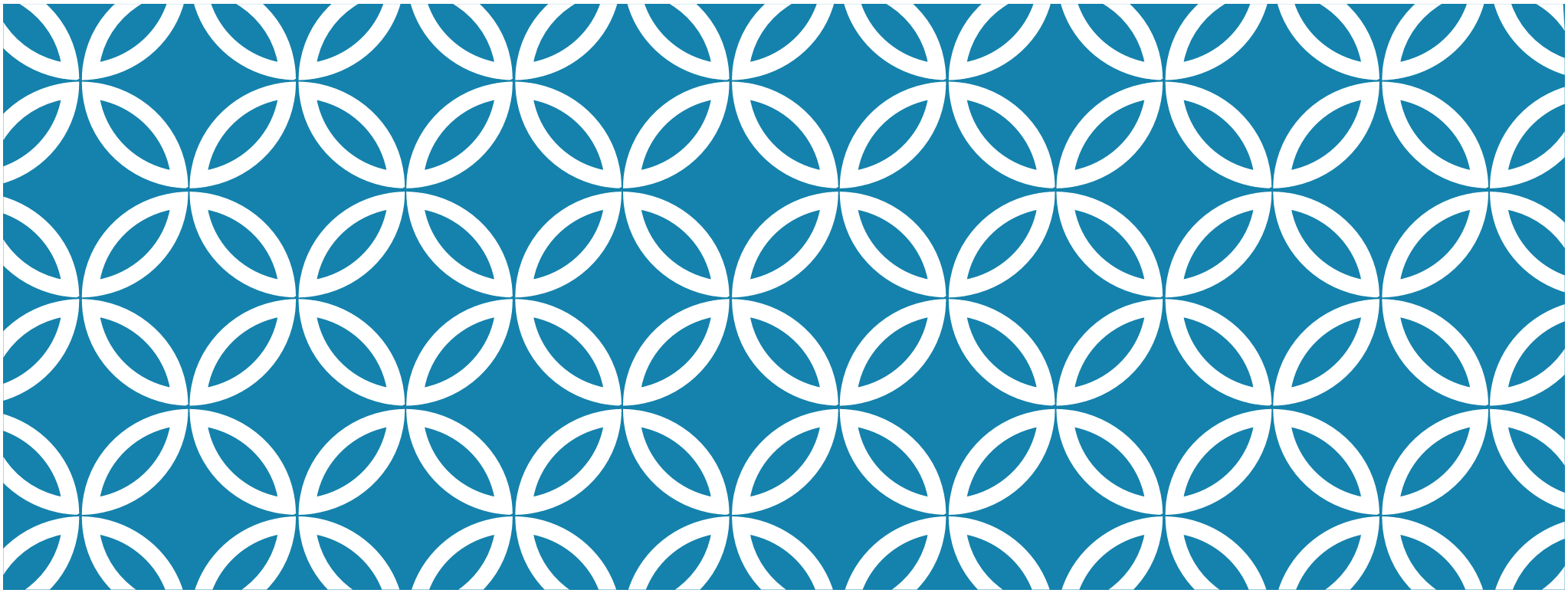


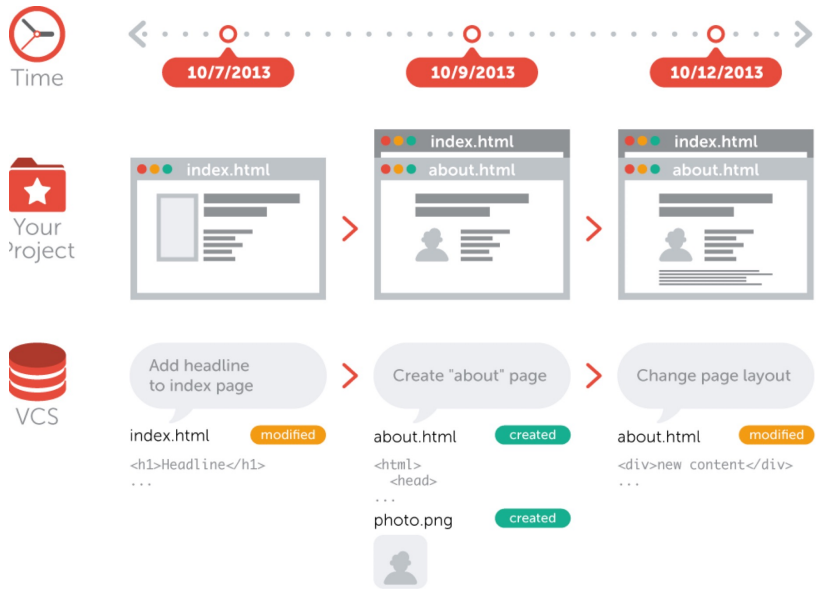
PRACTICES > GIT BRANCHING

CS 346: Application Development

VERSION CONTROL W/ GIT

Version Control Systems (VCS) are systems that track changes to your files.

- e.g., Git, Subversion (SVN), Perforce.

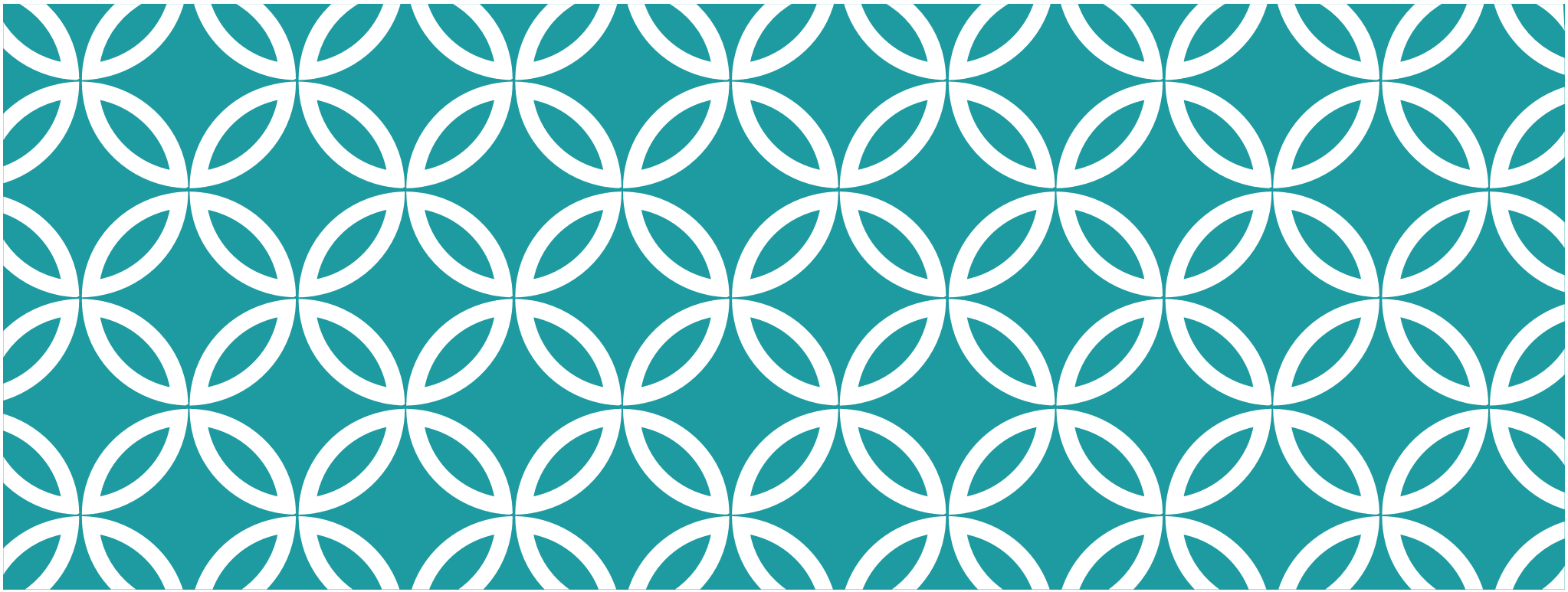


Git tracks changes to **sets** of files over time.

BENEFITS OF VERSION CONTROL

A VCS provides some major benefits:

- **History**: a VCS provides a long-term history of every file. This includes tracking when files were added, or deleted, and every change that you've made.
- **Versions**: the ability to version sets of files together. Did you break something? You can always unwind back to the "last good" change that was saved, or ever compare your current code with the previously working version to identify an issue.
- **Collaboration**: a VCS provides the necessary capabilities for multiple people to work on the same code simultaneously, while keeping changes isolated.



GIT BASICS

CS 346: Application Development

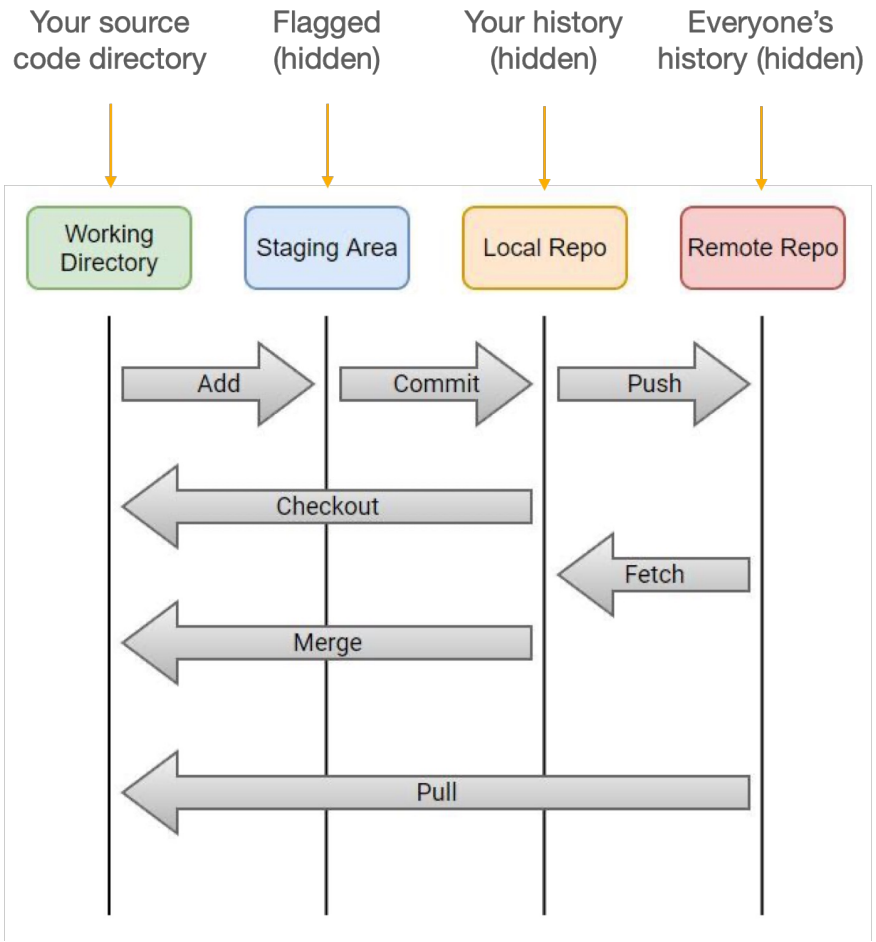
CONCEPTS

Git is designed around these core concepts:

Working Directory: A copy of your repository, where you will make your changes before saving them in the repository.

Staging Area: A logical collection of changes from the working directory that you want to collect and work on together (e.g. a feature that resulted in changes to multiple files).

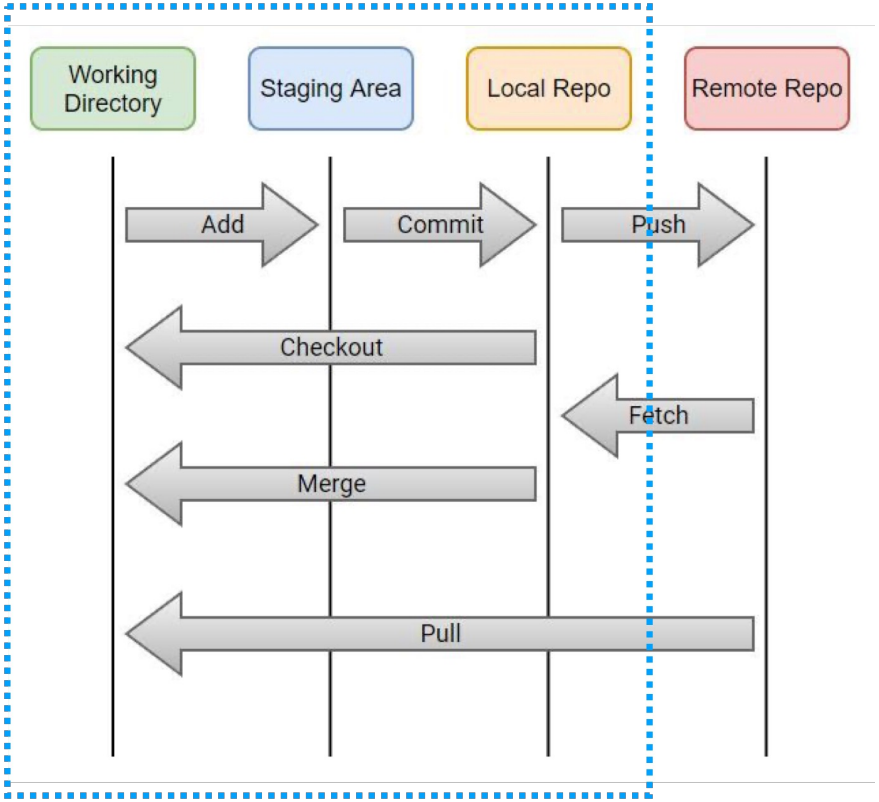
Repository: The location of the canonical version of your source code ("Local Repo" in this diagram).



LOCAL WORKFLOW

A local Git workflow looks like this:

1. **Create a project directory** for your source code.
2. **Initialize a git repository** in this directory. This doesn't change your source code but adds a hidden .git directory to track it as a repository.
3. **Make changes to your source code** in your favourite editor (e.g. add features, fix a bug!).
4. **Add the changed files to your staging area** (git add). Commit the files in the staging area to save to the repository (git commit). This two-step process ensures that these files are tracked and versioned as a single change.
5. **Check that your changes have been saved** by using git status.



LOCAL COMMANDS

Command	Description	Example

git init	Create a new repository in the current directory.	$ mkdir repo; cd repo
		$ git init
		Initialized empty Git repository in /repo/.git/

git add	Add a file to the staging area	$ vim readme.md $ git add readme.md
git commit	Commit all staged files to the repo	$ git commit -m “Added readme” [master (root-commit) d3c834b] Added readme 1 file changed, 0 insertions(+), 0 deletions(-) create mode 100644 readme.md
git status	Display the status of the current staging area.	$ git status On branch master nothing to commit, working tree clean
git checkout	Checkout a specific commit to this working area. Use to revert a file.	$ git checkout main.kt Updated 1 path from index.

REMOTE WORKFLOW

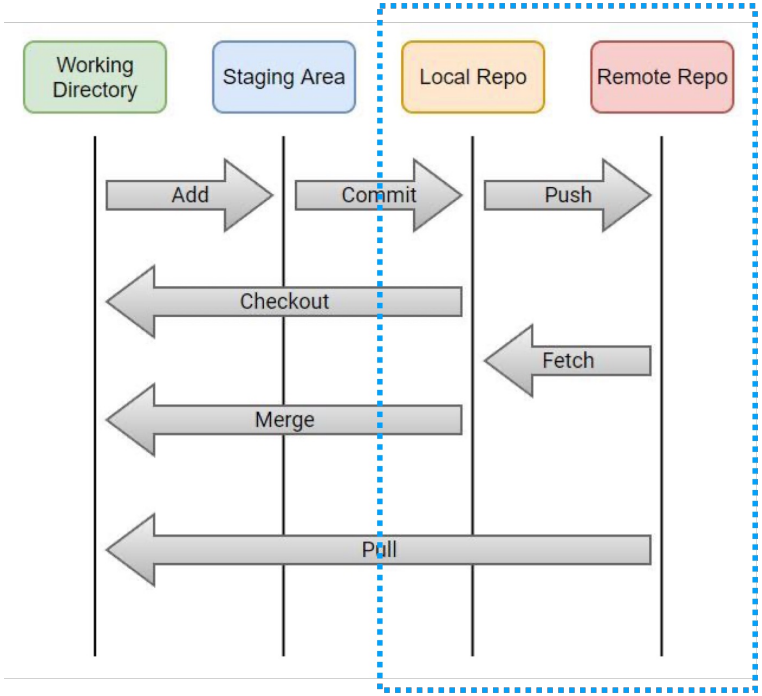
You can use Git locally without any restrictions.

However, we often want a remote repository:

- This provides a single "source of truth" that contains everyone's changes (and which we can backup!)
- It helps us coordinate changes with our team (i.e. if I make a change, it gives me a mechanism to share that change with everyone else).

To work with a remote repository:

- We need to setup a remote repository, or use an existing hosting site (e.g. GitLab, GitHub).
- We add a connection between local and remote repositories.
- We continue to make changes locally, but then "push" the changes to the remote as an additional step (git push).



REMOTE WORKFLOW

The common workflow for working with a remote repository is like the local workflow:

1. **Initialize a git repository** in the remote directory (e.g. in GitLab create a new project).
2. **Clone the remote repository** to create a local project directory (git clone using the URL of the repo that you created in the previous step).
3. **Make changes to your source code** in your favourite editor (e.g. add a new feature, fix a bug!).
4. **Add the changed files to your staging area** (git add). Commit the files in the staging area to save to the repository (git commit). This two-step process ensures that these files are tracked and versioned as a single change.
5. **Push the changes** from your local repo to the remote repo (git push).
6. **Check that your changes have been saved** by using git status.

REMOTE COMMANDS

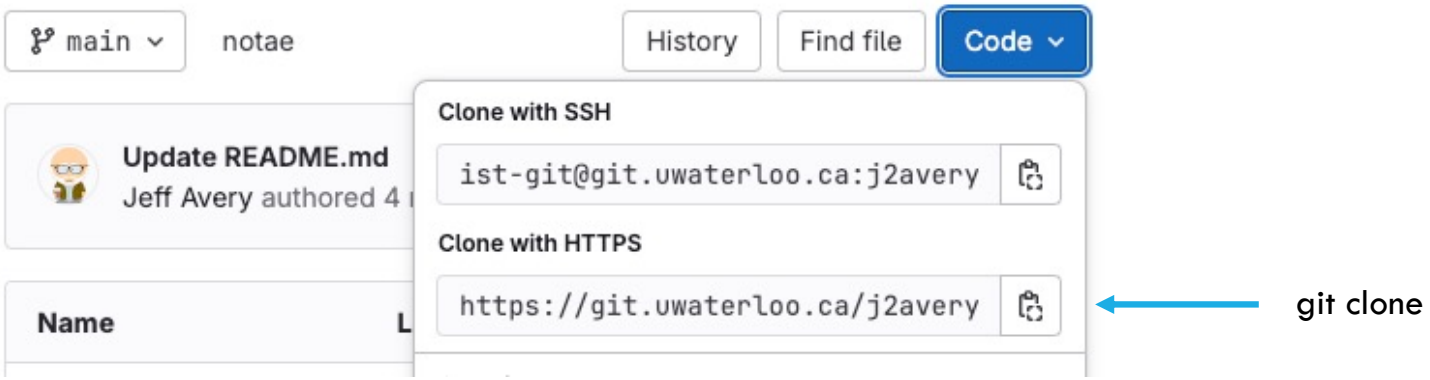
Command	Description	Example
git clone	Clone the remote repository to a local directory.	$ git clone https://git.uwaterloo.ca/j2avery/cs349-public.git repo Cloning into 'repo'... remote: Enumerating objects: 531, done. remote: Counting objects: 100% (531/531), done. remote: Compressing objects: 100% (280/280), done. remote: Total 2702 (delta 209), reused 320 (delta 100), pack-reused 2171 Receiving objects: 100% (2702/2702), 7.30 MiB | 13.00 MiB/s, done. Resolving deltas: 100% (939/939), done.

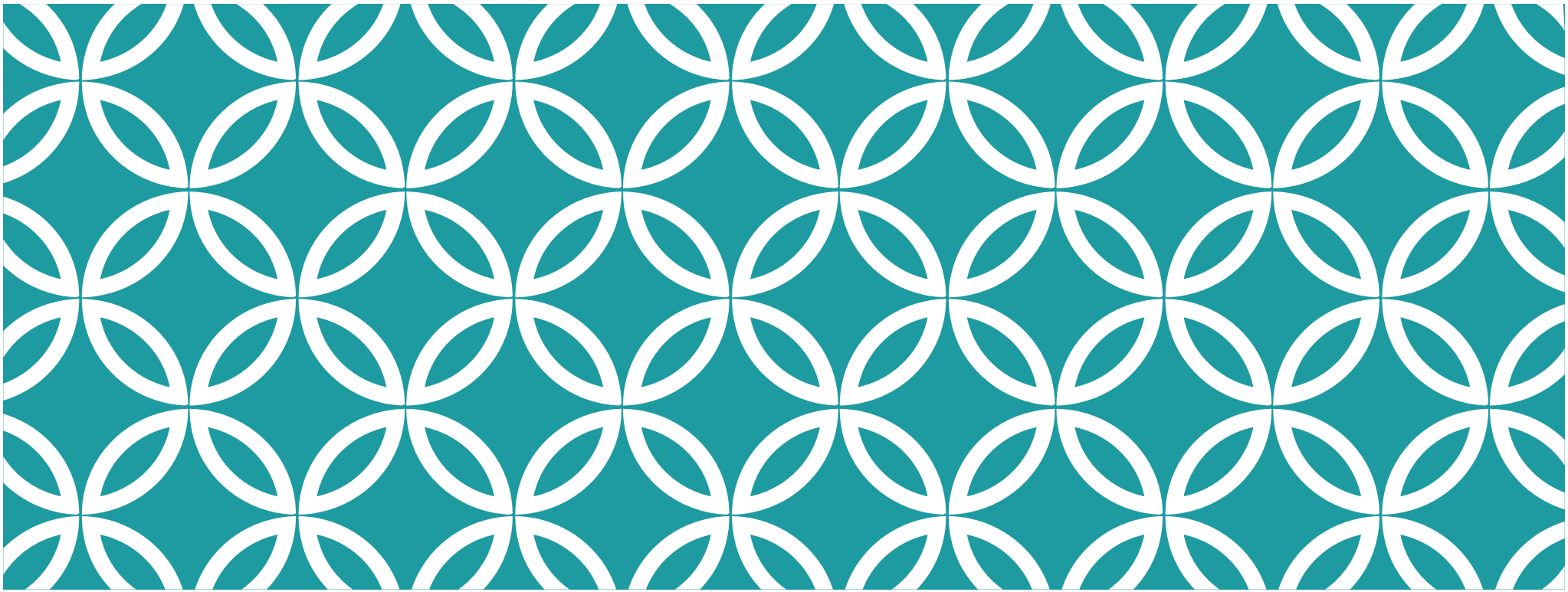
git pull	Merge changes into the local repo.	$ cd repo
		$ git pull
		Already up to date.

git remote	Modify the remote connection.	$ git remote origin

TIP: GETTING STARTED WITH GITLAB?

1. Create your project in GitLab.
2. Clone the project URL from the project home page. This gives you a working copy.
3. Add your source code to that directory, commit and push it.





BRANCHING STRATEGIES

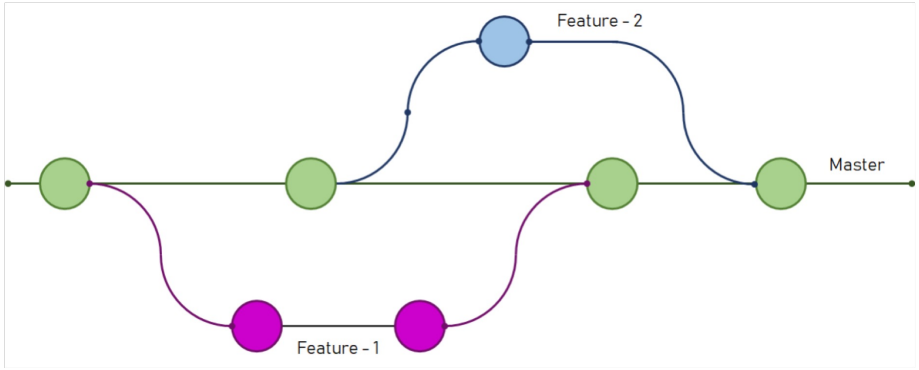
CS 346: Application Development

CONCEPT: BRANCHING

Think of a repository the central location for storing changes or commits.

The main set of commits is like a trunk of a tree, where the commits are added in sequence (aka **main**). A **branch** is a fork in the tree, where we "split off" work and diverge from one of the commits.

Branches diverge from a specific commit, and do not include changes that happened on the trunk after the branch occurred.



Feature branches, merged back into Main once the feature is complete and tested.

FEATURE BRANCHES

A main reason to create branch is to isolate our work from other changes on the trunk. Once we have a feature implemented and tested, we can merge our changes back into the trunk ("integration").

These type of branches are called **feature branches** and isolate untested work.

A typical workflow for adding a feature would be:

1. Create a feature branch for that feature.
2. Make changes on your branch only. Test everything.
3. (Optional) Have changes code reviewed by someone on your team (see *Pull Request*).
4. Switch back to master and merge from your feature branch to the master branch.

```
$ git checkout -b test // create branch
Switched to a new branch 'test'


$ vim file1.md // make some changes
$ git add file1.md
$ git commit -m "Committing changed to file1.md” // changes on test branch


$ git checkout master // switch to master
$ git merge test // merge changes from test
Updating 09e1947..ebb5838
Fast-forward
 file1.md                    | 136 ++++++++++++++++++++++++++++++++++++++++++++++
 1 file changed, 118 insertions(+), 18 deletions(-)


$  git branch -d test // remove branch (optional)
Deleted branch test (was ebb5838).
```

COORDINATING WORK

How do you coordinate work across branches?

There are different approaches that have been taken, but some common ideas:

1. Create feature branches for development.
2. Merge changes from feature branches to trunk; main should always build.
   - Best practice: have tests on main that will automatically execute when you merge.
   - Always be ready to release from main.
3. Release from main branch.

BRANCHING MODELS: GIT FLOW

In the Git flow model, you have two main branches:

- `Develop` where all development takes place.
- `Main` or trunk that is only used for releases.

Developers create feature branches from the `Develop` branch. Pull requests are made, and changes reviewed on feature branches. After approval, changes are merged back to `Develop`.

Eventually, a collection of features are approved and merged from `Develop` to `Main` (trunk) and released as a product version.

Characteristics

- Long-lived feature branches, so merges are difficult.
- Careful control over integration into release branch.

https://www.toptal.com/software/trunk-based-development-git-flow

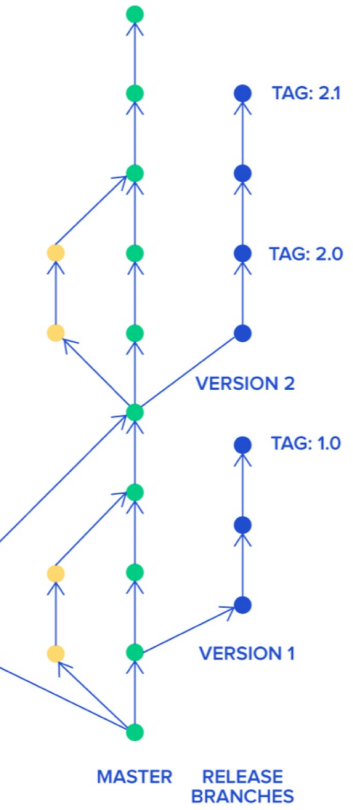
BRANCHING MODELS: TRUNK-BASED

In the trunk-based development model, all developers create feature branches from main, and merge changed directly back to main (after code reviews and tests are passing).

There is no release branch. Releases are cut directly from main, usually with tags to track the specific version that was released.

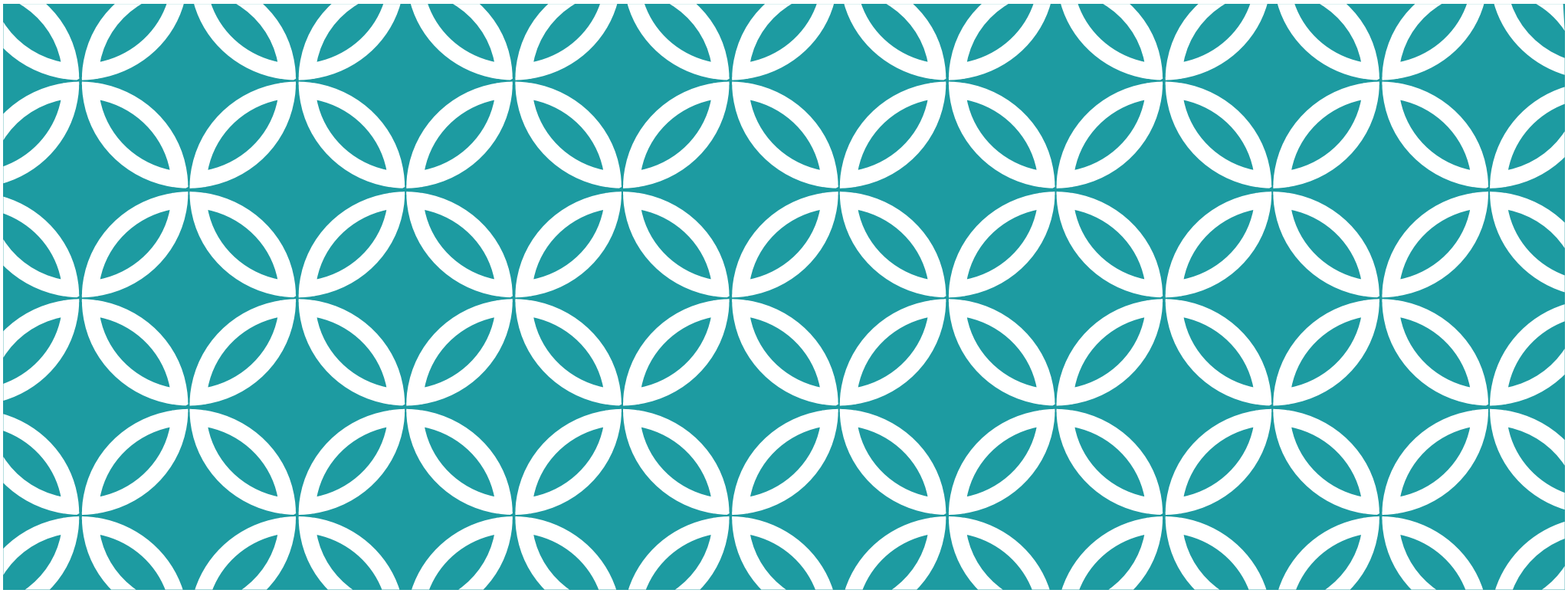
Characteristics

- Feature branches are short-lived.
- Development is continuous. Merges are frequent and issues are easier to resolve.
- Integration testing becomes critical!



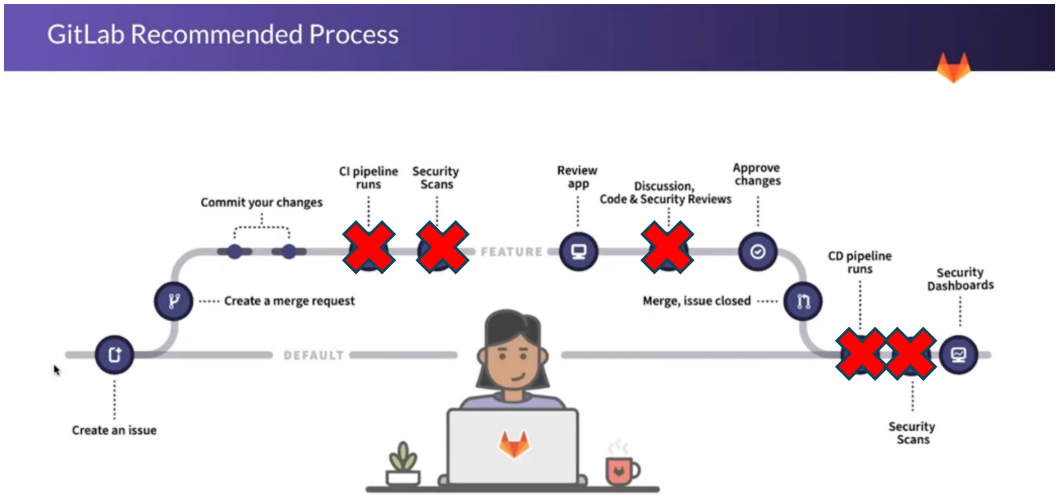
Do this.

https://www.toptal.com/software/trunk-based-development-git-flow



AUTOMATING WITH GITLAB

CS 346: Application Development



We're using trunk-based branching.

1. Make sure that you have an issue created.
2. Create a merge request and assign a Reviewer from your team. This will create a feature branch for you automatically to track this issue's work.
3. Make code changes and commit the changes to GitLab.
4. The reviewer can Approve the issue and flag it Closed. The feature branch will automatically merge to main, and then (optionally) will be removed.

https://xkcd.com/1597/