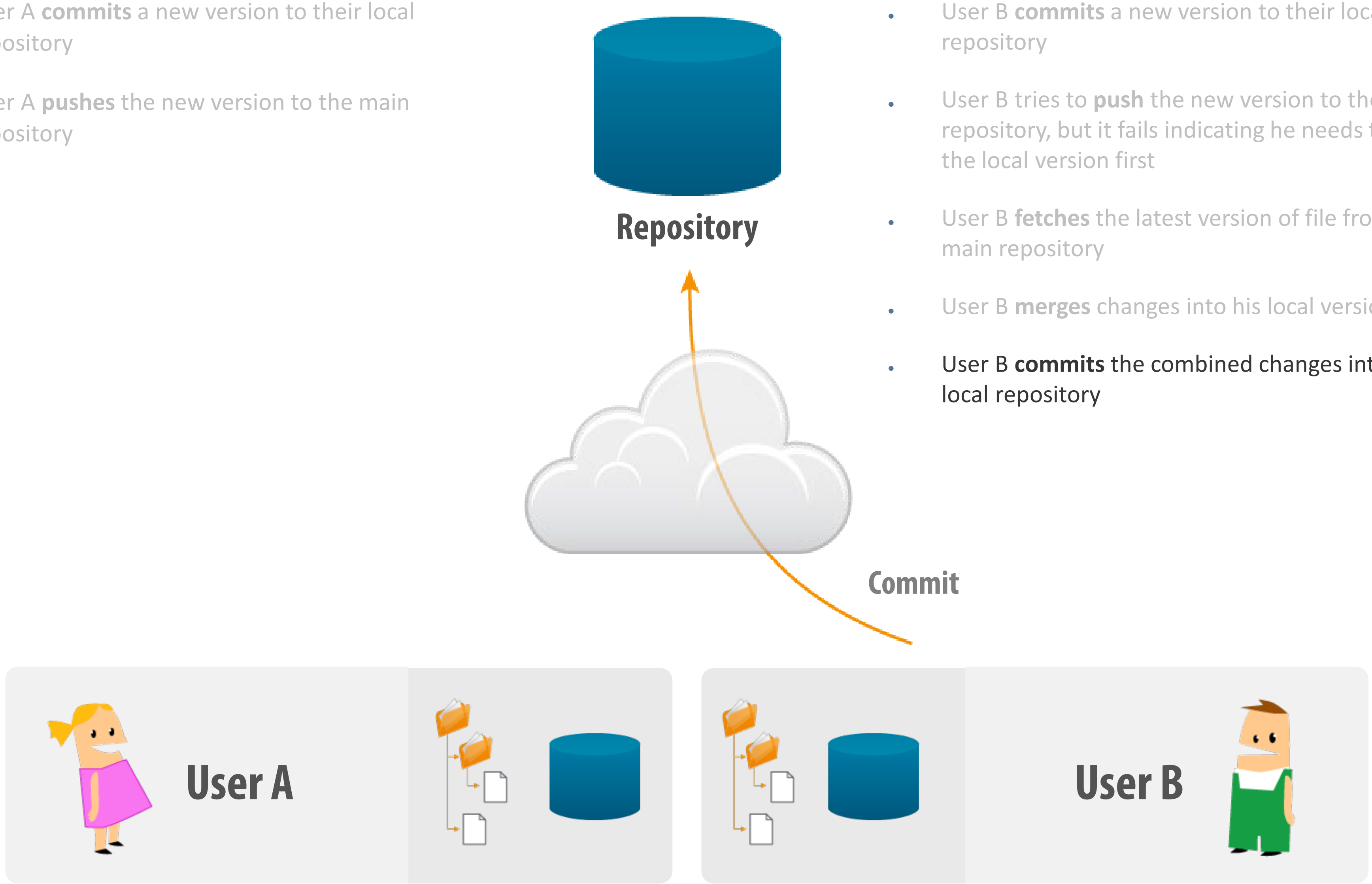


CollabNet.

Copy – Modify - Merge

Two users clone a main repository, each work in their local repository.

- User A **commits** a new version to their local repository
- User A **pushes** the new version to the main repository

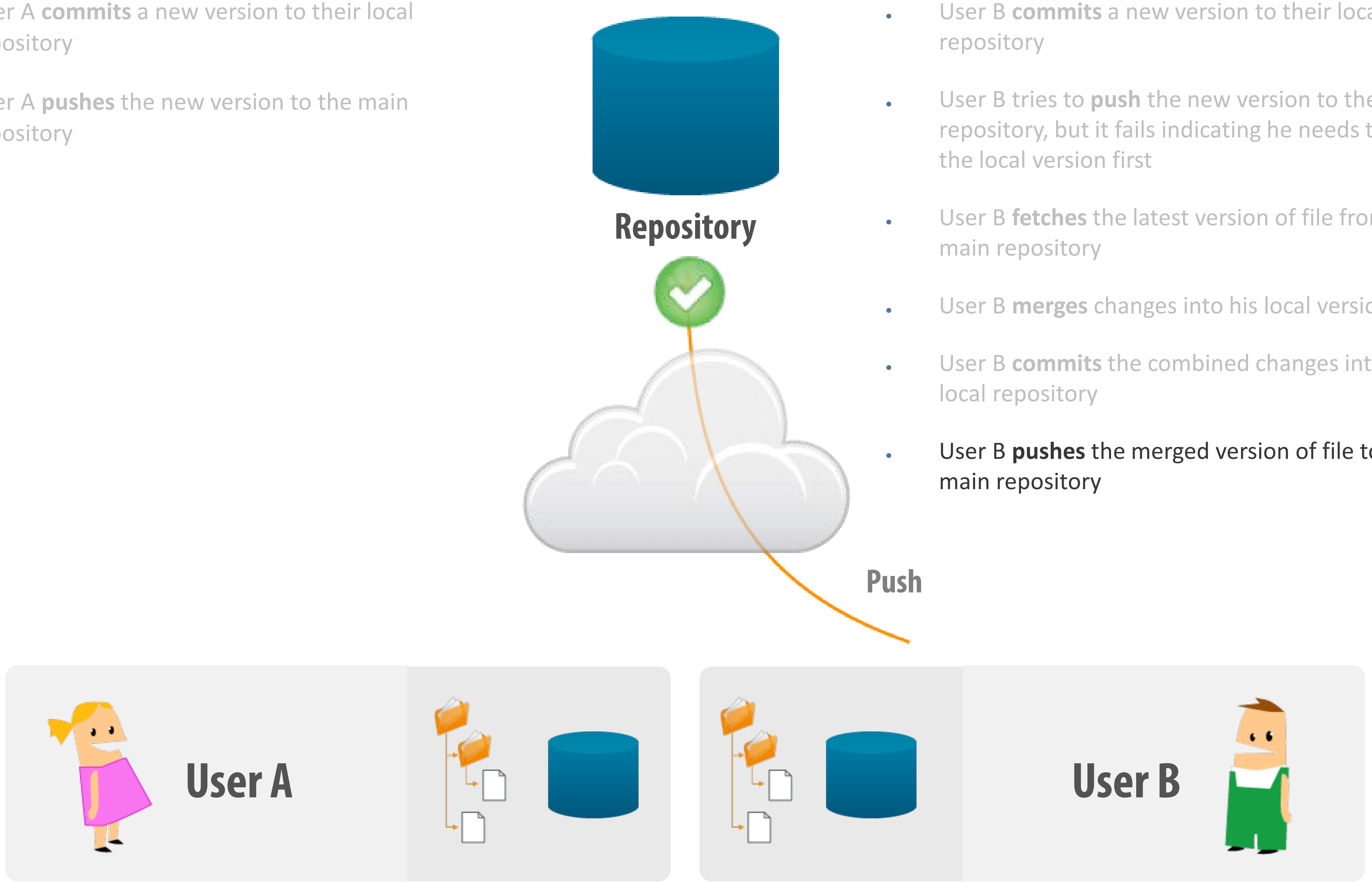


- User B **commits** a new version to their local repository
- User B tries to **push** the new version to the main repository, but it fails indicating he needs to update the local version first
- User B **fetches** the latest version of file from the main repository
- User B **merges** changes into his local version of file
- User B **commits** the combined changes into his local repository

Copy – Modify - Merge

Two users clone a main repository, each work in their local repository.

- User A **commits** a new version to their local repository
- User A **pushes** the new version to the main repository



- User B **commits** a new version to their local repository
- User B tries to **push** the new version to the main repository, but it fails indicating he needs to update the local version first
- User B **fetches** the latest version of file from the main repository
- User B **merges** changes into his local version of file
- User B **commits** the combined changes into his local repository
- User B **pushes** the merged version of file to the main repository

Deltified storage vs. Snapshot storage

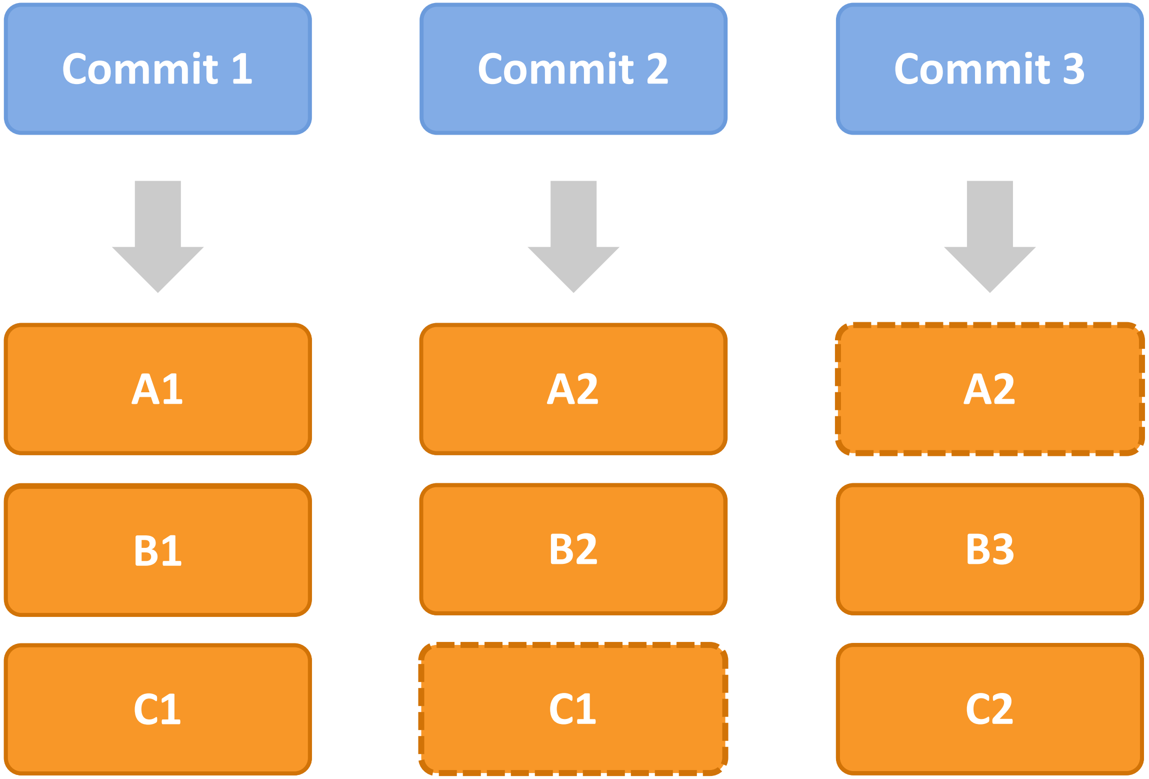
Two ways to store your changes in a repository:

- **Deltified storage** - encoding the representation of a chunk of data as a collection of differences against some other chunk of data
- **Snapshot storage** - stores the complete files changed by a commit along with references to files that were not changed by that commit

Deltified storage
Snapshot storage

Deltas to the previous version are stored

Each version holds the full source tree

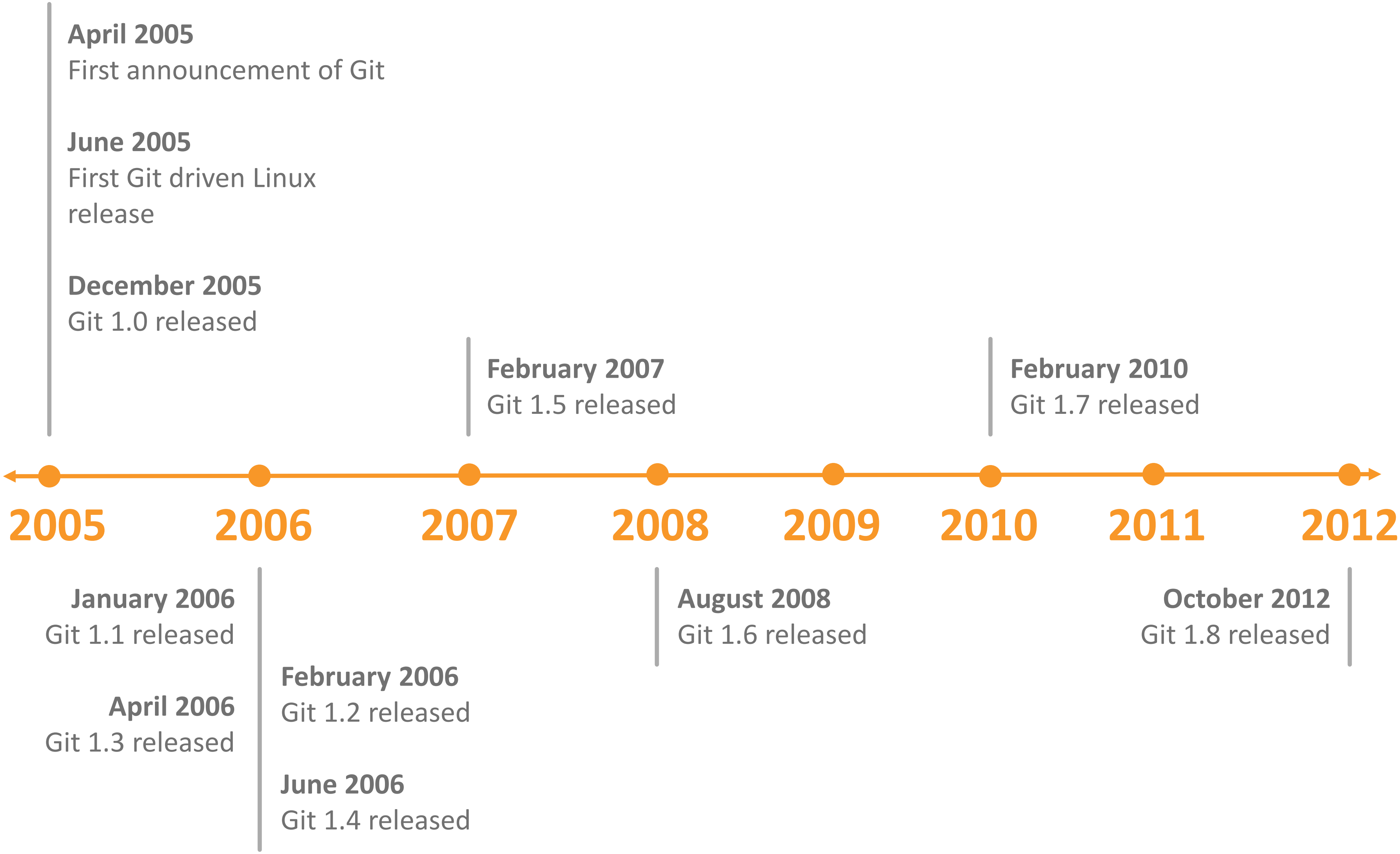


Git

Git is a distributed version control system:

- Inspired by BitKeeper and Monotone
- GPLv2
- Initiated by Linus Torvalds (father of Linux)
- Strongly influenced by Linux kernel development initially and Android development more recently
- Strong support for non-linear development
- support for `ssh`, `http(s)`, and `git` protocols
- Simple design
- Potentially complex tool to master

Git History



April 2005
First announcement of Git

June 2005
First Git driven Linux release

December 2005
Git 1.0 released

January 2006
Git 1.1 released

February 2006
Git 1.2 released

April 2006
Git 1.3 released

June 2006
Git 1.4 released

February 2007
Git 1.5 released

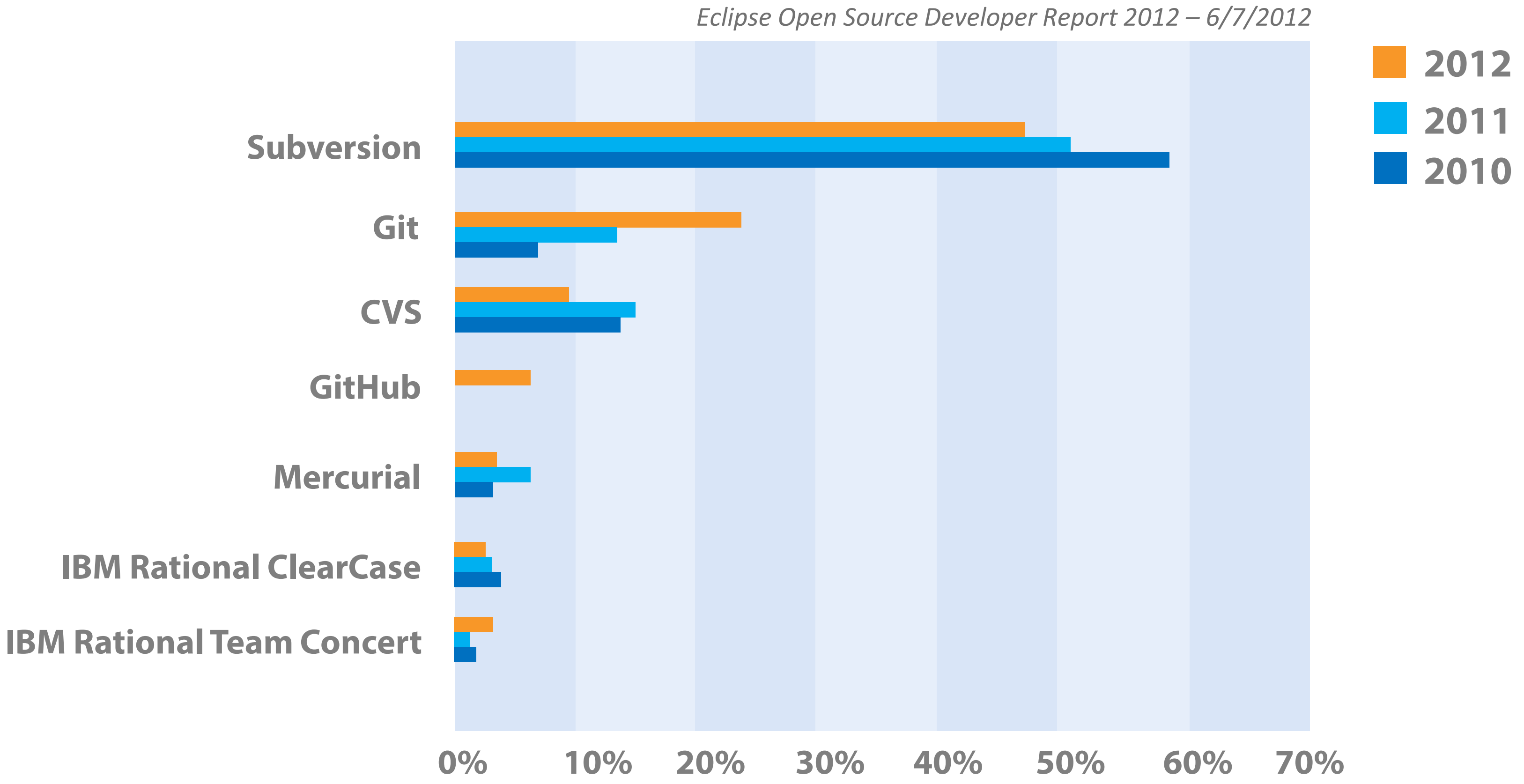
August 2008
Git 1.6 released

February 2010
Git 1.7 released

October 2012
Git 1.8 released

Git Growth Chart

What is the primary source code management system you typically use? (Choose one.)



- Git increased to 27% from 13% clearly showing momentum
- Subversion decreased in 2012 but still #1 SCM

Popular projects using Git

GNOME
http://www.gnome.org

Django
https://www.djangoproject.com

Android
http://android.git.kernel.org

Ruby on Rails
http://rubyonrails.org

Linux Kernel
http://kernel.org

TYPO3
http://typo3.org

GIT Projects
http://git.wiki.kernel.org/index.php/GitProjects

Terminology

Term	Description
repository	Copy of your project with full history
remote	A remote repository on another computer
.git	Directory where repository metadata (references, object store, etc.) is stored
path	Location of a file/directory in your repository or working tree
index	Staging area to assemble the next commit
working tree	Files and directories you are working on
SHA-1	Hash algorithm used by Git to identify objects like commits
object	General term for all object types used by Git, identified by SHA-1 hash
<type>-ish	An object of the type or which can be peeled to the corresponding type

Terminology (Cont'd)

Term	Description
commit	Object representing a snapshot of your working tree at a certain point in time
blob	Object representing a file
tree	Object representing a directory
tag	Marks a specific object
lightweight tag	Like an immutable branch, just a file
annotated tag	Tag object
branch	A line of development
master	By convention, name of the main branch
HEAD	Pointer to the currently checked out branch

Key Features

- Distributed development
- Speed
- Commit early – commit often
- Strong support for non-linear development
- Easy merging with multiple strategies
- Simple object model
- Staging area
- Cryptographic authentication of history
- Efficient object storage

Clients & Platforms

- Graphical Clients:

  gitgui (Linux, Mac, Windows)

  msysgit (Windows)

  gitk (Linux, Mac, Windows)

  TortoiseGit (Windows)

  GitX (Mac)

  SmartGit (Linux, Mac, Windows)

  Tower (Mac)

  Git Extensions (Windows)

  Gitbox (Mac)

  Git-cola (Linux, Mac, Windows)

  GitHub for Windows

  GitHub for Mac

- IDE-Integration

  Git Eclipse Client

  Xcode

- Web Interfaces:

  cgit

  gitweb

- Web applications:

  Gerrit

  GitLab

  GitHub

Outline

- Git Basics
- Standard Work Cycles
  - Working with Git Locally
  - Working with Git Remotely
- Branching
- Checkout vs. Reset
- Examining full history
- .gitignore
- Where to get help

Git for Developers

Essential Concepts 1

Git Basics

The Git repository

- Essential parts of a Git repository
  - Repository metadata (.git)
    - `refs/`
    - `objects/`
    - `index`
    - `HEAD`
  - Working tree

Git repository structure (.git)

```
$ ls -al
total 11
drwxr-xr-x   7 sheta   Administ   4096 Dec  3 15:17 .
drwxr-xr-x   3 sheta   Administ   4096 Nov 30 11:26 ..
-rw-r--r--   1 sheta   Administ     23 Nov 30 11:09 HEAD
-rw-r--r--   1 sheta   Administ    363 Nov 30 11:46 config
-rw-r--r--   1 sheta   Administ     73 Nov 29 17:03 description
drwxr-xr-x   2 sheta   Administ   4096 Nov 29 17:03 hooks
-rw-r--r--   1 sheta   Administ     32 Nov 30 11:26 index
drwxr-xr-x   2 sheta   Administ      0 Nov 29 17:03 info
drwxr-xr-x   3 sheta   Administ      0 Nov 29 17:03 logs
drwxr-xr-x  25 sheta   Administ   4096 Nov 30 11:08 objects
-rw-r--r--   1 sheta   Administ     94 Nov 29 17:03 packed-refs
drwxr-xr-x   5 sheta   Administ      0 Nov 30 11:07 refs
```

```
├── .git
│   ├── ...
│   ├── HEAD
│   ├── config
│   ├── descritpion
│   ├── hooks
│   ├── index
│   ├── info
│   ├── logs
│   ├── objects
│   ├── packed-refs
│   └── refs
├── Makefile
├── README.txt
└── src
    └── watchdog.c
```

HEAD	This file holds a reference to the branch you are currently on
config	This is the main Git configuration file. It keeps specific Git options for your project
descritpion	It will show when you have viewed your repository or the list of all versioned repositories.
hooks	Directory contains shell scripts that are invoked after the corresponding Git commands
index	The Git index is used as a staging area between your working directory and your repository
info	Contains additional information about the repository.
logs	Keeps records of changes made to refs.
objects	In this directory the data of your Git objects are stored – all the contents of the files you have ever checked in
packed-refs	The file consists of packed heads and tags.
refs	This directory normally contains three subfolders – heads, remotes and tags

Types of repositories

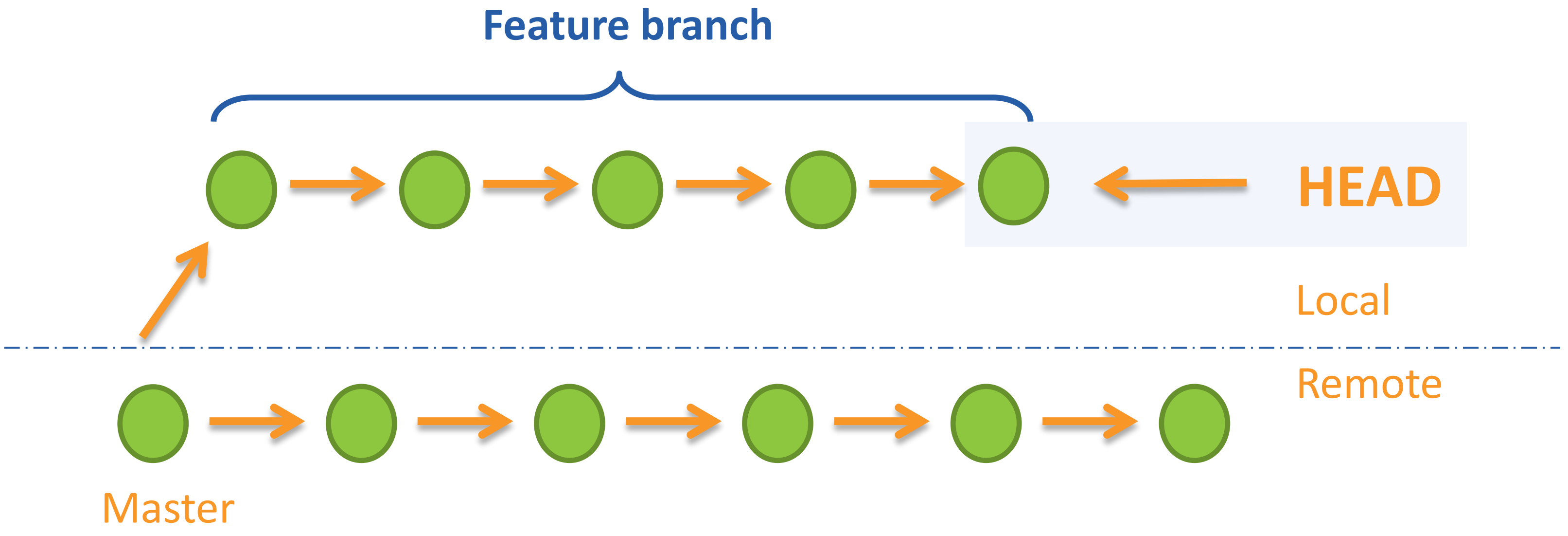
- **Bare**, used for hosting, code exchange, etc.
  - No working tree
  - No index

- **Non-bare**, used by developers and includes:
  - refs/
  - objects/
  - index
  - HEAD

The HEAD

- HEAD is a 'pointer' to the tip of the currently checked out branch
  - In a *detached HEAD* state, HEAD points directly to a commit

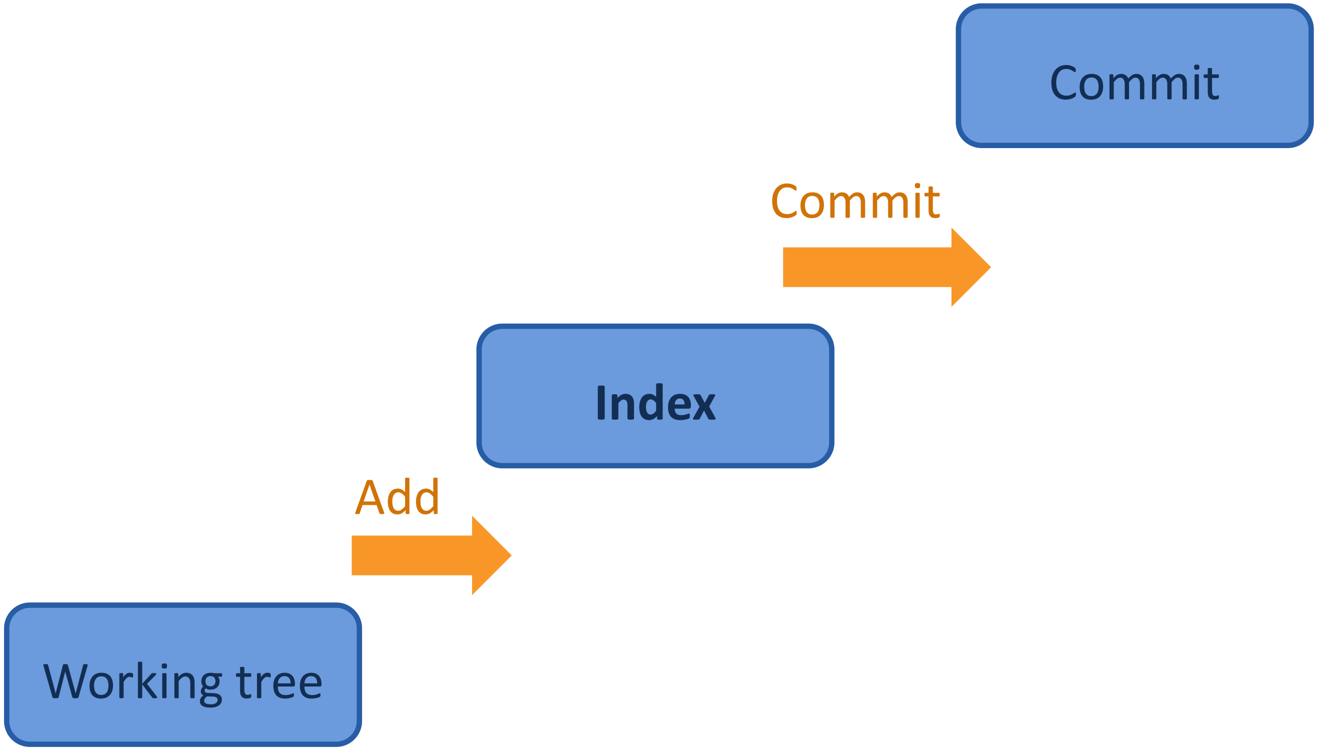
- Only one HEAD per repository



The Index

- Also called ***staging*** area or ***cache***

- Used to 'compose' the next commit
  - Powerful and important feature of Git



The Index (Cont'd)

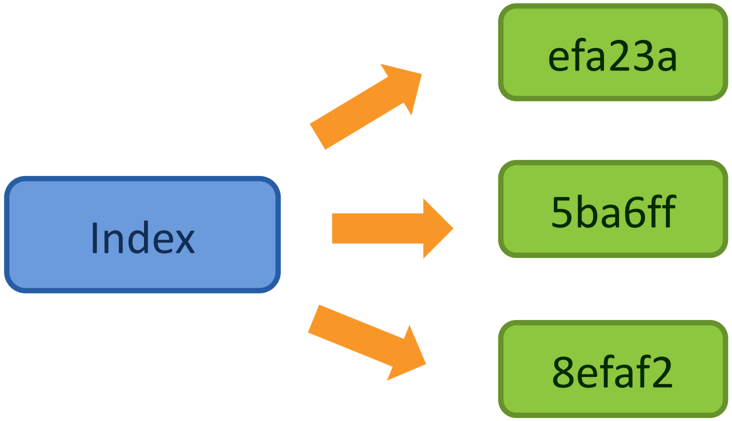
- A repository can hold multiple versions of a file as found in the:
  - Last commit or earlier in history
  - Index
  - Working tree

- A file is added to the object database when you stage it

- You can make multiple commits to your local repository before pushing to a remote repository

The Index (Cont'd)

- Think of the index as a virtual working tree, tracking:
  - Permissions
  - SHA1 of *blob* object
  - Current stage (important for merging)
  - Path (e.g. doc/install.txt)



- It is a one level undo

Mode bits	Object ID	Stage number	Name (Path)
100644	efa23a...	0	doc/readme.txt
...			

The Index (Cont'd)

`git ls-files --stage` shows the currently staged files

```
$ git ls-files --stage
100644 f9264f7fbd31ae7a18b7931ed8946fb0aebb0af3 0    README.txt
100644 e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 0    foo/bar.txt
```

NOTE

It shows all tracked files in their current state.

The working tree

- Working tree has all files and folders as found in your HEAD, plus the changes you made since your last commit
- There is only ONE working tree per repository (and only 1 .git folder as well)

```
├── .git
│   ├── ...
│   ├── HEAD
│   ├── index
│   ├── objects
│   └── refs
├── Makefile
├── README.txt
└── src
    └── watchdog.c
```

Revisioning

Git revisions are SHA1 hashes of commits, not revision numbers

- A commit includes
  - The hash of the root tree
  - The hash of the parent commit(s)
  - Commit message
  - Author
  - Committer

```
$ git log -1
commit 5cf2b7013b1504c1a5e09e363e538c7bea82bf06
Author: Alice <alice@collab.net>
Date:   Thu Dec 6 18:11:19 2012 +0100

    README: fix typo
```

Revisioning

- Example – `cat-file` command shows the content of an object:

```
$ git cat-file -p HEAD
tree 4ddabe2b65a5f7529e556b36c18db308227e7092
parent 494e2cb73ed6424b27f9766bf8a2cb29770a1e7e
author alice <alice@collab.net> 1354809881 +0100
committer alice <alice@collab.net> 1354809881 +0100

Added jGit submodule
```

- Given a commit hash, we can verify both the full tree and the full history, since we have
  - The hash of the tree (which references its subtrees)
  - The hash of the parent (which references its parents)

Why two user fields?

- Why both an author and a committer fields?
  - Allows developers and maintainers to preserve authorship

- By default, both author and committer are set to the configured user name and email

Author

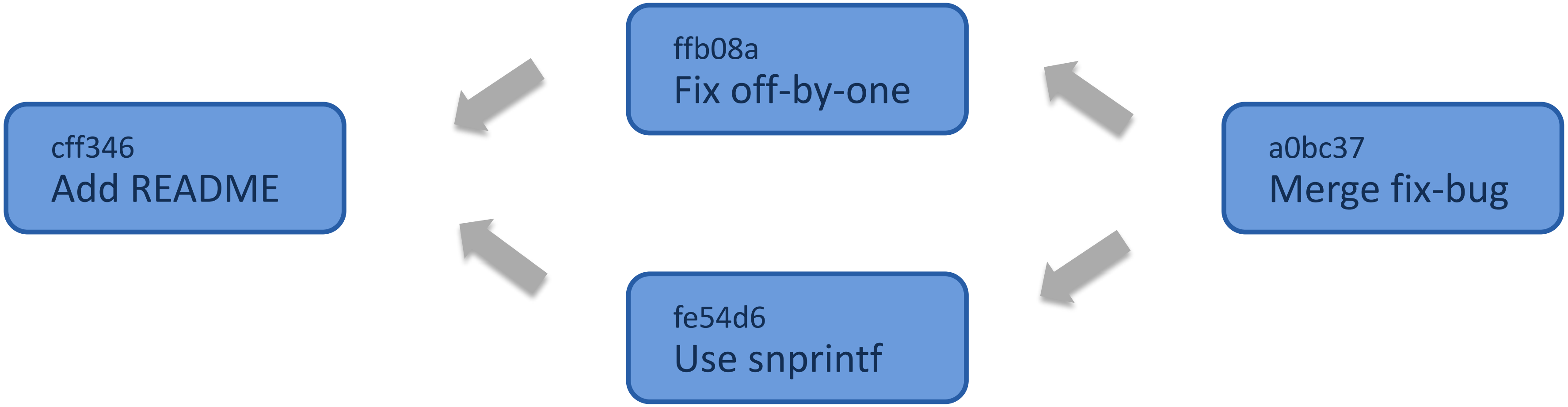
Commiter

References

- Basic concept of references: a ***pointer*** to a Git object
- Usually a file in .git/refs/... points at an object –
  - Branches
  - Tags
  - Notes
  - ...

Git history

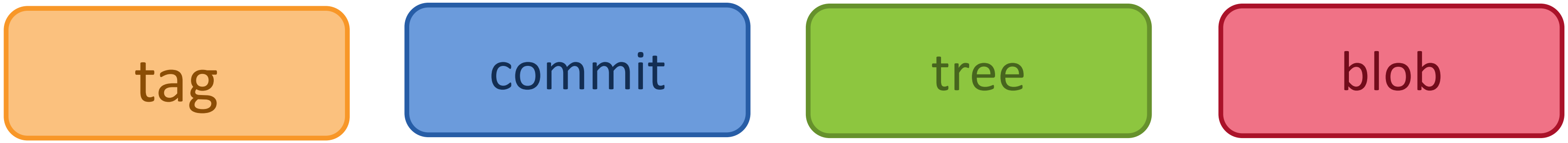
- Git follows a snaphot model
  - Each commit is a snapshot of a given state
- A line of ***snapshots*** builds a directed acyclic graph called ***DAG*** (remember: each commit refers to its parents)



Git objects

There are four different object types

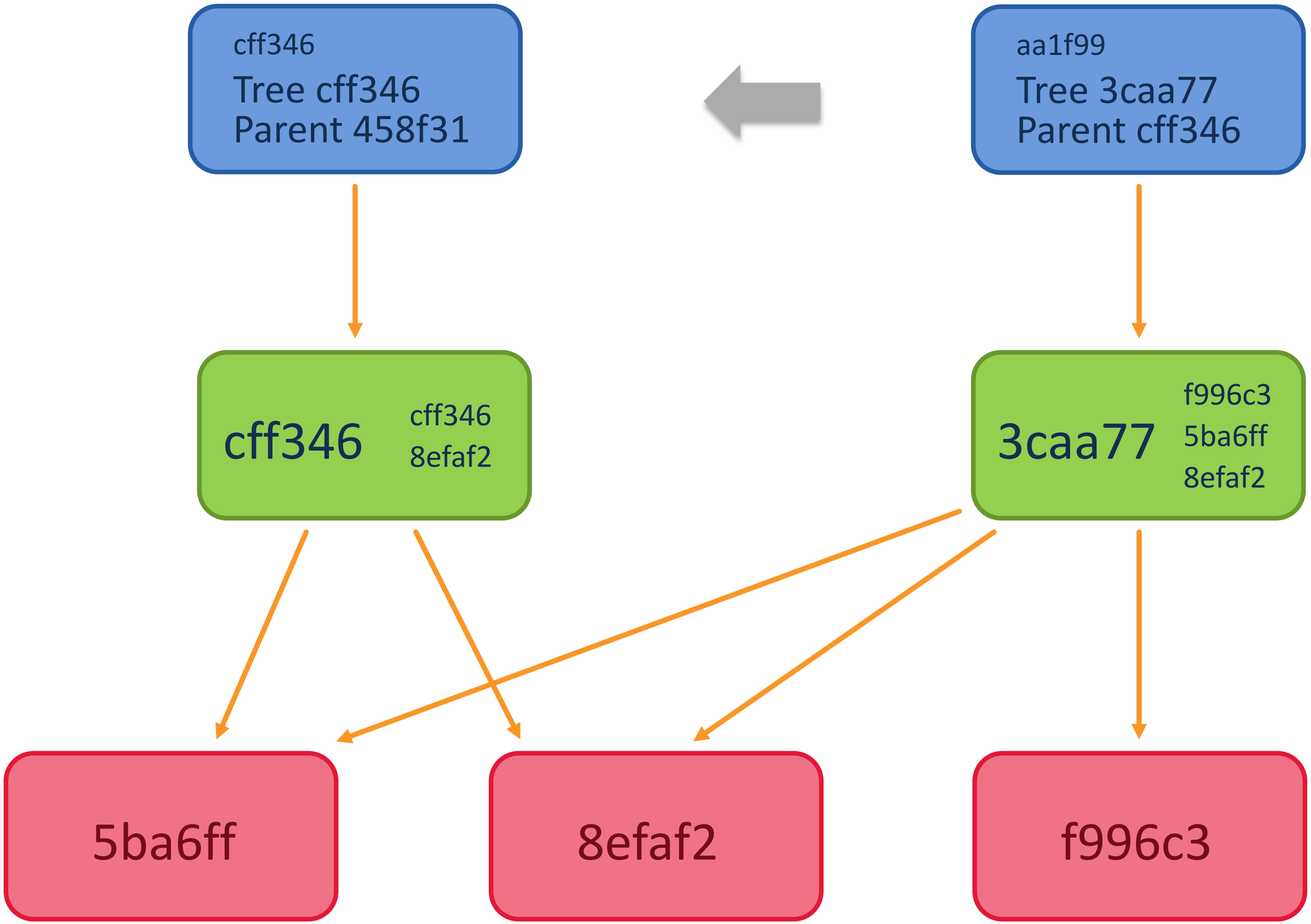
- **Annotated tag:** a specific ***named*** pointer to a commit in history
- **Commit:** a snapshot
- **Tree:** representation of a directory
- **Blob:** representation of a file (i.e., its content)


tag commit tree blob

Git objects (Cont'd)

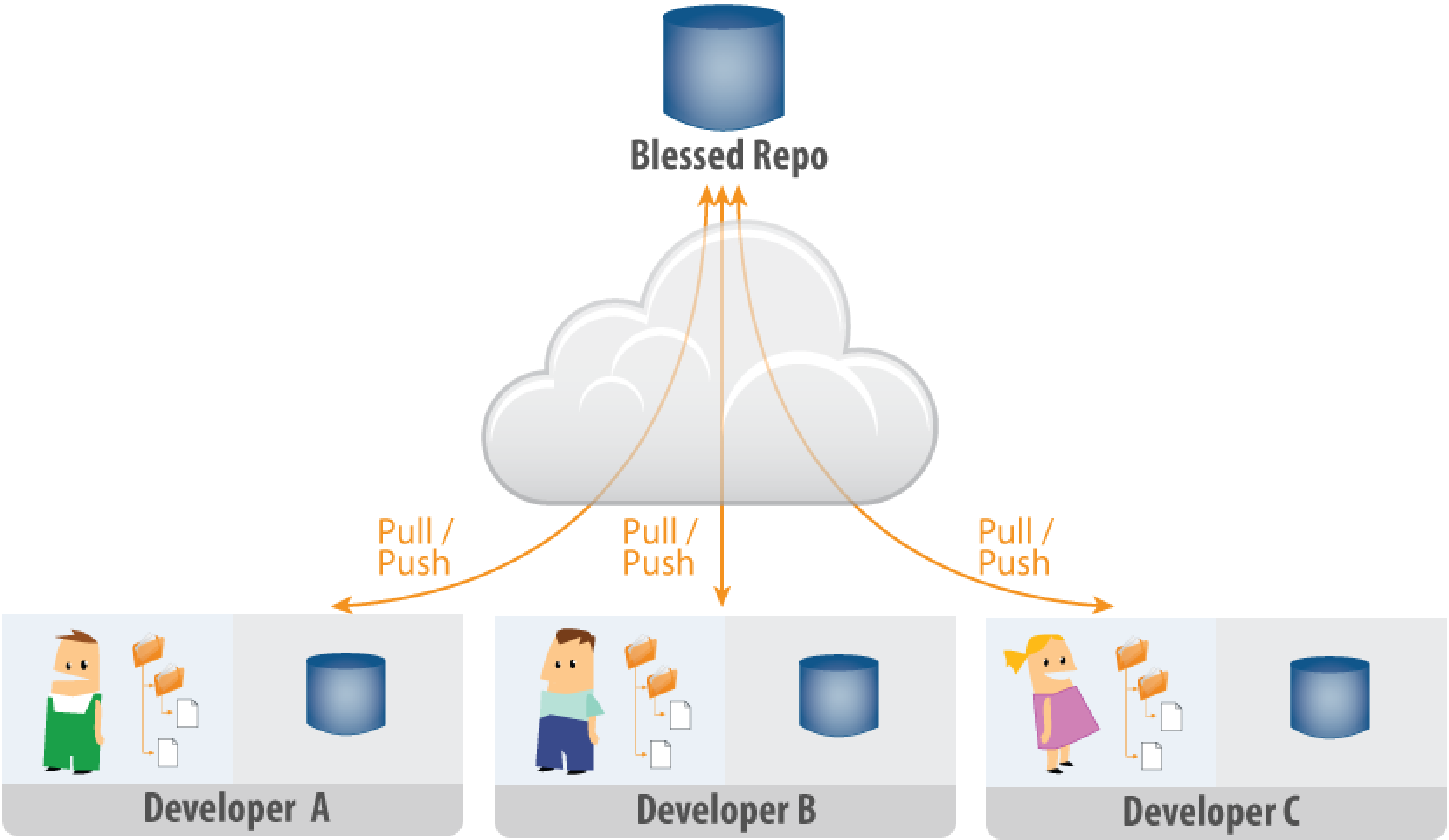
- Git tracks content, not files
  - Two files with the same content result in only one ***blob*** in the ***object database (ODB)***
  - Two identical subtrees result in only one tree object
  - As a result there is very little duplication in the ODB
- You cannot track empty folders with Git

Git objects (Cont'd)

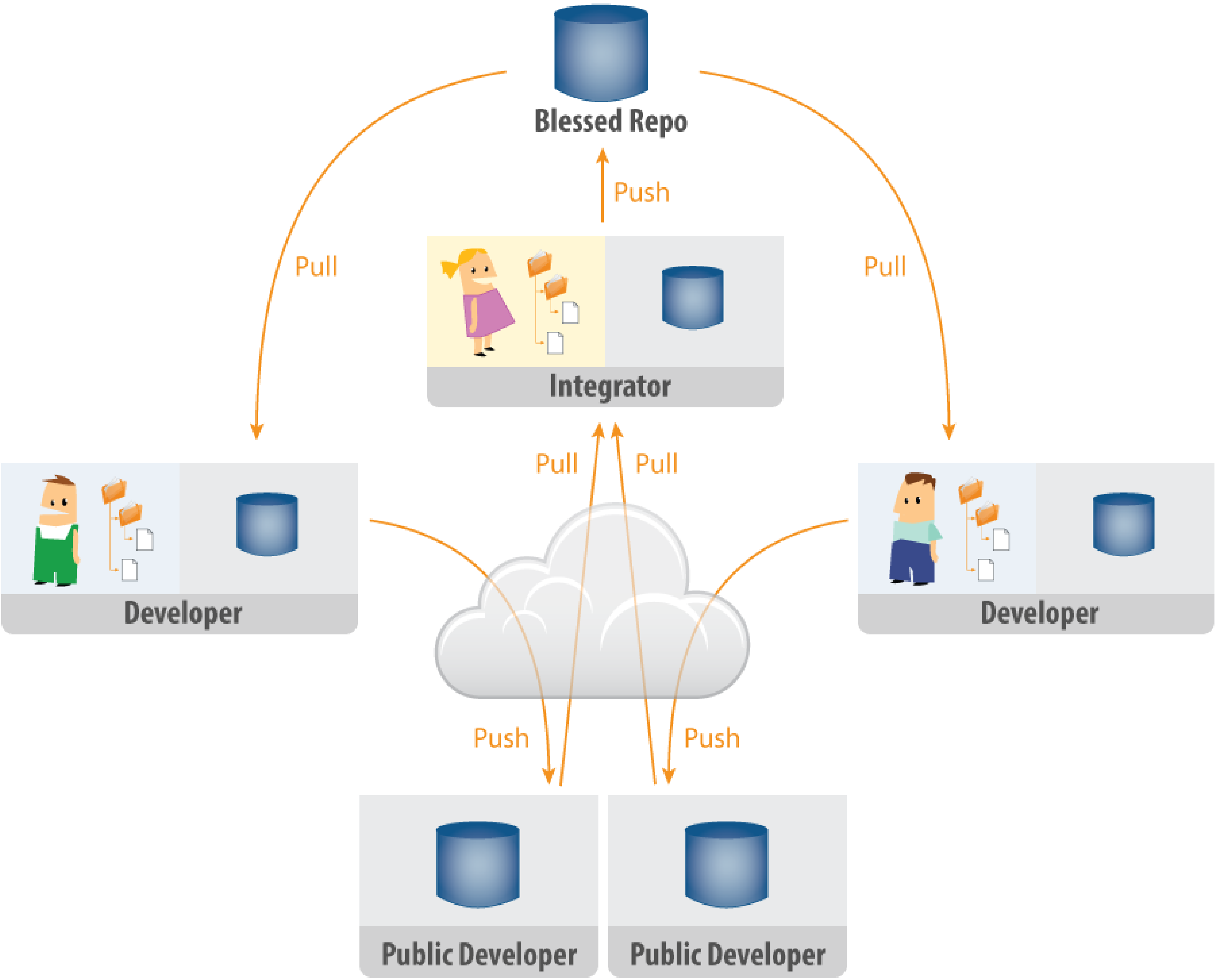


Common Team Workflows

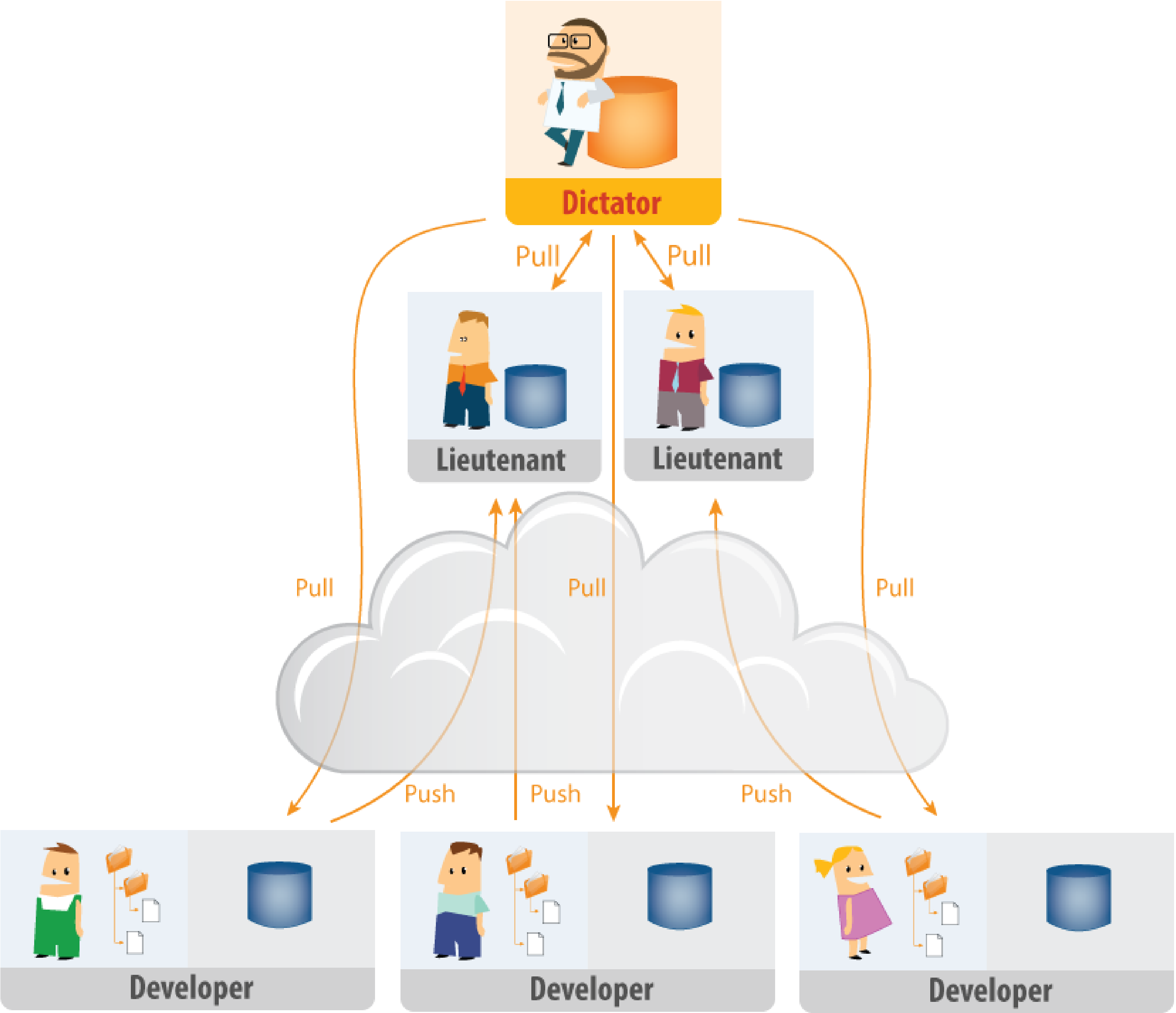
Centralized Workflow



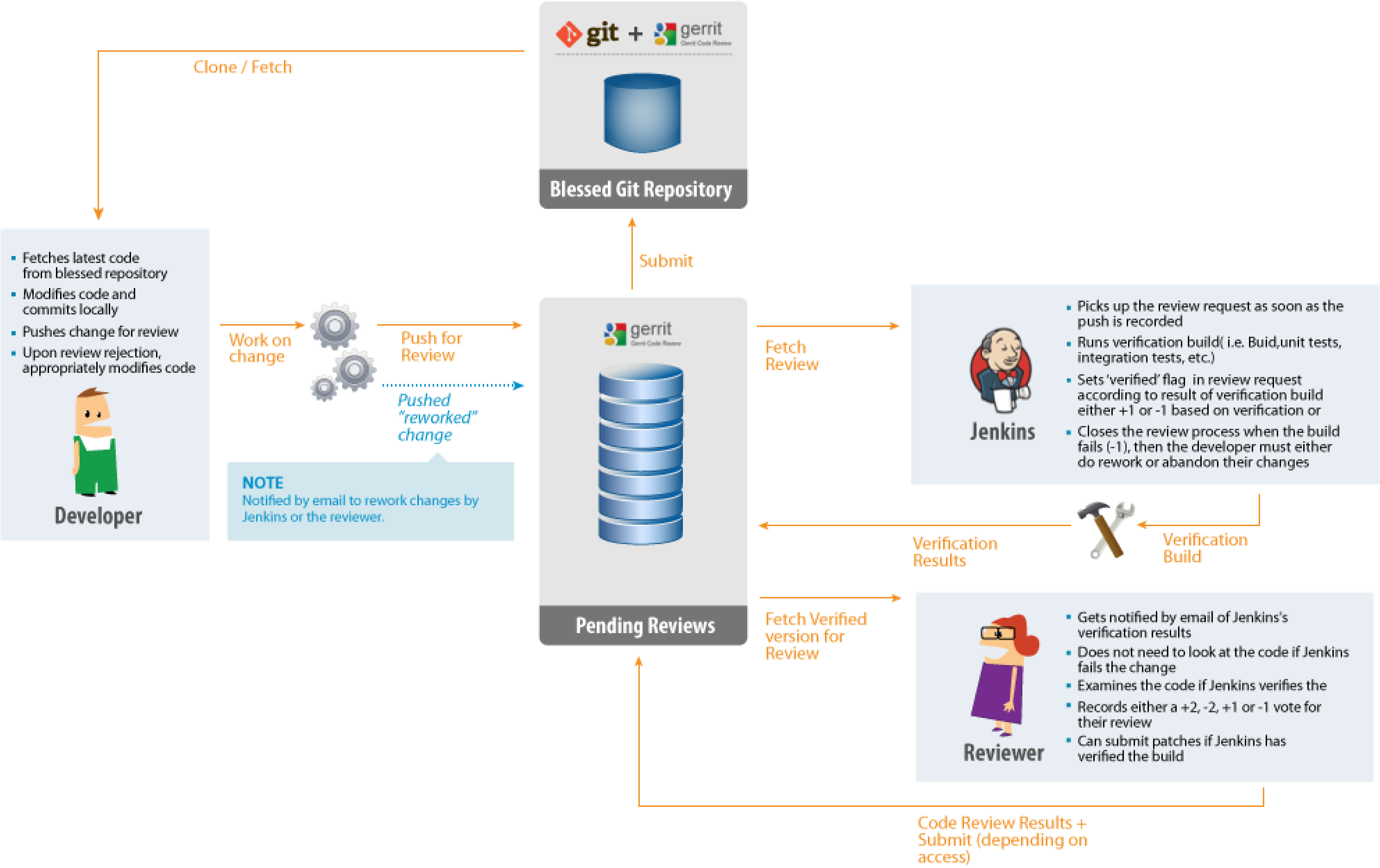
Integrators Workflow



Dictator / Lieutenants Workflow



Gerrit Code Review Workflow



Developer

- Fetches latest code from blessed repository
- Modifies code and commits locally
- Pushes change for review
- Upon review rejection, appropriately modifies code

NOTE
Notified by email to rework changes by Jenkins or the reviewer.

Jenkins

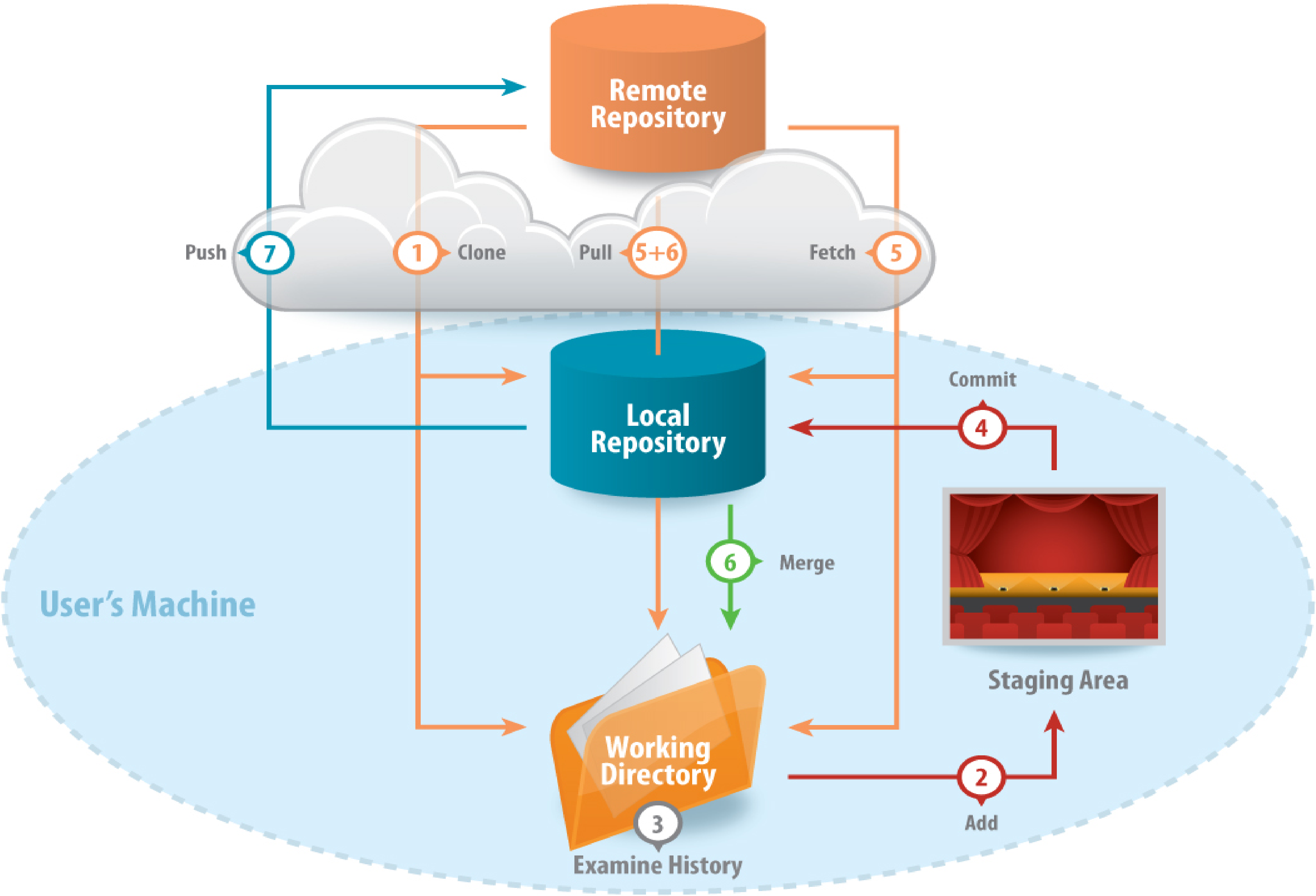
- Picks up the review request as soon as the push is recorded
- Runs verification build(i.e. Buid,unit tests, integration tests, etc.)
- Sets 'verified' flag in review request according to result of verification build either +1 or -1 based on verification or
- Closes the review process when the build fails (-1), then the developer must either do rework or abandon their changes

Reviewer

- Gets notified by email of Jenkins's verification results
- Does not need to look at the code if Jenkins fails the change
- Examines the code if Jenkins verifies the
- Records either a +2, -2, +1 or -1 vote for their review
- Can submit patches if Jenkins has verified the build

Standard Git Work Cycles

Overall work cycle



Git config

- With `git config` you can customize how git behaves

- Information is stored as hierarchical key-value pairs. Git has several different configuration files:
  - System wide /etc/gitconfig
  - Specific user ~/.gitconfig
  - Repository .git/config

Initial configuration

- At a minimum, you need to configure your name, email address, and editor
- By default, git modifies the repository config file
- To modify the user config file, use `--global`

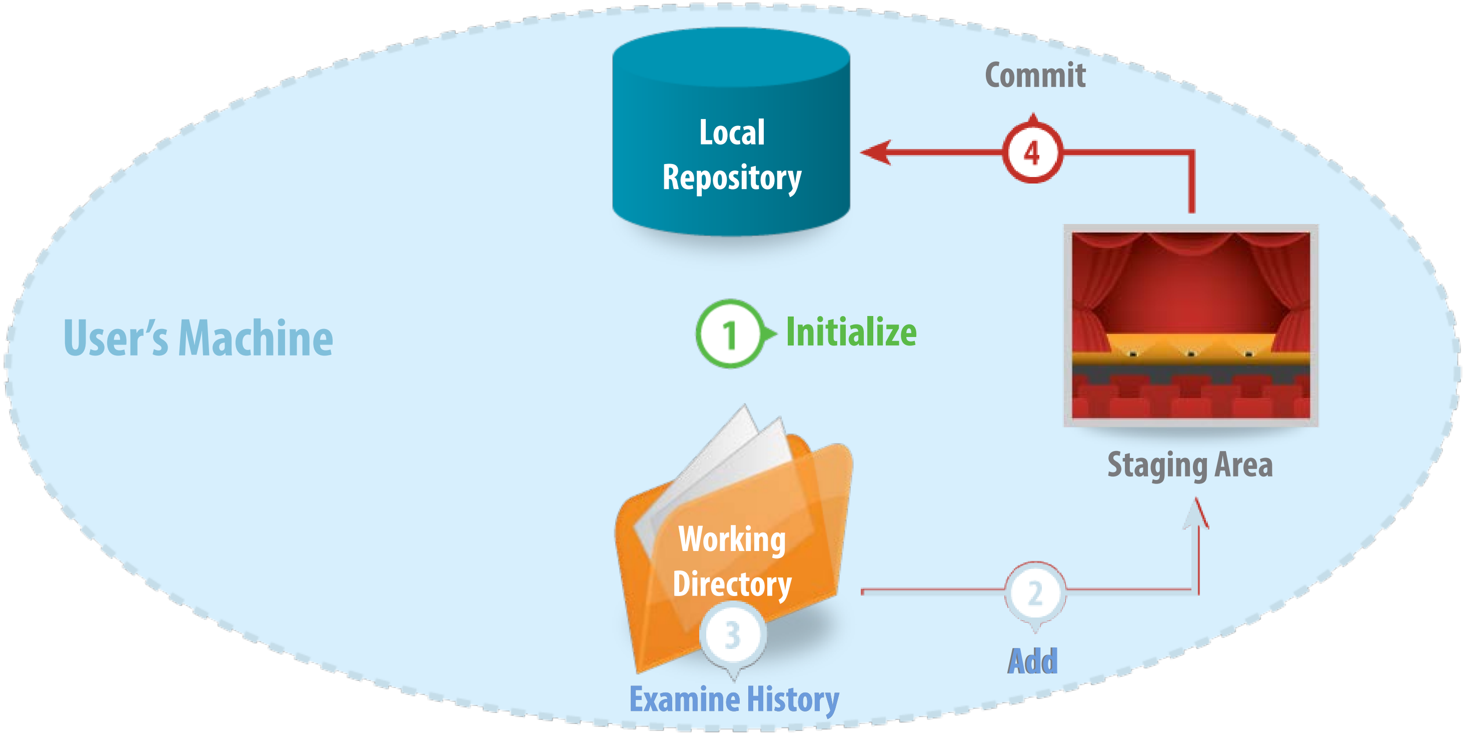
```
$ git config --global user.email 'alice@collab.net'

$ git config --global user.email 'Alice‘

$ git config core.editor gedit
```

- `'--global'` options can be overwritten by the 'local'/'repository' configuration

Local Git Work Cycle

Local work operations

```
git init my-project
git add file1 (git mv file2 file 4    git rm file3)
git status (git diff)
git commit -m  “Added file1”
```



Create a local repository

git init my-project

- Creates directory `my-project`
- Initializes repository metadata in `my-project/.git`
  - `index`
  - `HEAD`
  - `config`
  - `etc.`

```
$ git init my-project
Initialized empty Git repository in C:/Users/sheta/my-project/.git/
```

Standard local work cycle

1. Make your changes

2. Examine your changes

3. Commit your changes

Step 1: Make your changes

- Create and modify paths normally
- Stage your modifications
  - Create or modify

    `git add <file>`

```
$ git add README.txt
```

  - Move or rename

    `git mv <file> <file>`

```
$ git mv foo.txt bar.txt
```

  - Remove

    `git rm <file>`

```
$ git rm bar.txt
```

Step 2: Examine your changes

Verify the status of changes with `git status`:

- Gives you detailed information about what is going on
- Provides help on how to undo changes or how to continue

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#     new file:   README.txt
#
```

Git diff

- Examine your changes using `git diff` for unstaged changes

```
$ git diff
diff --git a/README.txt b/README.txt
index e84566..1b7c705 100644
--- a/README.txt
+++ b/README.txt
@@ -1 +1,2 @@
 README
+adding third line
```

- And `git diff --cached` to examine your staged changes

```
$git diff --cached
diff --git a/README.txt b/README.txt
new file mode 100644
index 0000000..e845566
--- /dev/null
+++ b/README.txt
@@ -0,0 +1 @@
+README
```

- You can also compare the HEAD of your local branch to the HEAD of the remote branch

```
$git diff master origin/master
diff --git a/README.txt b/README.txt
index bd5f0ef..e845566 100644
--- a/README.txt
+++ b/README.txt
@@ -1,2 +1 @@
README
-Added something in README locally
```

Step 3: Commit your changes

```
git commit -m 'Add README file',
```

```
$ git commit -m 'Added README file'
[master (root-commit) 9fbed8a] Added README file
 1 files changed, 1 insertions(+), 0 deletions (-)
create mode 100644 README.txt
```

- Creates a new commit object based on the index
- Moves current branch to the new commit
- New tree and blob objects are created

```
$ git cat-file -p 9fbed8aff
tree f79cbae241a835d49f493b38f564424058768013
parent 494e2cb73ed6424b27f9766bf8s2cb29770a1e7e
author alice <alice@collab.net> 1355307007 +0100
committer alice <alice@collab.net> 1355307007 +0100
```

Commit amend

`git commit --amend` allows you to ***rework last commit*** (modify commit message, add/remove changes etc..) and commit back to your repository.

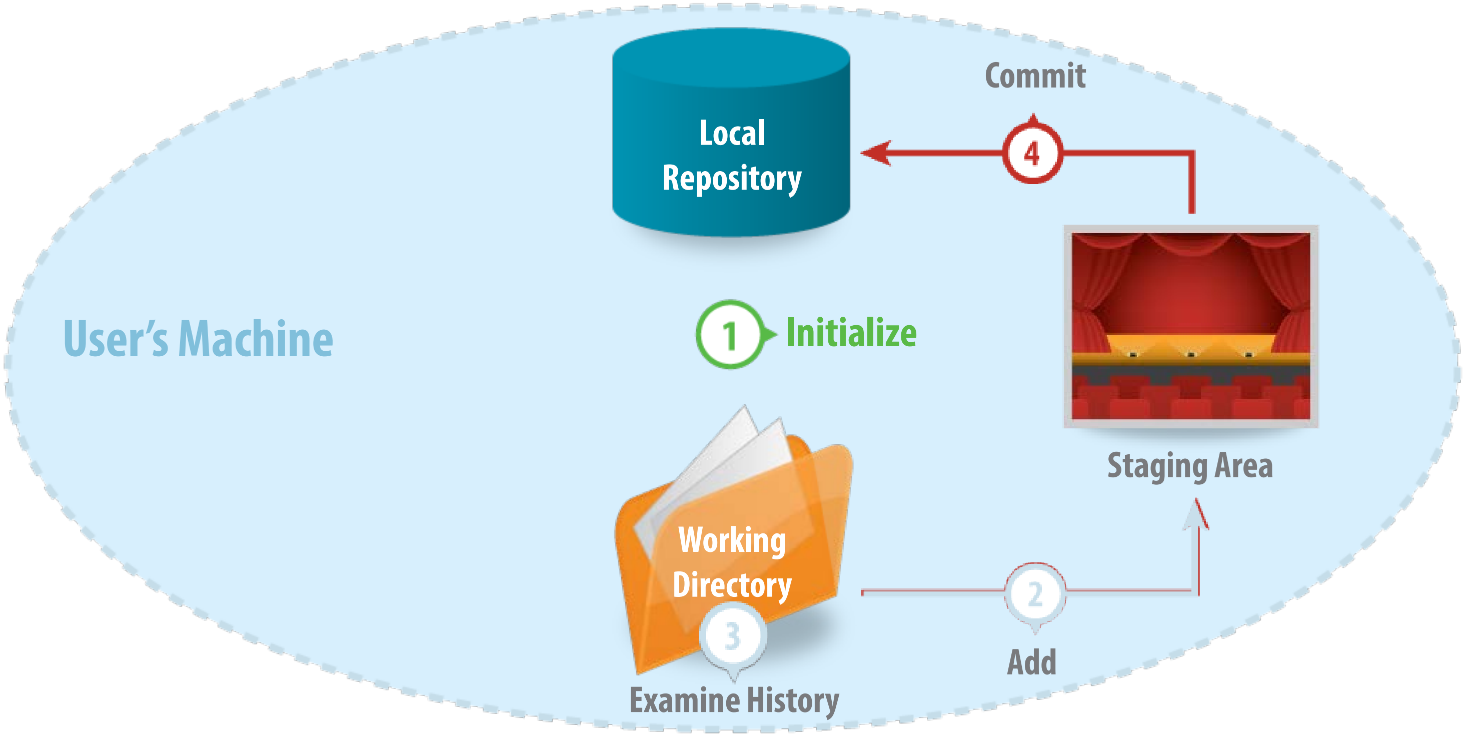
```
$ git commit --amend
```

```
Added README file with additional info
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD^1 <file>..." to unstage)
#
#       new file:   README.txt
#
```

```
[master 85b32a4] Added README file with aditional info
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 README.txt
```

Local work operations

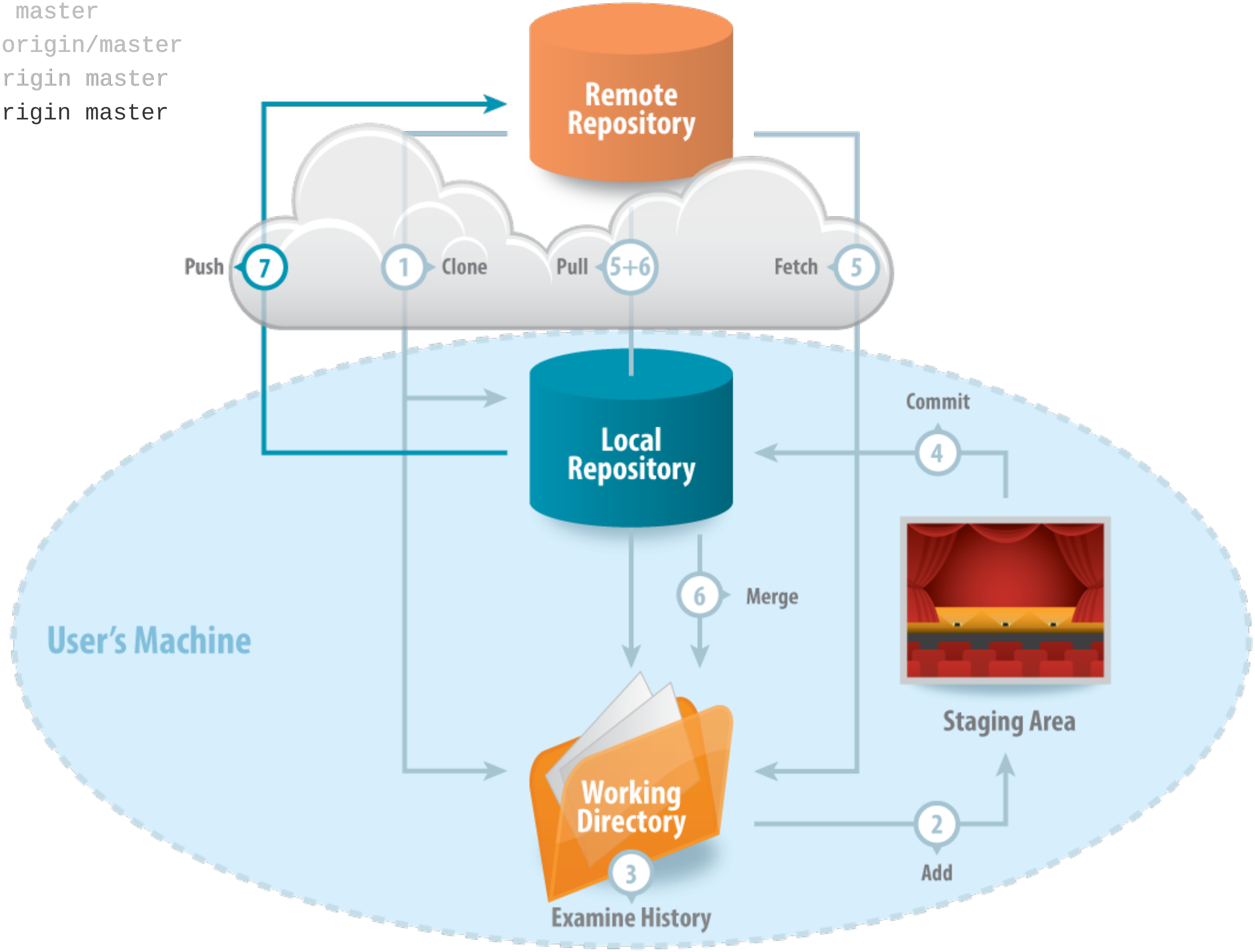
```
git init my-project
git add file1 (git mv file2 file 4    git rm file3)
git status (git diff)
git commit -m  “Added file1”
```



Remote Git Work Cycle

Remote work operations

```
git clone ssh://serverpath/my-project
git add file1 (git mv file2 file4   git rm file3)
git origin master
git merge origin/master
git pull origin master
git push origin master
```



Authentication

Generate SSH key pair for authentication

```
$ ssh-keygen -t rsa -C "alice@collab.net"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/sheta/ .ssh/id_rsa):
Created directory '/c/Users/sheta/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/sheta/.ssh/id_rsa.
Your public key has been saved in /c/Users/sheta/.ssh/id_rsa.pub.
The key fingerprint is:
db:0b:67:c6:a0:69:e4:26:75:17:cb:81:13:53:6c:f8 alice@collab.net
```

Cloning remote repository

`git clone <url>` to clone an existing repository

```
$ git clone ssh://alice@gate.collabnet.medienstadt.net:29418/my-project
Cloning into 'my-project'...
remote: Counting objects: 2, done
remote: Finding sources: 100% (2/2)
remote: Total 2 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (2/2), 186 bytes, done.
```

Standard remote work cycle

1. Fetch changes from remote

2. Merge changes

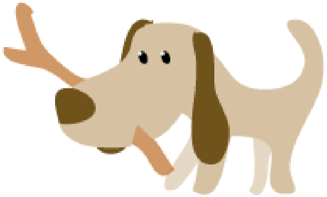
3. Push your change

Alternative: ***Pull*** *changes from remote (****Fetch*** *+* ***Merge****)*

Step 1: Fetch changes from remote repository

To receive others‘ changes from remote:

– git fetch



Gets all changes from the remote ‘origin‘ branch (if specified) and makes them available locally, but does not place them into the working tree in order to allow you to evaluate the changes before having them applied

```
sheta@SHETA-THINK ~/my-project (master)
$ git fetch origin master
From ssh: //gate.collabnet.medienstadt.net:29418/my-project
 * branch             master     -> FETCH_HEAD
```

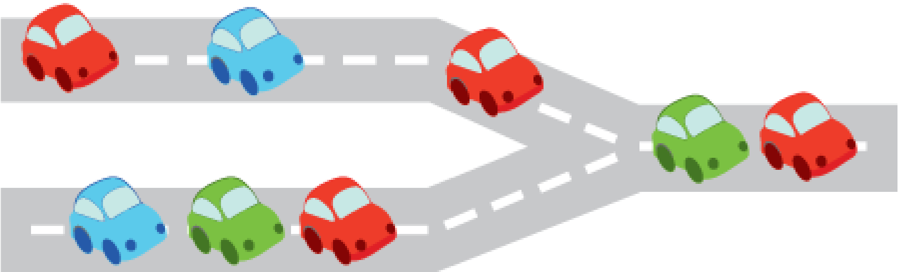
Step 2: Merge the fetched changes

To merge others‘ changes from remote into your work:

- `git merge origin/master`

If you are happy with the fetched changes, then you can merge them

```
$ git merge origin/master
Updating 4445682..494e2cb
Fast-forward
 README.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 README.txt
```



Alternative to steps 1+2: Pull changes from remote

If you are comfortable with immediately getting the changes applied to your working directory (instead of exectuting two distinct operations), you can use the pull operation

- `git pull`

  Gets all changes from the remote origin and merges them into your working directory in single operation (equivalent to git fetch+merge)

```
$ git pull origin master
From ssh://gate.collabnet.medienstadt.net:29418/my-project
 * branch            master     -> FETCH_HEAD
Updating 4445682..494e2cb
Fast-forward
 README.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 README.txt
```

Bare repositories

- Do not have a working directory

- Only used for fetching and pushing

  Pushing to ***non-bare*** repoitories is discouraged, since you may push to a checked-out branch

- `git init --bare my-project.git` creates a bare repository

- By convention, bare repositories have a suffix `.git`

Step 3: Push your changes to the remote repository

- To publish your changes, you simply push them to a remote repository*

```
$ git push origin master
Counting objects: 4, done.
Writing objects: 100% (3/3), 251 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://alice@gate.collabnet.medienstadt.net:29418/my-project
   4445682..494e2cb  master -> master
```

- If you cloned a repository, remote is already set.
- If you want to clone from one repository, but you would like to push to another, then you can add remote to your configuration

```
$ git remote add origin ssh://alice@gate.collabnet.medienstadt.net:29418/my-project
```

NOTE

If you have one remote repository, “origin“ is a default convention. In the case where you want to “push” to another remote repository, you have to explicitly specify the remote repository’s name as parameter to the push command

```
$ git push production master
```

Remotes

- A ***remote*** is another Git repository your repository 'knows' about

  Each remote is an entry in the config file of the repository

- Supported ***transports***
  - `ssh`
  - `http(s)`
  - `git`

Remotes (Cont'd)

A remote defines:

- Where to fetch from
- What to fetch
- A *fetchspec*
- Optionally: what URL should be used when pushing

```
$cat .git/config
...
[remote "upstream"]
       url = https://github.com/libgit2/libgit2.git
       fetch = +refs/heads/*:refs/remotes/upstream/*
       pushurl = https://alice@github.com/libgit2/libgit2.git
```

Git push

If you have cloned an existing repository, you can just execute a straight `git push`, since your branch is already ***tracking*** the remote branch

```
$ git remote add my-remote https://exmaple.com/project.git
$ git push my-remote master
Counting objects: 3, done.
Writing objects: 100% (3/3), 224 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To /tmp/remote
 * [new branch]      master -> master
```

What are tracking and remote tracking branches?

- The combination of these branches defines a relationship between a local branch and one in the remote repository. A relationship understood by Git internally.
- When a repository is cloned, Git automatically creates **remote tracking branches** (e.g., origin/master) for the remote branches and a **tracking branch** (e.g., master) to allow for local changes in relationship to the remote branch.
- **Remote tracking branches** should be seen as read-only branches with no local changes made directly to them.
- **Tracking branches** should be seen as the place where changes are made locally and where remote changes are merged with local changes.

Tracking and remote tracking branches, (Cont‘d)

- When pushing the tracking branch, it automatically pushes to the remote branch associated with it (e.g., your local master publishes to master on the remote)

  It also updates **the head** of your remote tracking branch to reflect the current state of the remote branch.

- When fetching the tracking branch, it automatically fetches from the remote branch (i.e., remote master is fetched to origin/master locally)

- The established relationship makes it easy to merge changes created by others with those created locally.

- You can create a tracking branch manually by using the ‘--track‘ option on the branch command.

Tracking branch example

- You create a local branch based on remote branch by:

```
$ git checkout -b myMaster origin/master
Branch myMaster set up to track remote branch master from origin.
Switched to a new branch 'myMaster'
```

  - This local branch is called a **tracking branch**

- Now that we have local branch, we might add a new commit:

```
sheta@SHETA-THINK ~/my-project (myMaster)
$ git commit -m "new sample file added"
[myMaster 850115f] new sample file added
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 bar.txt
```

- We could then update the local with changes from the remote:

```
$ git fetch origin
```

- If we now check status :

```
$ git status
# On branch my-Master
# Your branch and 'origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
#
nothing to commit (working directory clean)
```

  - It shows we have one new commit in our remote branch which is not yet available in the local branch. So you have to merge /rebase before pushing the newly created commit to the remote branch.

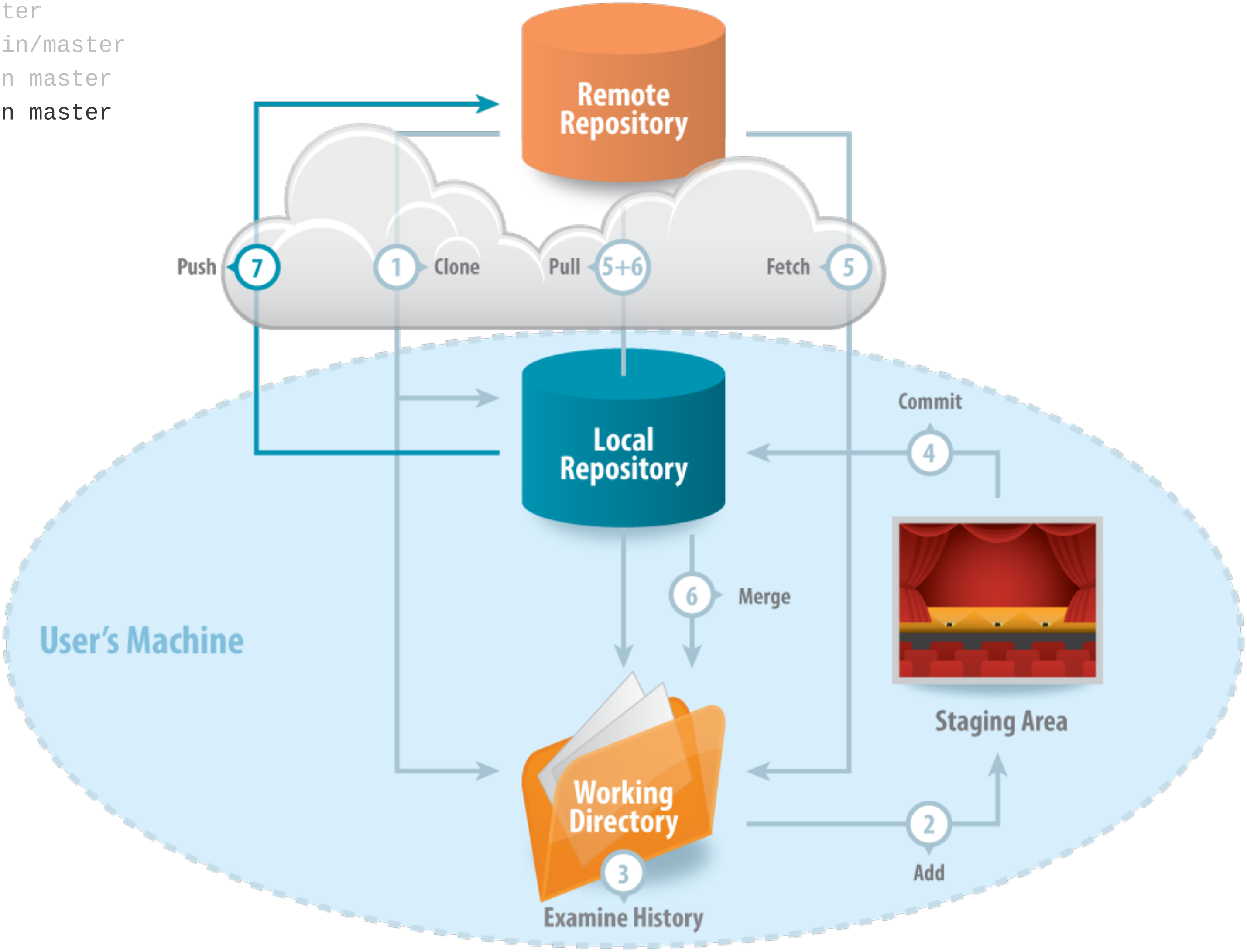
Detached HEAD

If you checkout any commit SHA1, tag, or remote branch then you will end up having a “detached HEAD“:

```
$ git checkout 494e2cb73ed6424b27f9766bf8a2cb29770a1e7e
Note: checking out '494e2cb73ed6424b27f9766bf8a2cb29770a1e7e'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 494e2cb... Added README file
```

Remote work cycle

```
git clone ssh://serverpath/my-project
git add file1 (git mv file2 file4   git rm file3)
git origin master
git merge origin/master
git pull origin master
git push origin master
```



Branching

Branches

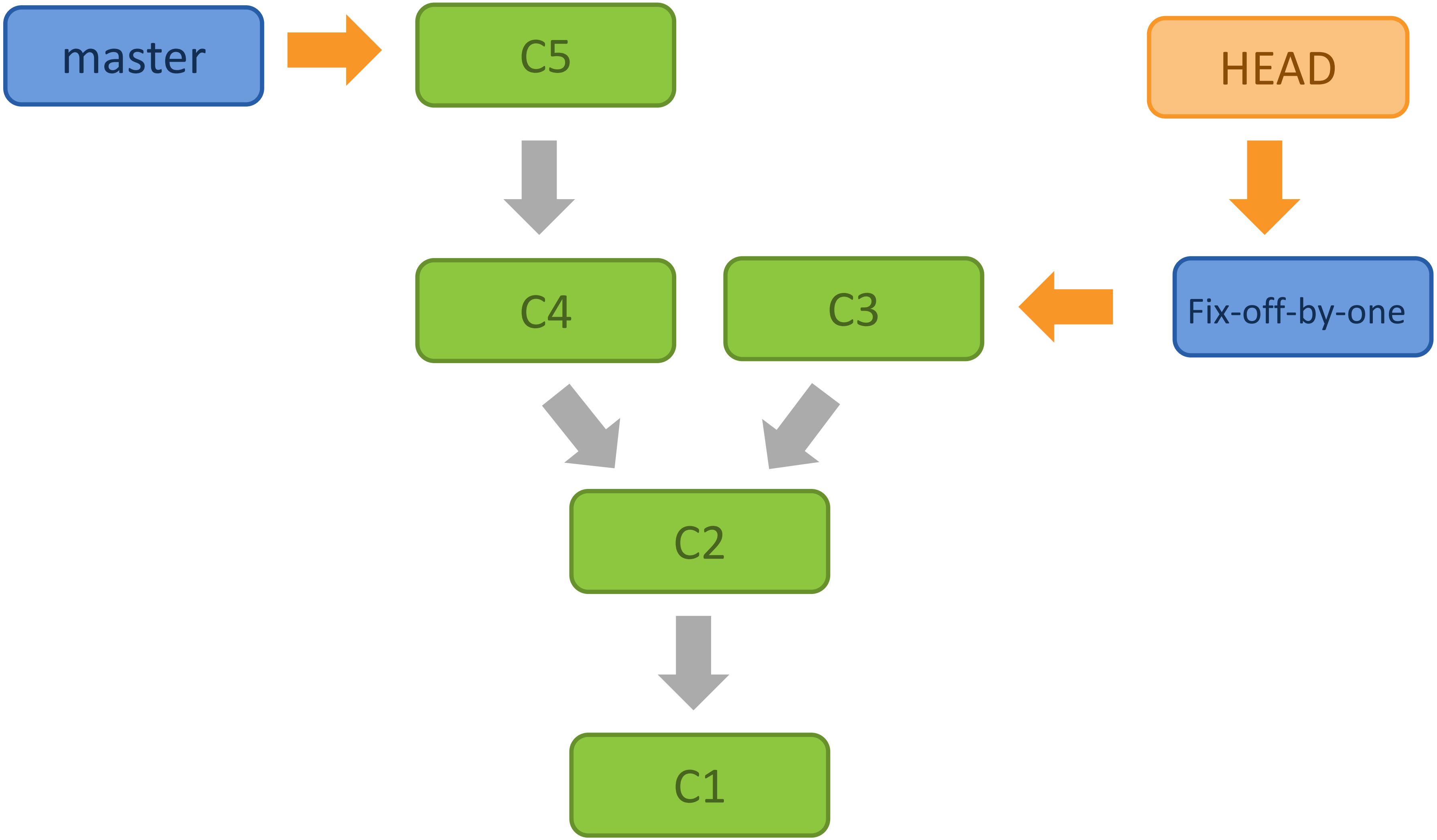
- A branch is a separate line of development, sharing a common history with other lines
- A branch is a complete tree (it cannot be created from a subtree)
- With Git, a branch is a 'pointer' (a file holding a commit hash) into the DAG

  This makes branches pretty cheap

- `git branch` shows you all existing branches with the branch prefixed by a * being the current branch

```
$ git branch
  fix-off-by-one
* master
```

Branches (Cont'd)



Branch layout

The branch layout is up to you, but there are some best practices though:

```
$ git branch # GOOD
  master
* devel
  feature/new-mailform
  fix/off-by-one
  fix/readme-grammar
```

```
$ git branch # BAD
  master
* devel
  new
  fix
  fix2
  t3rrible-br@nch-name
```

Branch creation

- `git branch fix-off-by-one` creates a new branch `fix-off-by-one` based on the current branch

```
$ git branch fix-off-by-one
```

- Alternatively, you can also create a branch using a commit SHA1

```
$ git branch feature_branch 4445682
```

Checkout a branch

- `git checkout <branch>` is used to switch a branch. Switching a branch means:
  - Updating the index
  - Updating the files in the working tree
  - Updating HEAD
- The checkout is aborted in the case where uncommitted changes to the working tree would be overwritten, otherwise changes float over

```
$ git checkout master
error: Your local changes to the following files would be
overwritten by checkout

        README.txt
Please, commit your changes or stash them before you can
switch branches.
Aborting.
```

Checkout a branch (Cont'd)

- `git checkout fix-off-by-one` to check out the branch

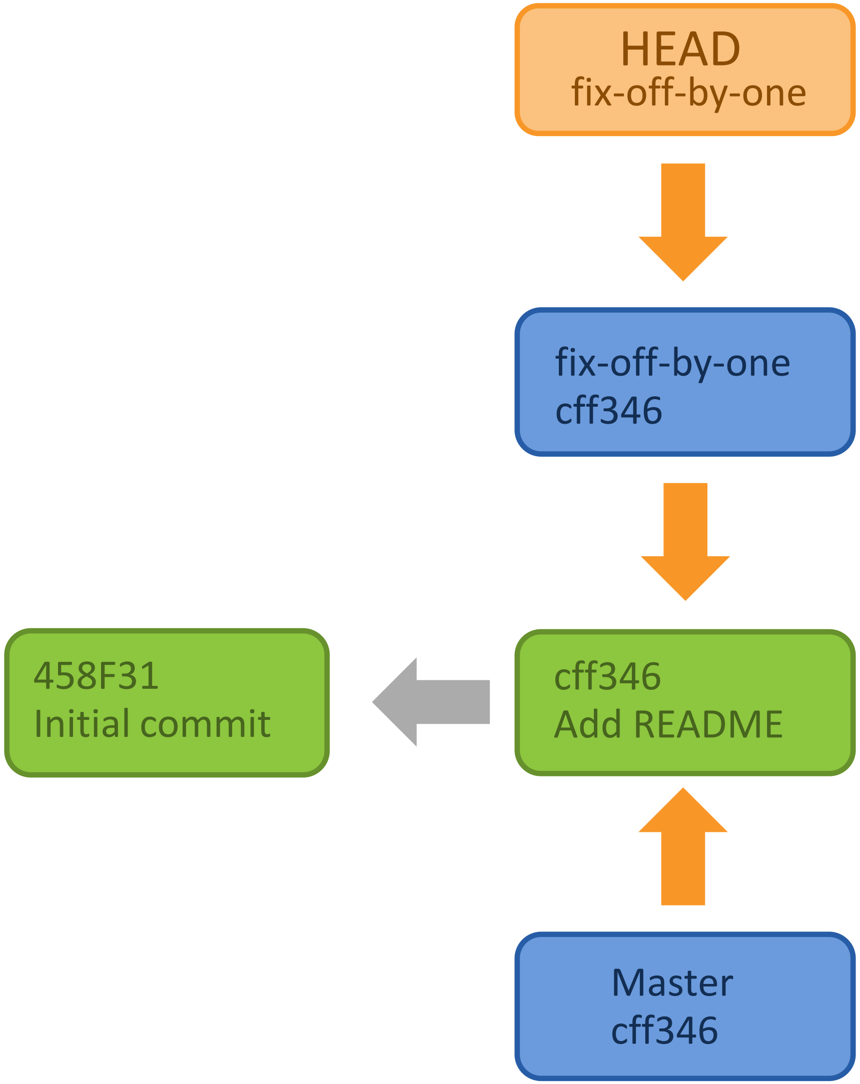
```
$ git checkout fix-off-by-one
Switched to branch 'fix-off-by-one'
```

- `git checkout -b fix-off-by-one` to create and checkout a branch in one command (creating a new branch "fix-off-by-one")

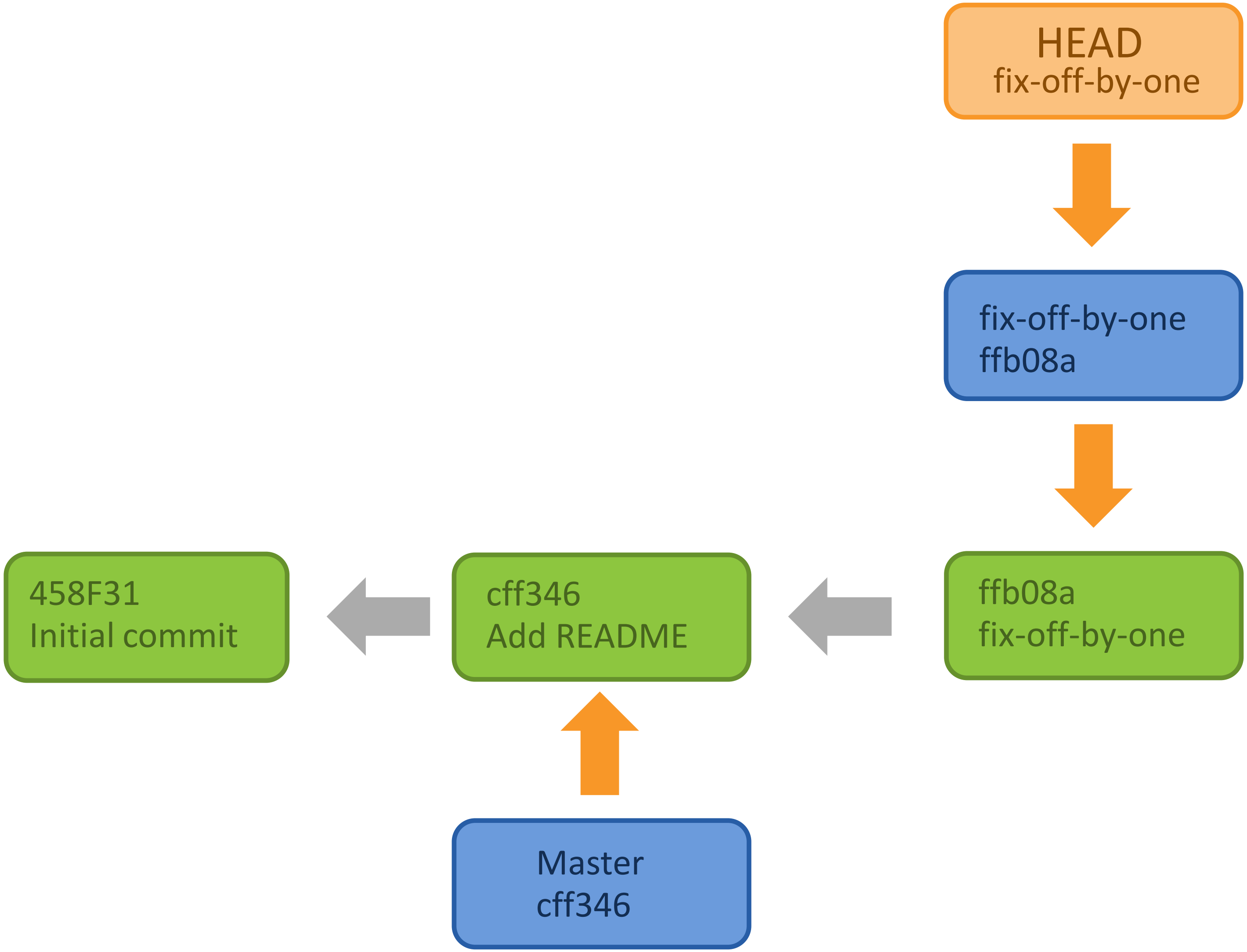
```
$ git checkout -b fix-off-by-one
Switched to a new branch 'fix-off-by-one'
```

Equivalent to `git branch fix-off-by-one + git checkout fix-off-by-one`

Branches (Cont'd)



Branches (Cont'd)



HEAD
fix-off-by-one

fix-off-by-one
ffb08a

458F31
Initial commit

cff346
Add README

ffb08a
fix-off-by-one

Master
cff346

Merging branches

After you commit your changes in the branch, you ***merge*** the branch back into master:

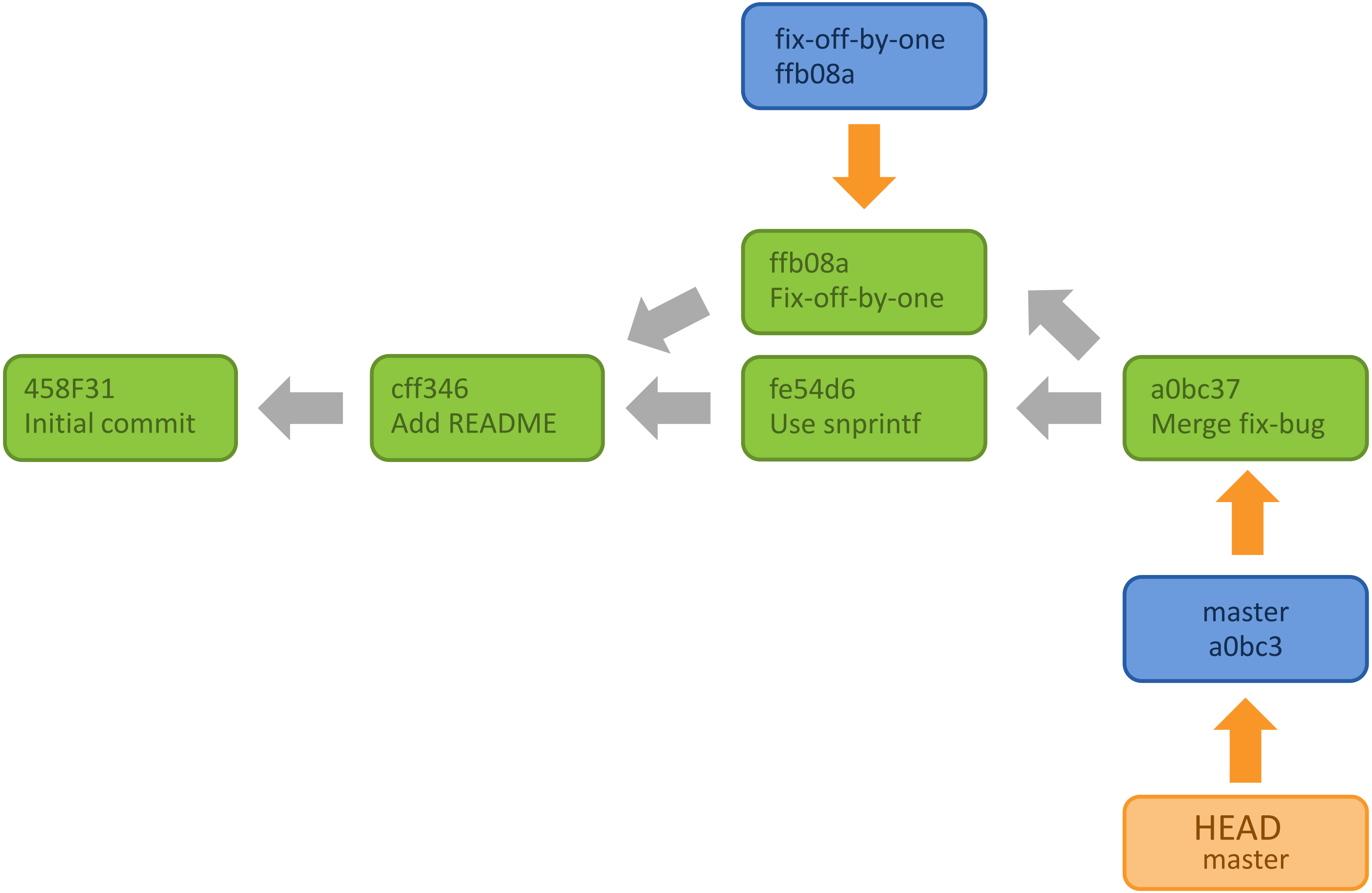
- `git checkout master` to switch back to master

```
$ git checkout master
Switched to branch 'master'
```

- `git merge fix-off-by-one` to merge your fix

```
$ git merge fix-off-by-one
Updating 494e2cb..7f7d0f3
Fast-forward
 README.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Branches (Cont'd)



Standard branch use workflow

- Create a new feature branch

```
git checkout -b feature/new-feature
```

- Implement your feature

```
git commit -m 'New feature implemented'
```

- Merge your change sets back

```
git checkout master
git merge feature/new-feature
```

- Remove local branch and push your change sets

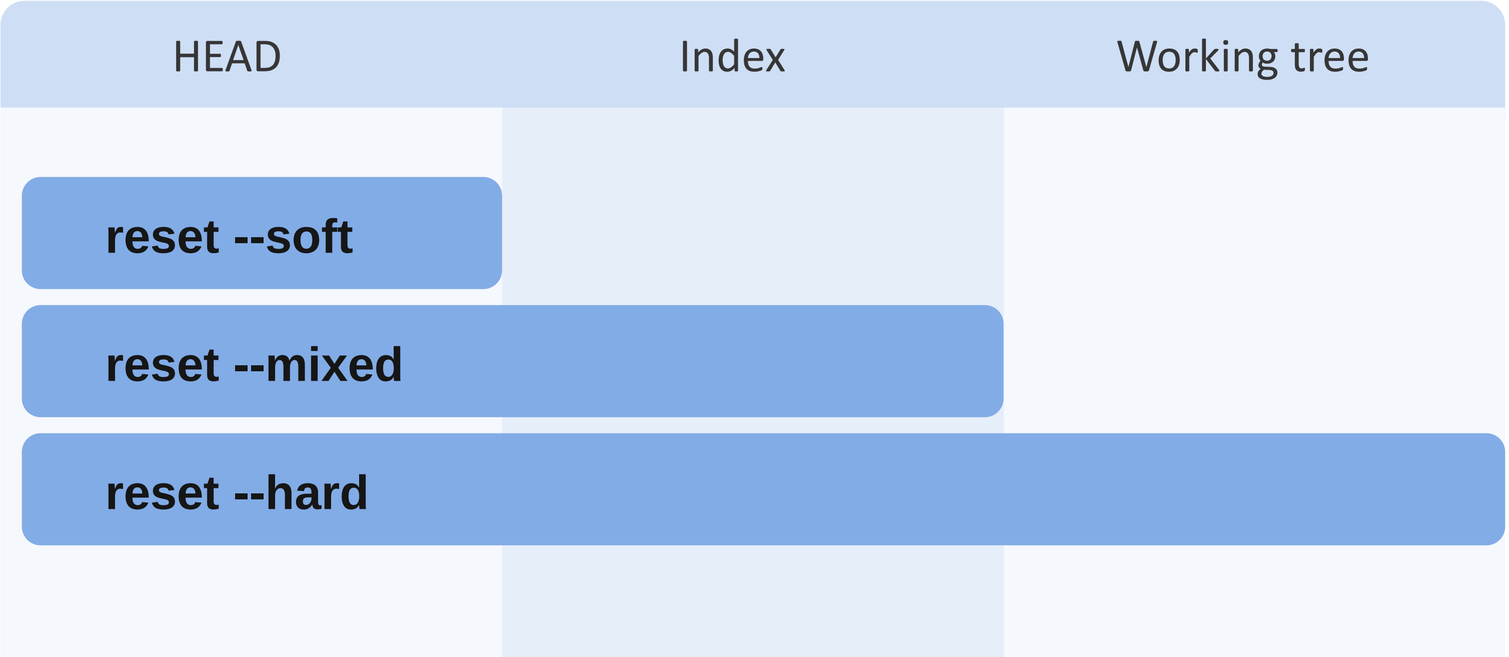
```
git branch -d feature/new-feature
git push origin HEAD:master
```

Reset vs. checkout

- `git reset` is used to reset the HEAD, index and working tree based on a specified state (defaults to `--soft`)
- It is also possible to reset only a certain path

You can also reposition pointer to the HEAD by specifying index

```
$ git reset --hard HEAD~1
HEAD is now at 4445682 Initial empty repository
```



Reset vs. **checkout**

- `git checkout` is used to checkout either a:

Branch	`git checkout [..][<branch>]`
Commit	`git checkout [..][<commit>]`
Patch	`git checkout [..][<patch>]`
Path	`git checkout [..][<path>]`

- Allows you to update the ***given paths with specified version in index , tree***
  - Updates the named paths according to the index file
  - Updates HEAD only in the case of checking-out a branch
- Common usage:
  - `git checkout -b <new branch> [<start point>]`

Reset vs. **checkout**

```
git checkout [..][<tree-ish>] [--]
<pathspec>...
```

Allows you to ***update*** the given (named) paths according to the index file or according to the given tree-ish

Remember: a `<tree-ish>` object is a tree object or an object which can be peeled to a tree (i.e. a tag or commit)

Examine full history

- `git log`
- `git diff`
- `git show`
- `git blame`

Git log

- `git log` shows you the history of the current branch or object (file or folder)
- Important options:

  `-<n>` to limit the number of commits

  `--stat` to see a short statistic for each commit

```
$ git log --stat
Commit 494e2cb73ed6424b27f9766bf8a2cb29770a1e7e
Author: Alice <alice@collab.net>
Date:   Thu Nov 29 18:14:22 2012 +0100

   Added README file

 README.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)

commit 4445682c417ae50096846366104a485a895a851f
Author: Gerrit Code Review <gerrit@localhost.localdomain>
Date:   Thu Nov 29 08:42:49 2012 -0800

    Initial empty repository
```

  `--since`, `--after`, `--until`, `--before` for time based history

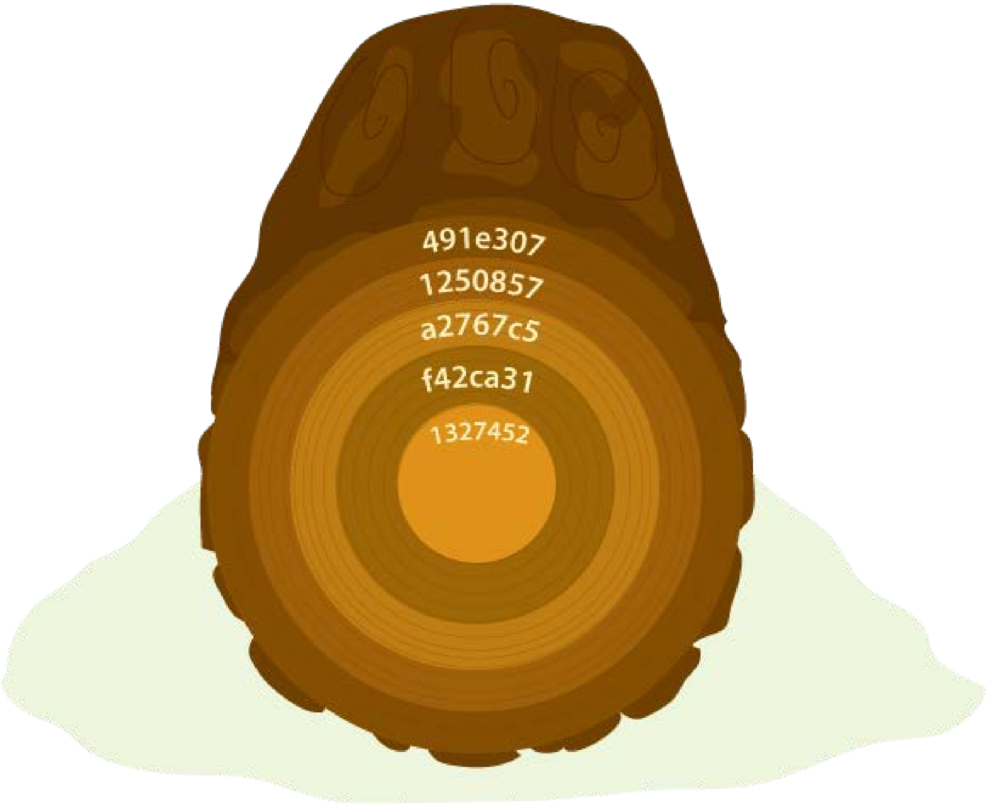
  `--grep` to grep the commit message

  `-S` to grep the commit content

  `--oneline` to show only message subjects

Git log example

```
$ git log --oneline --after=2012-11-30 --author "Jeff King"
491e307 status: respect advice.statusHints for ahead/behind advice
1250857 launch_editor: propagate signals from editor to git
a2767c5 run-command: do not warn about child death from terminal
f42ca31 launch_editor: refactor to use start/finish_command
1327452 run-command: drop silent_exec_failure arg from wait_or_whine
```



Git diff

- `git diff` shows changes between commits, files, etc.
- Important options:

  `--name-only` or `--name-status`

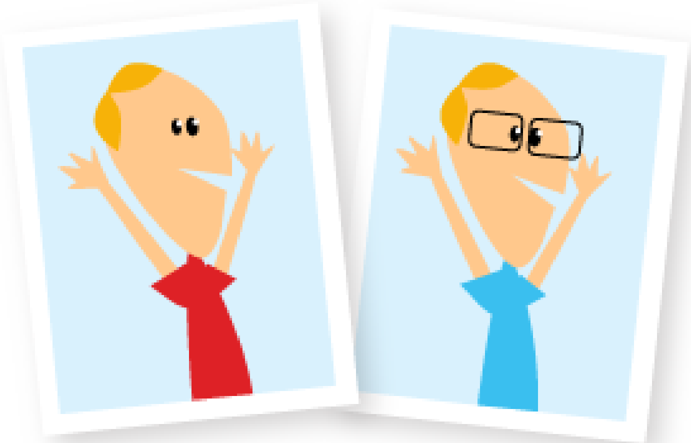
  `--numstat` or `--shortstat`

  `-S<string>` to find differences that add or remove `<string>`

  `--ignore-space-change` ignores white space when comparing lines

  `--summary` shows smallest info about change

  `--color` shows colored diff



Git diff example

Example – showing a diff between the current and last revision:

```
$ git diff HEAD~1
diff --git a/README.txt b/README.txt
index e965047..f9264f7 100644
--- a/README.txt
+++ b/README.txt
@@ -1 +1,2 @@
 Hello
+World
```

Git show

- `git show` can be used to show information about an object including a diff for commit objects
- Important options:

  `--pretty[=<format>], --format=<format>`

  `--notes | --no-notes`

```
$ git show
commit 494e2cb73ed6424b27f9766bf8a2cb29770a1e7e
Author: Alice <alice@collab.net>
Date:   Thu Nov 29 18:14:22 2012 +0100

   Added README file

diff --git a/README.txt b/README.txt
new file mode 100644
index 0000000..e845566
--- /dev/null
+++ b/README.txt
@@ -0,0 +1 @@
+README
```

Git blame

- git blame annotates a file showing the revision and author for the last modification of each line of a file

```
$ git blame README.txt
494e2cb7 (Alice 2012-11-29 18:14:22 +0100 1) README
d4c1adde (Alice 2012-12-04 11:18:20 +0100 2) Added more into readme
```

- You can also specify a specific line number you want to have annotated

```
$ git blame README.txt -L 2
d4c1adde (Alice 2012-12-04 11:18:20 +0100 2) Added more into readme
```

.gitignore

- Specify files not to be tracked
  - Each line specifies a pattern
  - Basic pattern format
    - Wildcard '*'
    - Negotiation '!'
    - Directory macthing '/,
- .gitignore used for public ignores
- .git/info/exclude for private files

.gitignore example

```
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#     main.o
nothing added to commit but untracked files present (use "git add" to track)

$ echo "*.o" >> .gitignore

$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#     .gitignore
nothing added to commit but untracked files present (use "git add" to track)
```

Git cheat sheet

Configure

user profile

```
git config user.name <uname>
git config user.email <uemail>
```

preferred editor /color

```
git config core.editor vim
git config color.ui true
```

Create

From existing files

```
git init
git add .
```

From existing repository

```
git clone ~/old ~/new
git clone git://...
git clone ssh://...
```

Publish

```
git commit [-a]
(-a: add changed files)
automatically)

git format-patch origin
(create set of diffs)

git push remote
(push to origin or remote)

git tag foo
(mark current version)
```

View

```
git status
git diff [oldid newid]
git log [-p] [file|dir]
git blame file
git show id
(meta data + diff)

git branch
(shows list, * = current)

git tag -l
(shows list)
```

Update

```
git fetch
(from def. upstream)

git fetch remote
git pull
(= fetch & merge)

git apply patch.diff
```

Where to get help

- `git <command> -h`
  - Short overview
- `git help <command>`
  - Detailed information on command (manpage)
- `git help -w|--web <command>`
  - Show in web browser

Where to get help (Cont'd)

- http://git-scm.com/docs - Git manpages, etc.
- http://git-scm.com/book - free version of ProGit
- #git on freenode.net
- http://groups.google.com/group/git-users - Git user community, unofficial
- http://vger.kernel.org/vger-lists.html#git - Git developer mailing list

Thank you!

About CollabNet

CollabNet is a leading provider of Enterprise Cloud Development and Agile ALM products and services for software-driven organizations. With more than 10,000 global customers, the company provides a suite of platforms and services to address three major trends disrupting the software industry: Agile, DevOps and hybrid cloud development. Its CloudForge™ development-Platform-as-a-Service (dPaaS) enables cloud development through a flexible platform that is team friendly, enterprise ready and integrated to support leading third party tools. The CollabNet TeamForge® ALM, ScrumWorks® Pro project management and SubversionEdge source code management platforms can be deployed separately or together, in the cloud or on-premise. CollabNet complements its technical offerings with industry leading consulting and training services for Agile and cloud development transformations. Many CollabNet customers improve productivity by as much as 70 percent, while reducing costs by 80 percent.

For more information, please visit www.collab.net.

CollabNet®

CollabNet, Inc.
8000 Marina Blvd., Suite 600
Brisbane, CA 94005

www.collab.net

+1-650-228-2500
+1-888-778-9793
blogs.collab.net
twitter.com/collabnet
www.facebook.com/collabnet
www.linkedin.com/company/collabnet-inc