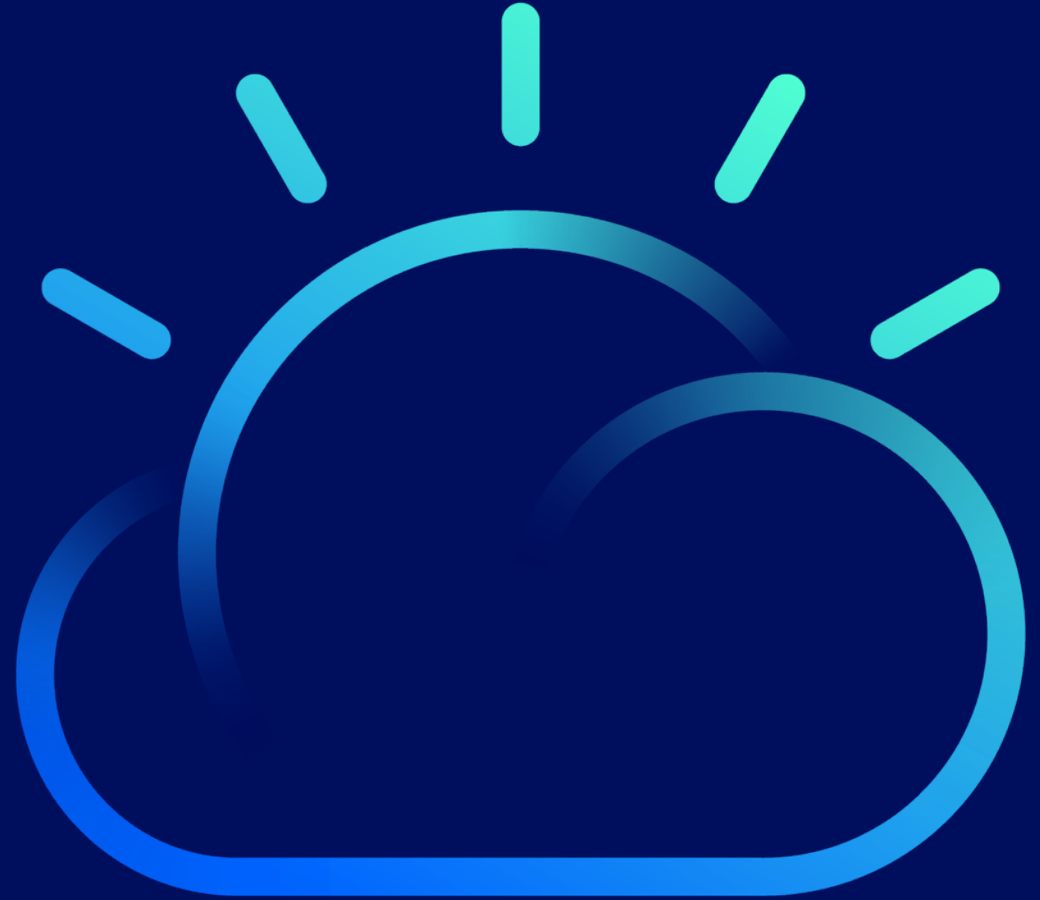


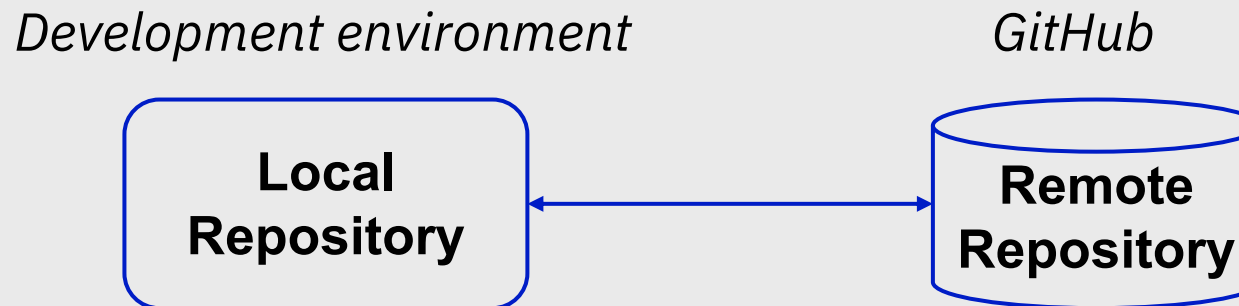
Working with GIT

Collaborative Infrastructure

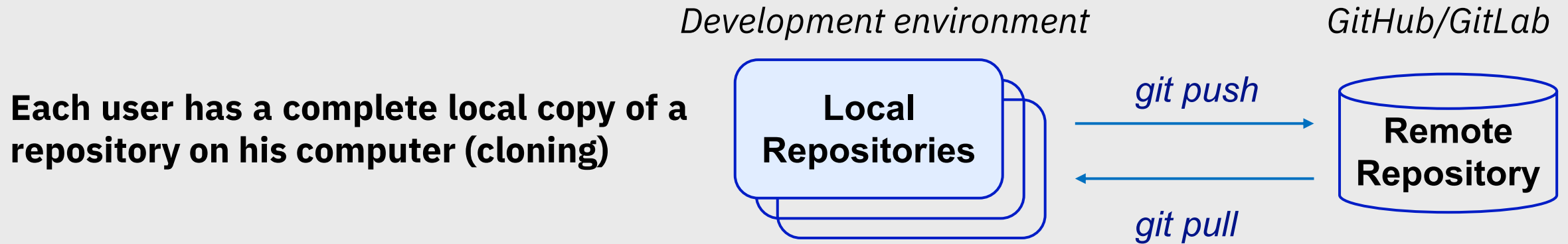


Why using a version control system such as Git ?

- Access to all versions of all terraform files at any time
- You don't lose a piece of code from a previous state of your environment
- Collaborate and work as a team without interfering with each other
- Terraform plan gives an opportunity for team members to review what has been done by each other



Distributed version control system



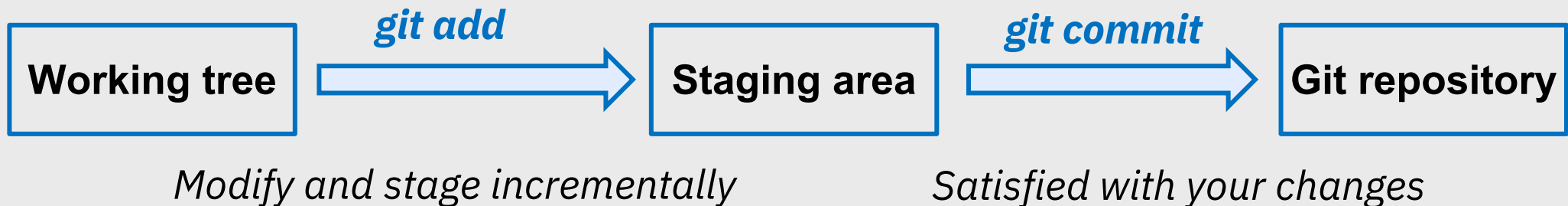
- The code is developed on each local computer
- GitHub is a central remote point hosting repositories and enabling users to obtain, alter and integrate changes through Git
- You have to setup a Git account as an hosting utility
- You can create a repository locally with the command ***git.init***
- Or you can create a repository on GitHub (web ui) and clone it locally using the command ***git.clone***

Working tree, staging area and repository

In a **local repository** a **working tree** is a collection of files which originate from a certain version of the repository.

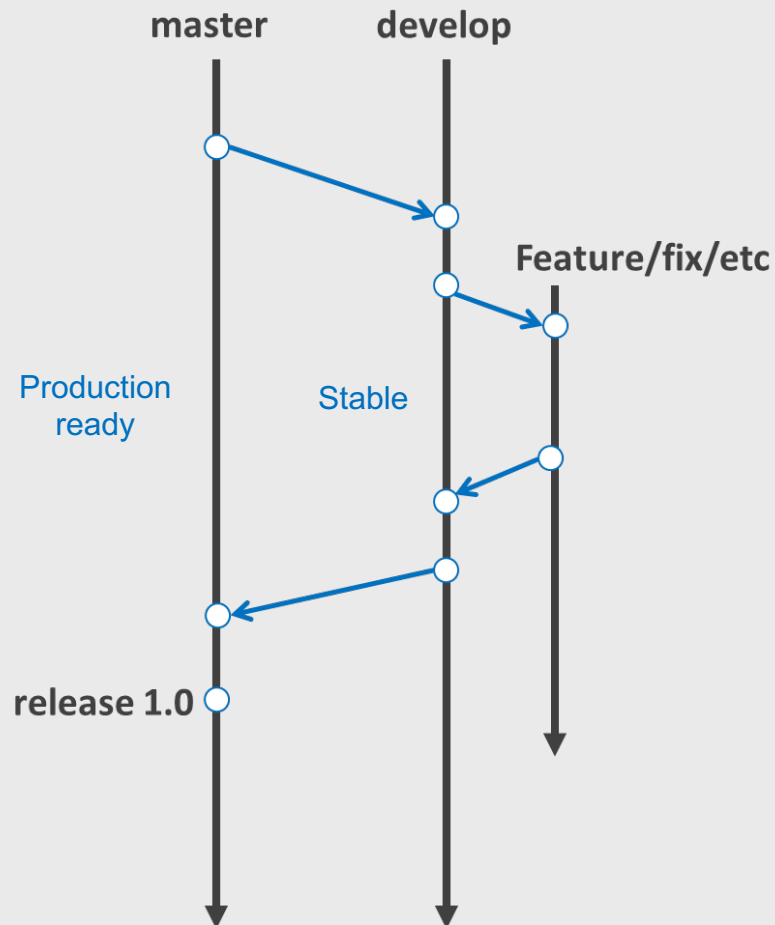
To persist changes in a working tree you need:

- To add selected changes to the staging area (Index) with the command *git add*
- To commit the staged changes into the git repository with the command *git commit*



A **branch** points to a specific commit. **HEAD** is a reference to the last commit in the currently check-out branch .

The concept of branches



The **master branch** is the default branch automatically created when cloning or creating a repository

A branch allows the user to switch between different versions of a collection of files so that he can work on different changes independently from each other.

Branches in Git are local to the repository.

The command *git checkout* allows to select a branch (switch to a branch) and the command *git branch* to create one. Use *git merge* to combine a branch in the current checkout branch.

Why branches ?

- Never commit on Master, instead uses branches created from the master and merge from your working branches
- Old code untouched until the new one works

Fetch, Merge and Stash

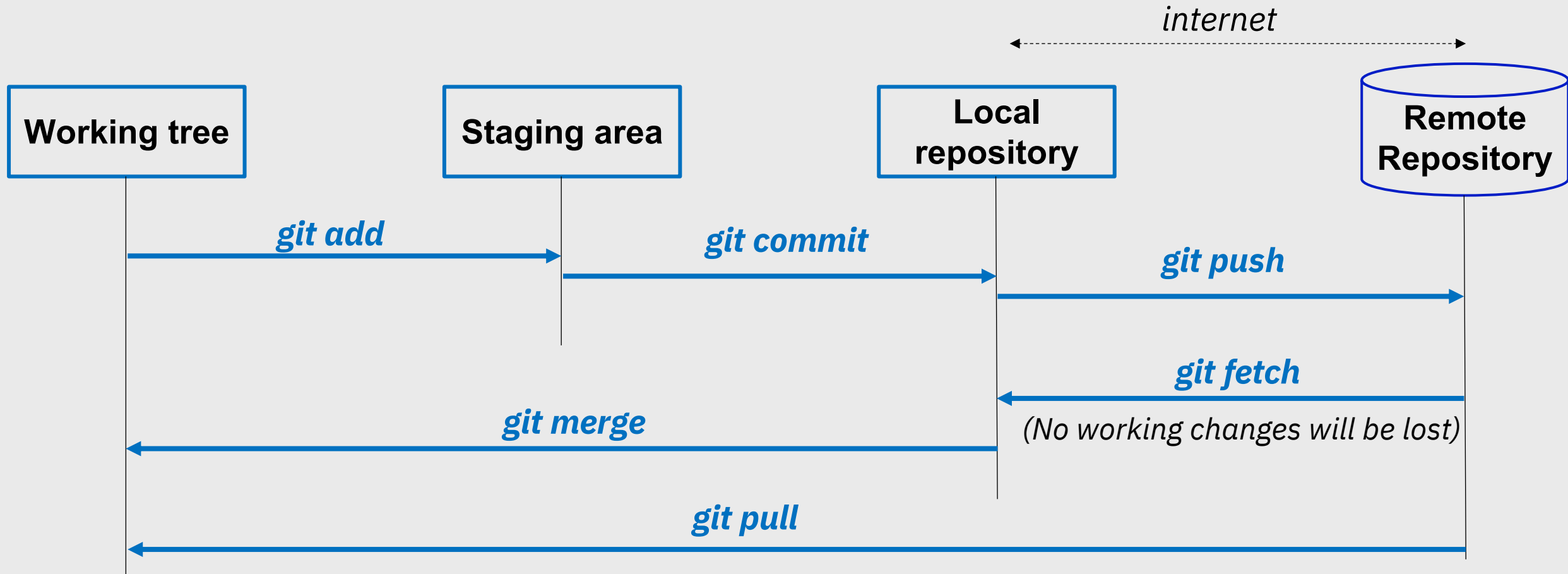
Fetch: See what everybody else has been working on. If you have conflicting files, you will need to manually **merge** them. Git fetch does not touch your working tree. It gives you the chance to decide what to do next.

Merge: When two files have been changed, merge them. After a fetch it brings the changes into your working tree. When merging branches the commit history get merged.

Pull: Fetch + merge

Stash: temporarily save uncommitted local changes and leave you with a clean working copy (*git stash*).... for example when you have to switch context to work on an urgent bug....to continue later on you can restore the saved state (*git stash pop*)

Put all together



Few more commands....

- The command *git status* retrieve the status of a file
 - ✓ Untracked: file whose changes are not monitored by Git and not declared in the gitignore file
 - ✓ Unstaged: file whose changes are being tracked by Git and changed since the last commit
 - ✓ Staged: file whose changes are being tracked by Git and ready for the commit
- Run *git reset* for unstaging stuff you don't want in your next commit
- Run *git rm* to remove files from your staging area and your working directory
- Run *git diff* to show changes between commits or commit and working tree
- Run *git show* to see an older version of a file (*run git show HEAD~2:file*)
- Run *git log -all* to see all commits on the branch currently checked out

Repository Tags

- Tags are used to identify specific significant points on a repository's history
- You tag a specific commit
- Use the command `git tag <tag name>`
- A tag is like a branch which does not change

Ignoring files

- Tell Git which files and directories to ignore, before you make a commit.
- Create a .gitignore file
- Commit the .gitignore file to share the ignore rules with any other users that clone the repository

```
vi .gitignore

# Local .terraform directories
**/.terraform/*

# .tfstate files
*.tfstate
*.tfstate.*

# .tfvars files
*.tfvars
```

Configure your environment

You already have a remote repository – Configure a remote repository URL

1 - Add a remote repo url to your local git configuration (create a connection to a remote repository)

➤ *git remote add <remote_name> <remote_repo_url>*

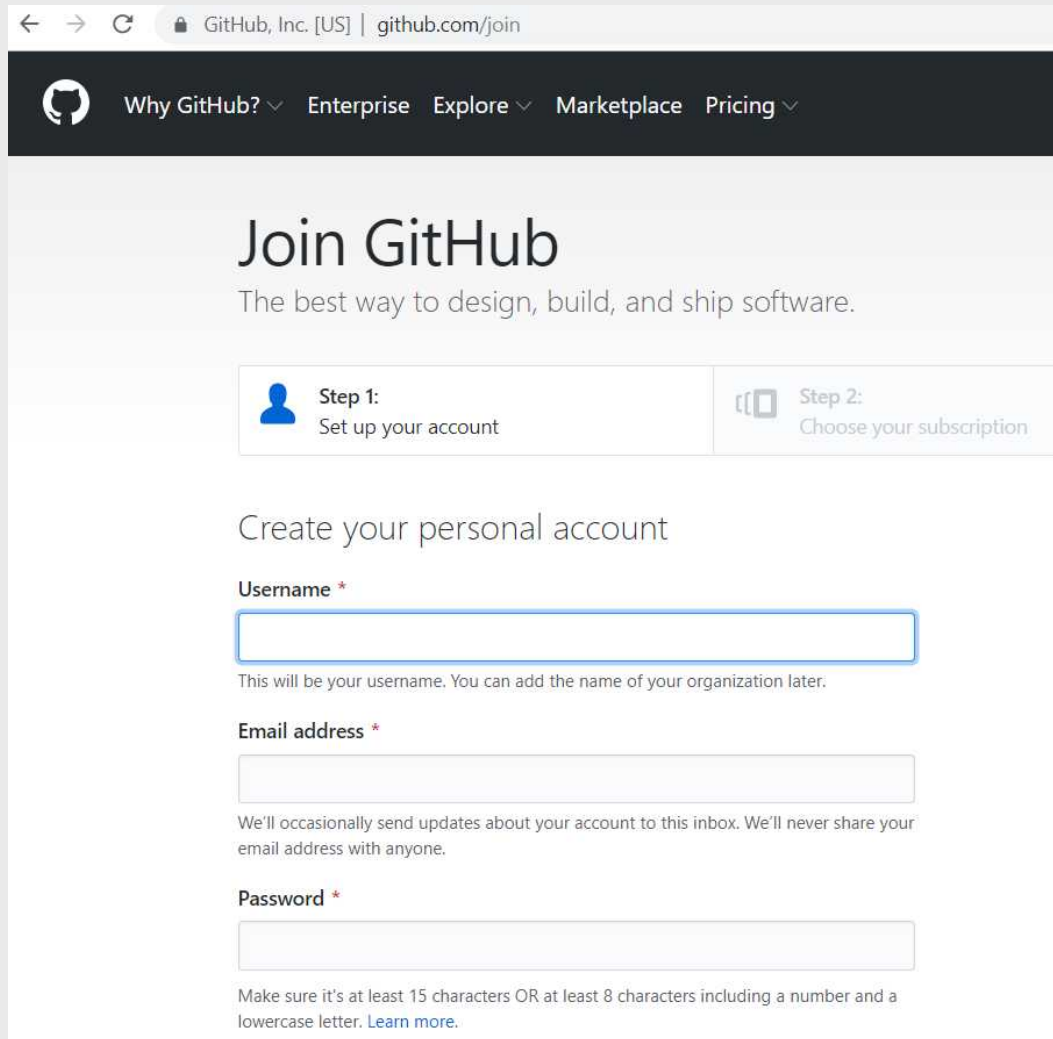
2 - Push a local branch to the remote repository

➤ *git push -u <remote_name> <local_branch_name>*

You can also configure global options such as user name to be used for all commits

➤ *git config --global user.name <name>*

Create a GitHub Account




The screenshot shows the GitHub 'Join GitHub' page. At the top is a navigation bar with the GitHub logo and links for 'Why GitHub?', 'Enterprise', 'Explore', 'Marketplace', and 'Pricing'. Below this is a large heading 'Join GitHub' with the tagline 'The best way to design, build, and ship software.' A progress bar indicates two steps: 'Step 1: Set up your account' (active) and 'Step 2: Choose your subscription'. The main section is titled 'Create your personal account' and contains three form fields: 'Username', 'Email address', and 'Password'. Each field has a red asterisk indicating it is required. Below the 'Username' field is a note: 'This will be your username. You can add the name of your organization later.' Below the 'Email address' field is a note: 'We'll occasionally send updates about your account to this inbox. We'll never share your email address with anyone.' Below the 'Password' field is a note: 'Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter. [Learn more.](#)'


← → ↻ 🔒 GitHub, Inc. [US] | github.com/join

Why GitHub? ▾ Enterprise Explore ▾ Marketplace Pricing ▾

Join GitHub

The best way to design, build, and ship software.

 **Step 1:**
Set up your account

 **Step 2:**
Choose your subscription

Create your personal account

Username *

This will be your username. You can add the name of your organization later.

Email address *

We'll occasionally send updates about your account to this inbox. We'll never share your email address with anyone.

Password *

Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter. [Learn more.](#)

What's GitLab ?

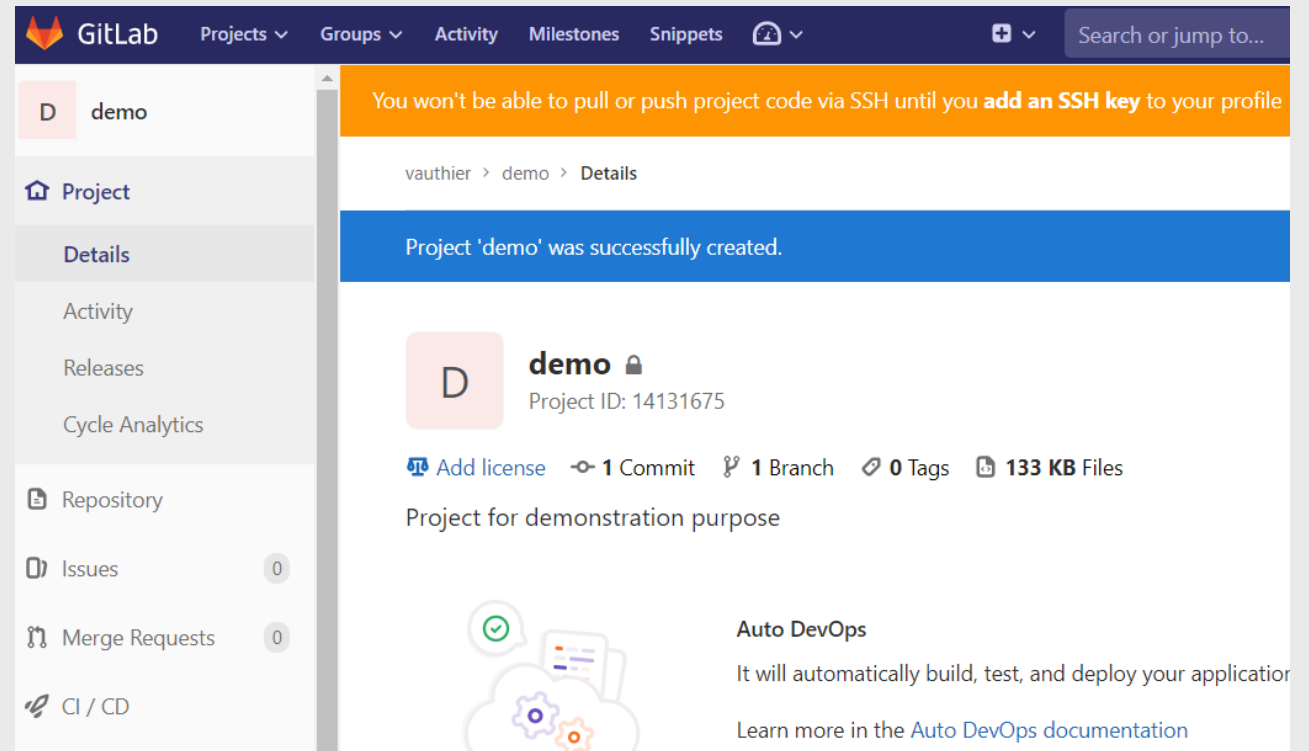
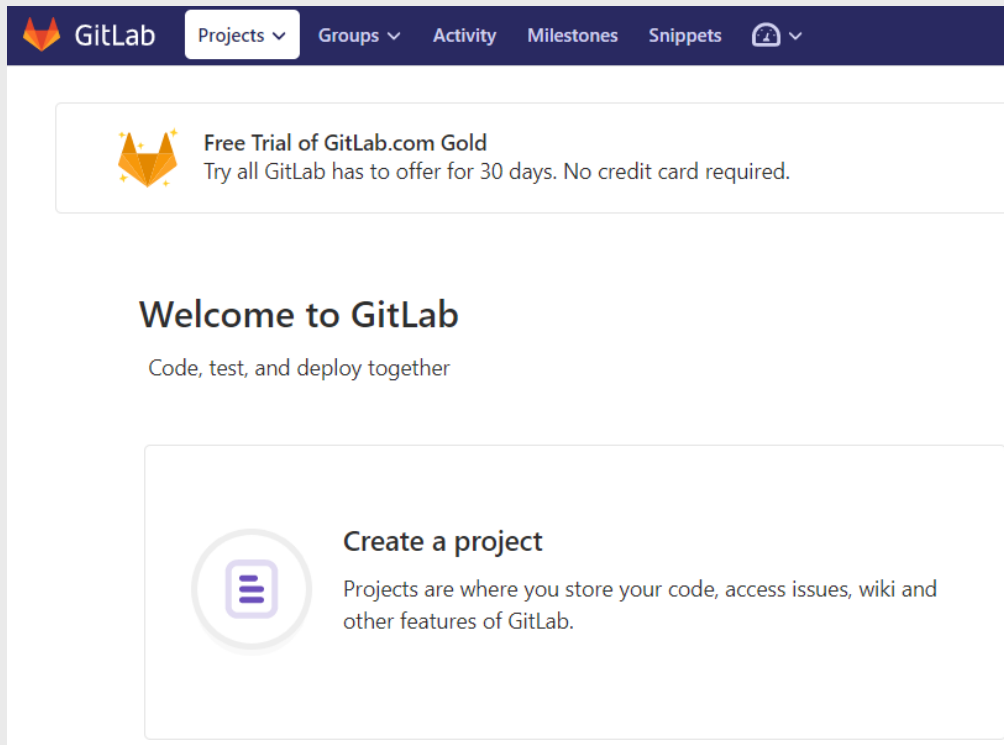
GitLab is built on top of git so that users who are contributing work to a project will have a copy of the project downloaded/checked out/cloned on their local computer.

It provides a web interface for handling many of git's more advanced workflows, and recommends a workflow for interacting with git for the best in productivity, efficiency, and ease of use.

- ✓ File browser to explore the files in your repository
- ✓ A branch viewer
- ✓ A tag viewer
- ✓ Analyze and view the commit graph
- ✓ Web interface to make changes to code and commit it

What is a GitLab project ?

- A Git repository to host the code with branches
- Several features such as issues tracking, CI/CD features....



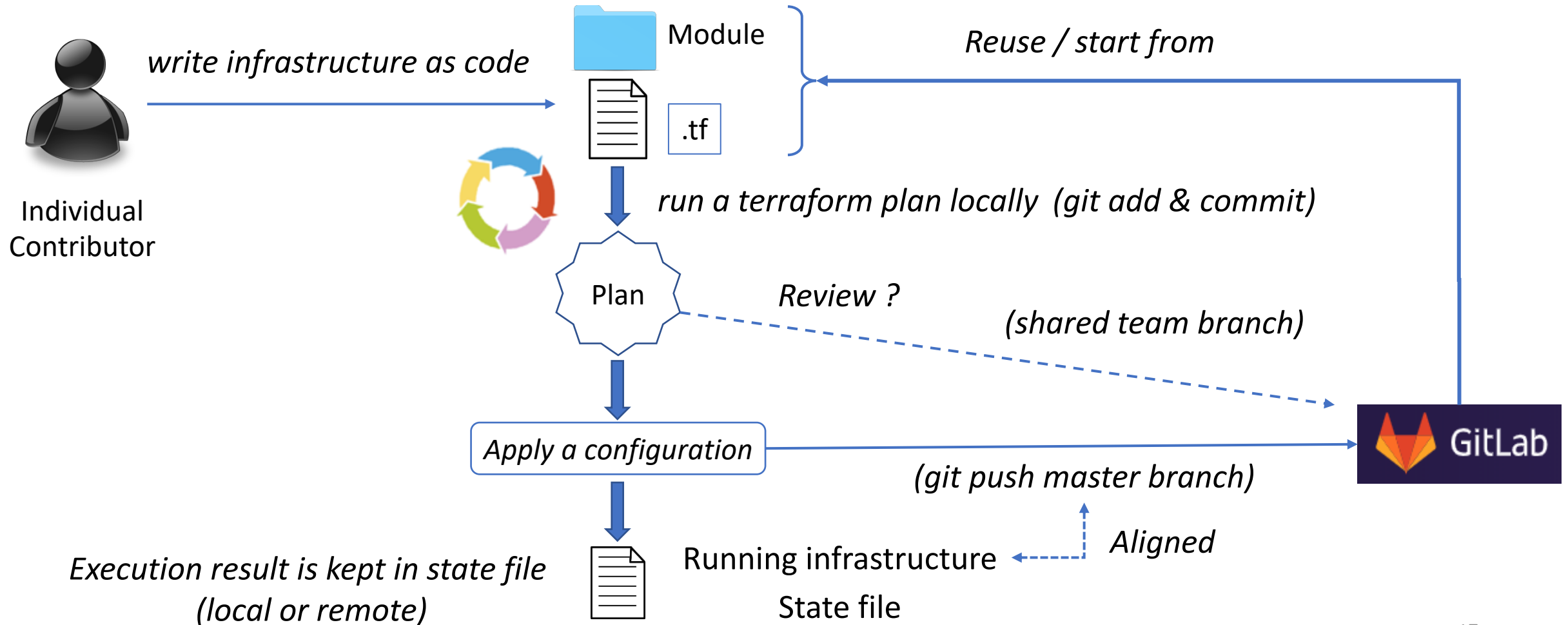
GitLab create a project

The screenshot shows the GitLab web interface for a project named 'demo'. The left sidebar contains navigation links: Project, Repository, Files, Commits, Branches (selected), Tags, Contributors, Graph, Compare, and Charts. The top navigation bar includes links for Projects, Groups, Activity, Milestones, Snippets, and a search bar. The main content area is titled 'vauthier > demo > Repository > Branches'. It features tabs for Overview, Active, Stale, and All. A filter box for 'Filter by branch name' and buttons for 'Delete merged branches' and 'New branch' are present. A message states: 'Protected branches can be managed in [project settings](#).' Below this, the 'Active branches' section shows the 'master' branch as the 'default' and 'protected', with its commit hash '7ad1c315' and the text 'Initial commit · 1 minute ago'. Action icons for cloning and deleting are visible next to the branch entry.

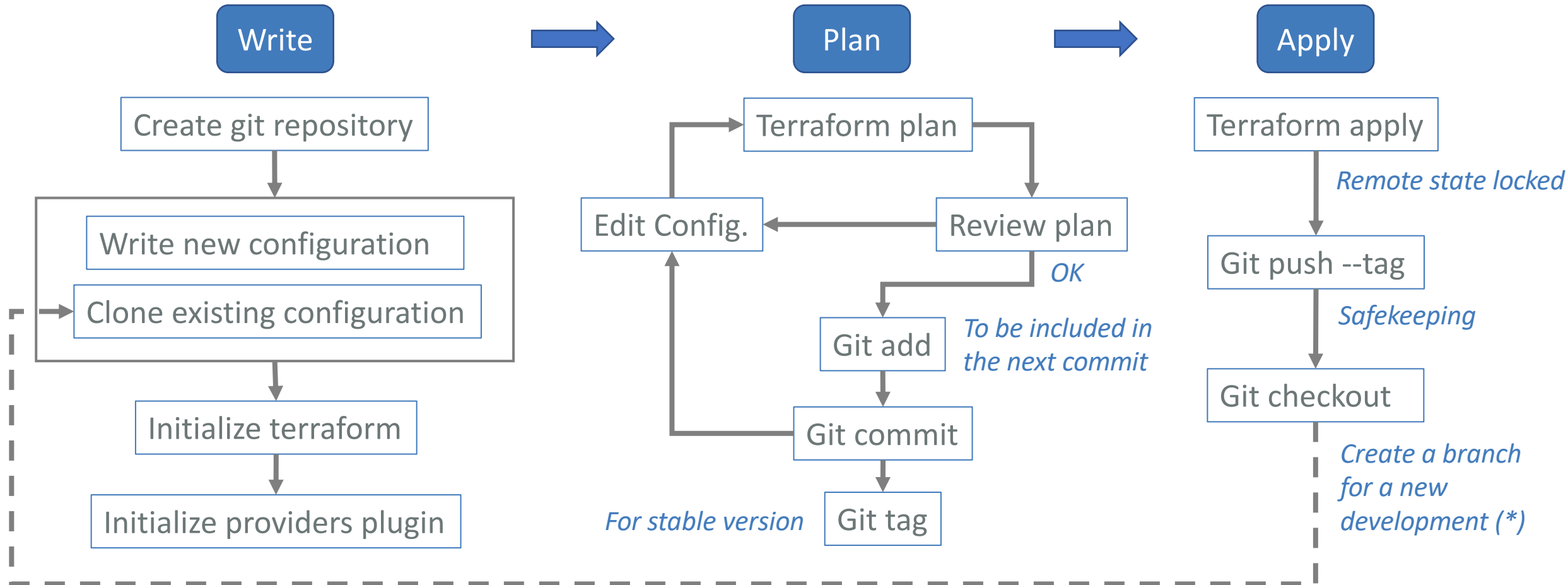
Git, Terraform and Devops

The good the bad and and the ugly

Terraform workflow, working with Git



Terraform workflow, working with Git

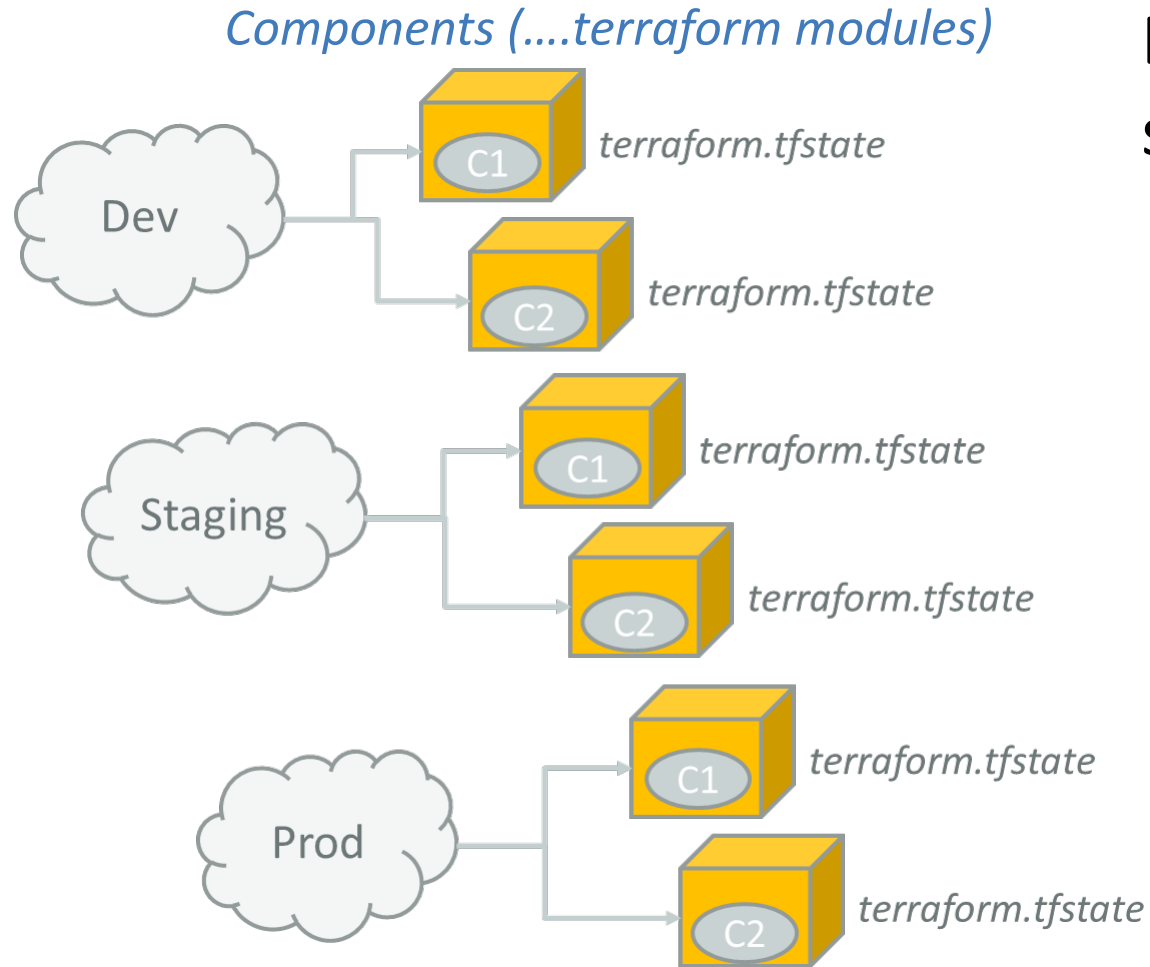


(*) Master is the default branch, but several branches can be used such as release, hot fix, feature dev.,... depending on the context

Troubles with branches

- Locking mechanism provided by Terraform backends avoid changes to overwrite in the terraform state file.... but not in the source code
- If 2 team members are deploying the same code to the same environment but from 2 different branches => conflicts can occur
- For one shared environment (*staging, prod,...*) always deploy from a single branch
- After a plan, review the changes. Then when all the tests are passed merge all your changes in the master branch
- Terraform is a declarative language and control over deployment is limited when applying changes. Be prepared to address efficiently situations where something goes wrong. Always test your changes in a staging environment before deploying in production.

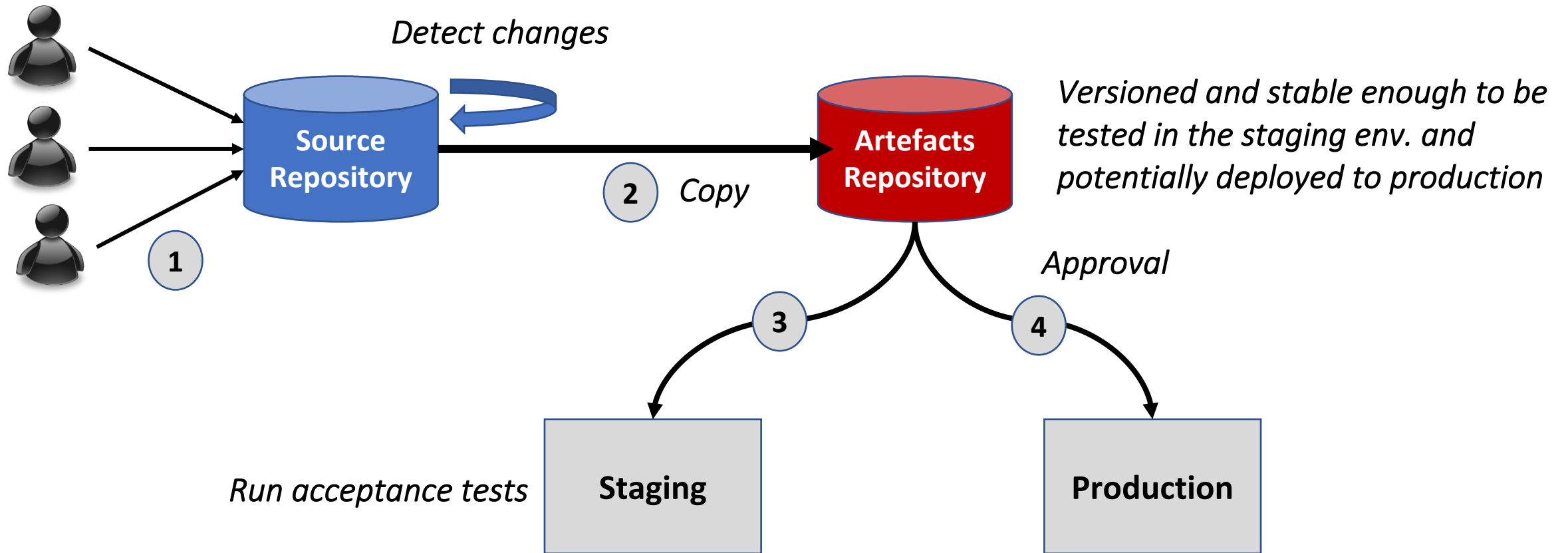
Environments isolation



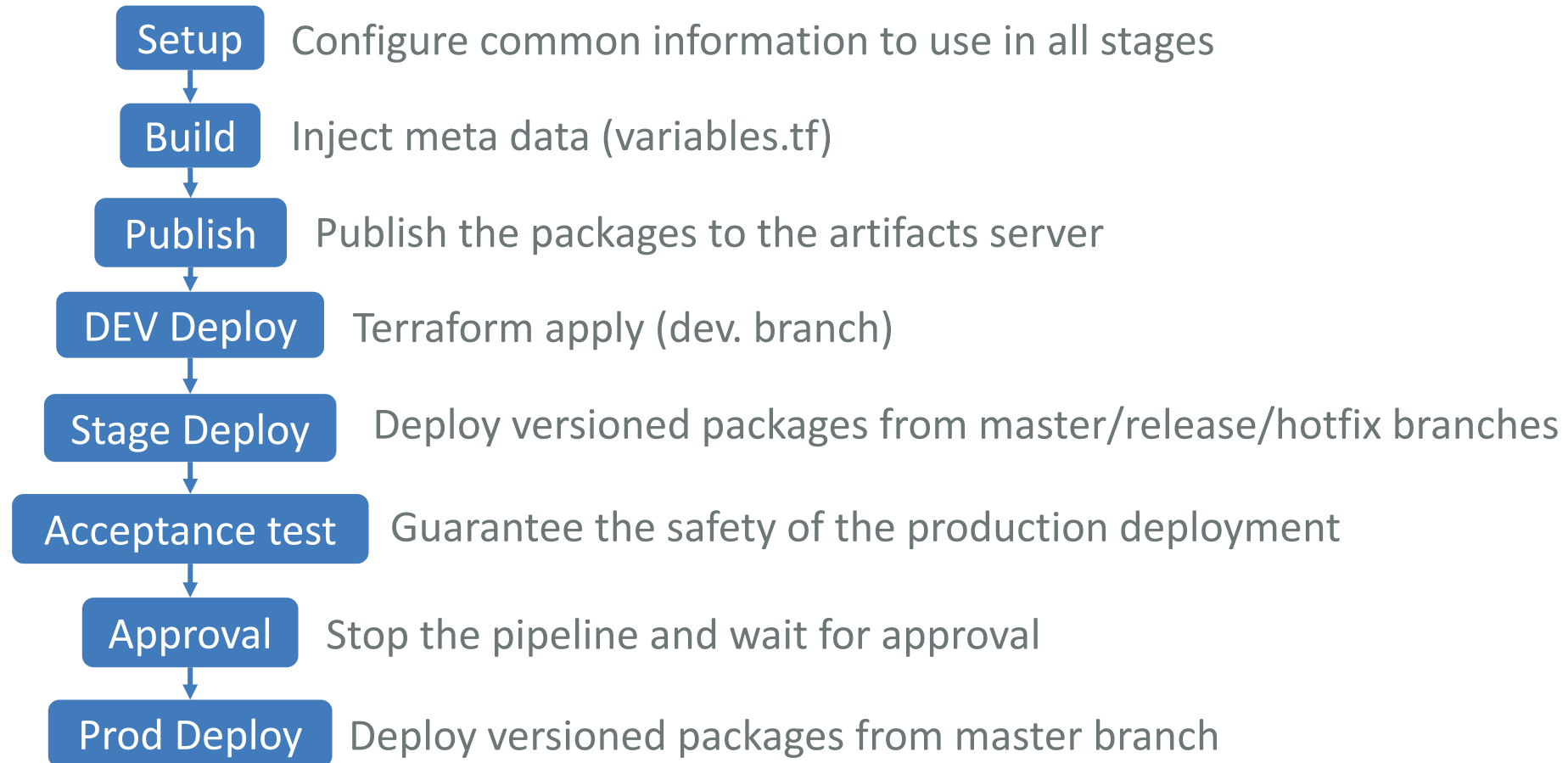
Do not put all your environments in a single stack

- An option could be to have a separate stack for each environment. Drawback: differences creep across environments. Maintaining separate definition required discipline to keep them consistent. For simple env.
- An alternative is to use a continuous delivery pipeline across environments. Promote component definition through a pipeline (parametrized).

Define your pipeline

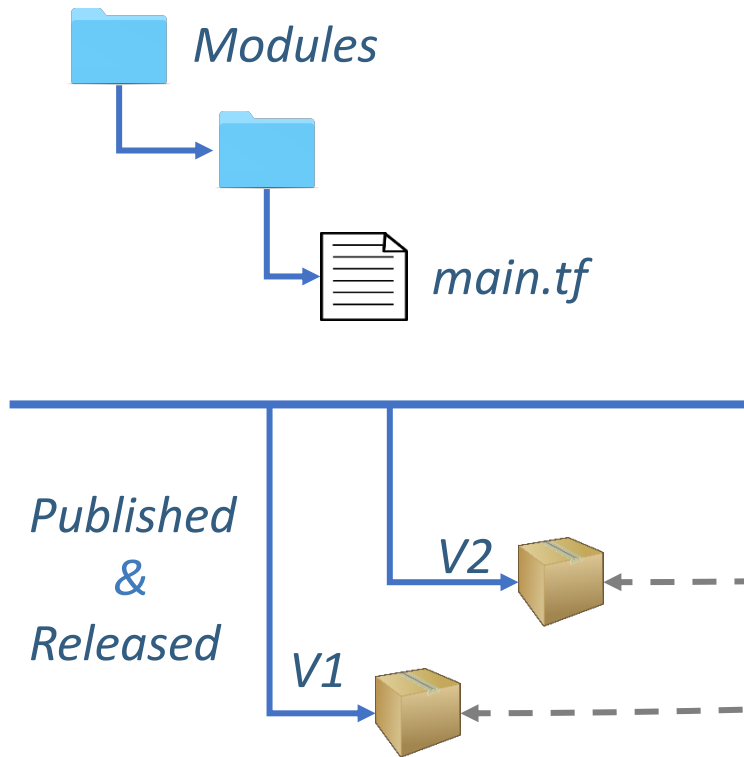


Define your pipeline

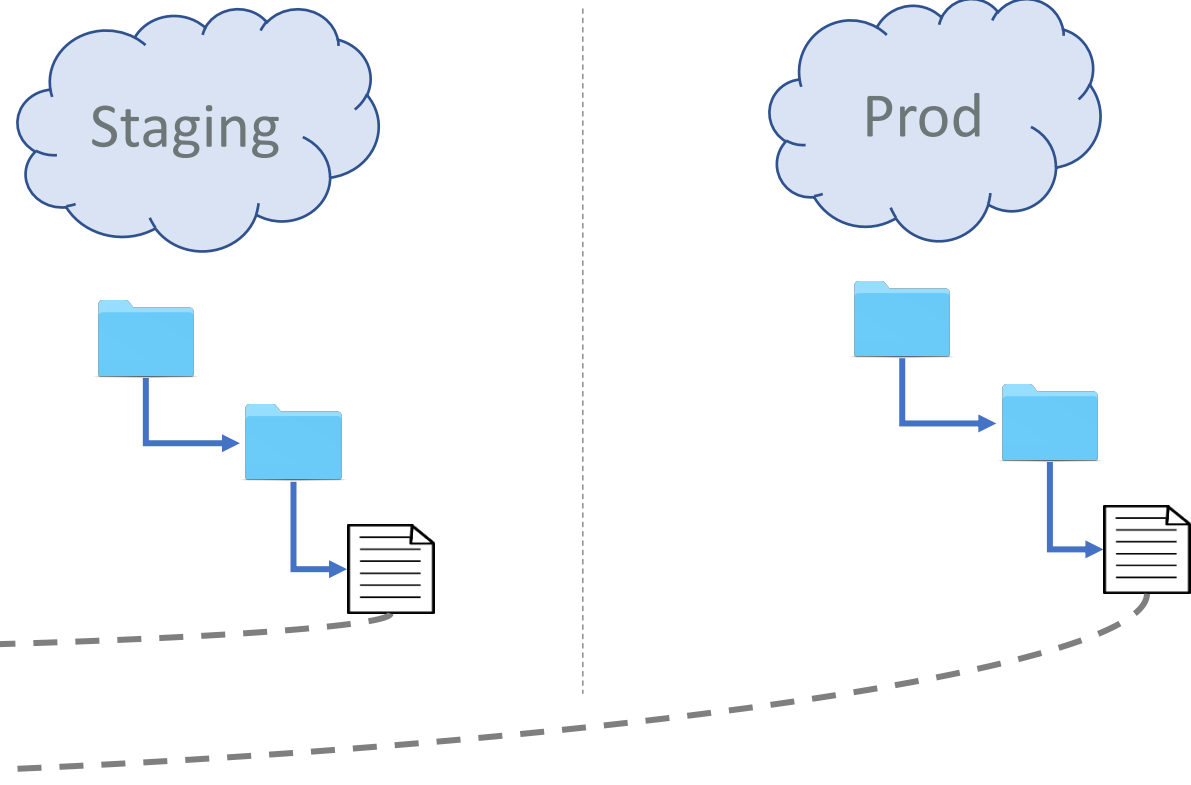


Reusing terraform modules

You need a repository for your modules



You need a live repository for infrastructure deployed in each environment



Some references

- ✓ <https://www.terraform.io/guides/core-workflow.html>
- ✓ Terraform: Up and Running, Yevgeniy Brikman

