

# Introduction to Git



Introduction to Version Control Systems

Git Fundamentals

Branching

Merging

Collaborating

Common Workflows

Git and Ansible and Kubernetes(k8)

Advanced Topics

Hands on Exercises



# Version Control Systems

# Why Do We Care About Version Control Systems

**Enables all modern practices (Infrastructure as Code, CI/CD, 12 factor, k8)**

- Revert production changes quickly (think outages)
- Creates reusable code instead of reinventing the wheel
- Provides a method for reverting to a prior state
- Tracked Changes (Who, What, When, Where, Why)
- Single Source of Truth
- High Availability and Disaster Recovery



# TL;DR

## The 10 most important commands to know:

1. `git clone` - make a local copy of a remote repository
2. `git add` - adds a file to your local repository
3. `git checkout` - checks out a branch ("`git checkout -b`" creates a new branch)
4. `git commit` - does a local commit of changes so you can push them to remote
5. `git push` - pushes commit to remote repository
6. `git merge` - merges a branch into another (usually used to merge into main/master)
7. `git stash` - saves changes without a commit and reverts to last commit state
8. `git log` - shows log of commits ("`git log --oneline --graph`" makes it human readable)
9. `git rebase` - moves entire `feature` branch to begin on the tip of `main/master` branch
10. `git diff` - shows differences between branches

# Why Version Control Systems

**Management of changes to resources so they can be recalled at a later point**

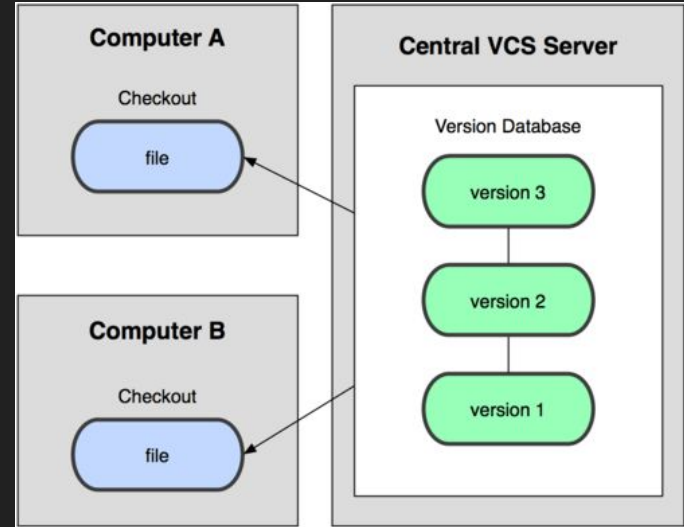
- Records what changed and who changed them
- Enables multiple individuals the opportunity to collaborate
- Provides a method for reverting to a prior state
- Makes code reusable
- Standard for all code storage for Automation, Kubernetes and Infrastructure as Code



# Types of Version Control Systems

## Centralized Version Control

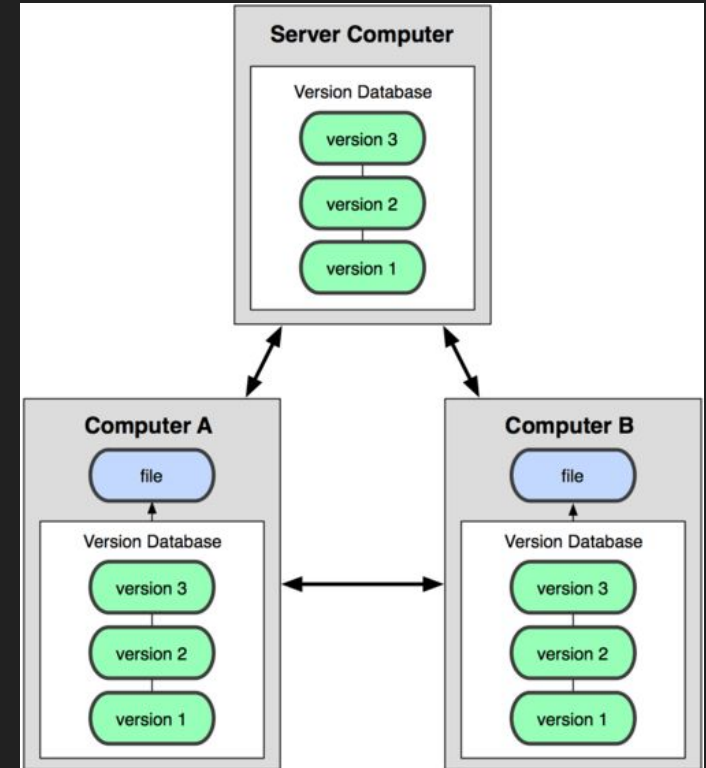
- Single copy of the project hosted on a central server
- Changes are made (“Committed”) against the server
- Server contains full project
- End user never has a full copy of the project
  - Only downloads what is needed
- Common VCS types:
  - CVS
  - Subversion
  - Perforce



# Types of Version Control Systems

## Distributed Version Control

- No reliance on a centralized server
- Each user has the full copy of the entire project
- Changes are made locally
  - No network dependencies
- Users can exchange changes directly with others
  - Most common approach is to use an agreed upon location the each member can reference
- Common types:
  - Git
  - Mercurial





# Git

## ***Git is an Open Source Distributed Version Control System***

- Created by Linus Torvalds (Yes.... the very same)
  - Initial use case: Managing Linux Kernel
- Written in C

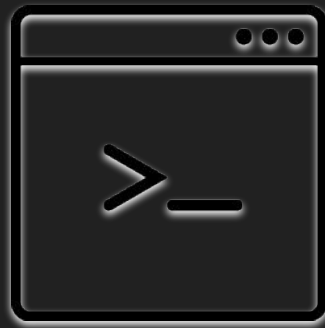
*Think of git as something that sits on top of the file system and manipulates files*



# Git

## Git is a binary

- Executable file (git)
- Executable installed by:
  - yum install git
  - Direct download
    - [git-scm.com](https://git-scm.com)



# Git

## Git has online presence

- Several popular online repository managers available
  - Provides a centralized git repository
  - Additional features found in many providers:
    - Repository viewer
    - Issue tracking
    - Task management
    - Wiki
- 
- Features provided by these services not core git functions



GitLab



GitHub

# Git Fundamentals

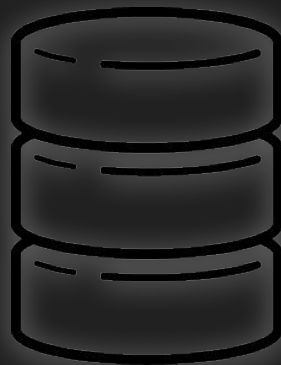
# Git Key Concepts

## Repository

Data structure that stores the configuration of files that change over time

- Collection of files and their history organized in branches, tags, etc
- Can be either local or remote
- To work with Git, a repository must exist somewhere
- New repositories are empty by default
- Creating a new repository

```
$ cd myproject  
$ touch README  
$ git init
```



# Git Key Concepts

## Setting the Stage

Those with prior VCS experience may have knowledge of *committing*, but not **staging**

- In Git, we are constantly dealing with **changes** to files and we only care about the **lines** that changed



# Git Key Concepts

## Commit

Fundamental concept behind git

- A commit is just a node in a tree
- Represents a snapshot in time
- Node contains:
  - a **H**ash of the commit -- a UUID
  - the **A**uthor
  - a **D**ate and time
  - the commit **M**essage
  - the diff of changes



```
$ git add README.md  
$ git commit -m "Added README"
```

# Git Key Concepts

## Viewing a Commit

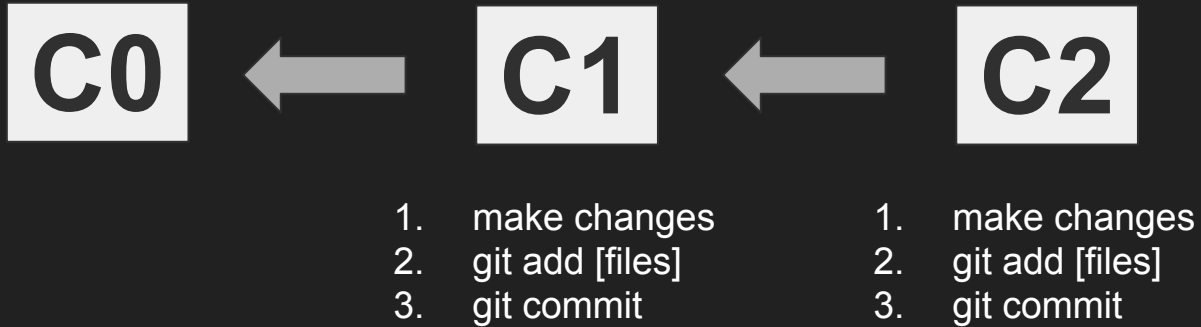
`git log` - Command for viewing commits

```
H → $ git log  
A → commit 09b129280c2b16c4d926d9b23a1ef544e8470936  
D → Author: John Doe <jdoe@exxonmobil.com>  
M → Date: Wed Sep 12 18:03:08 2018 -0600  
  
created README
```



# Git Key Concepts

## Making More Commits



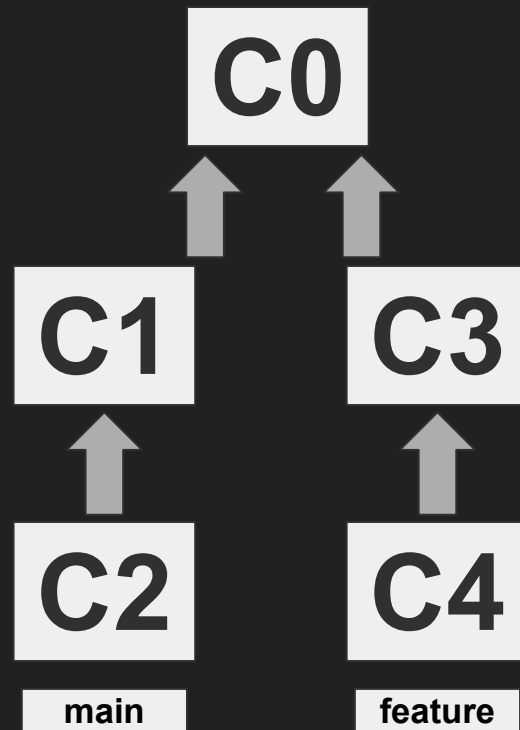
# Branching and Merging

# Branching and Merging

## Branches

Lightweight movable pointer on a commit

- Branching model is one of the best capabilities of git
- In Git, you are **always** working on a branch
  - **main** is the default branch (typically long-term stable)
  - Each branch is given a name
- Only **one** branch can be active at a given time
  - Signified by HEAD
- Allows for features to be created

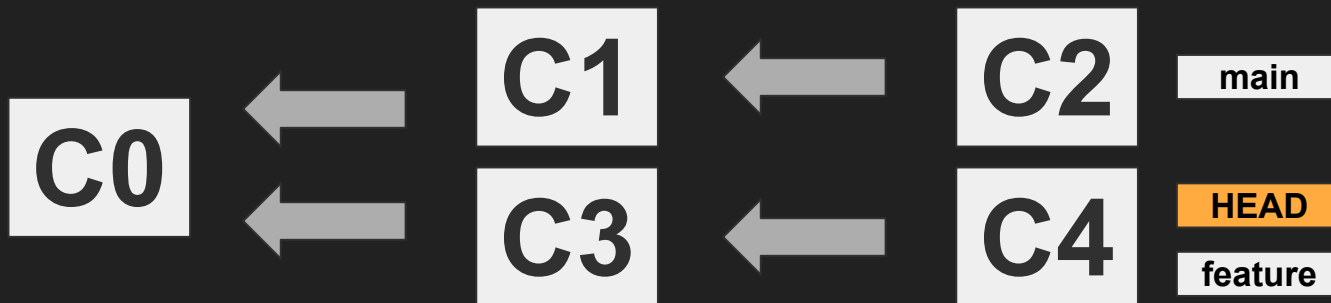


# Branching and Merging

## Branch Operations

Commands for managing branches

- `git branch` - List branches
- `git branch <branch name>` - Creates a new branch
- `git checkout <branch name>` - Switches to new branch



# Branching and Merging

## Merging

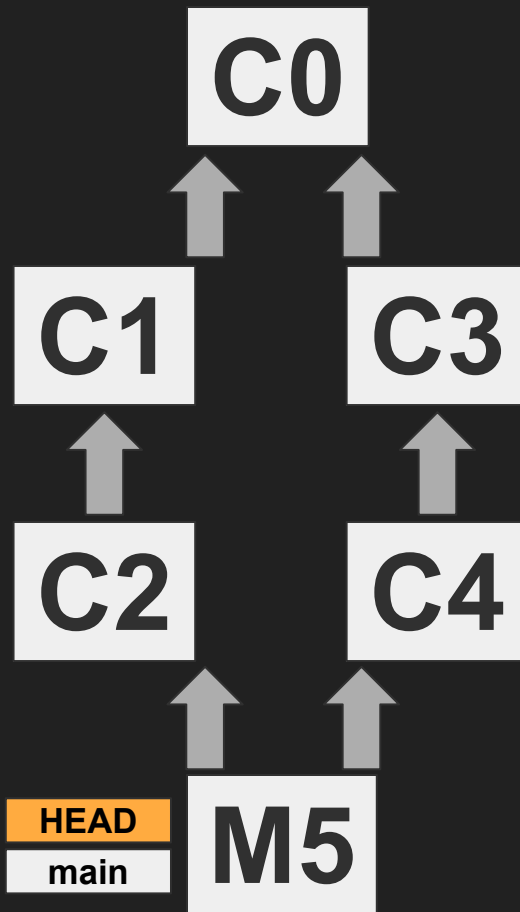
Integrates all commits from a specific branch into the current branch (HEAD)

- `git merge` command

```
# Checkout main branch  
$ git checkout main
```

```
# Merge feature branch into main branch  
$ git merge feature
```

- M5 is a **merge commit**



# Collaborating

*Showcasing the true power of Distributed Version  
Control Systems(DVCS)*

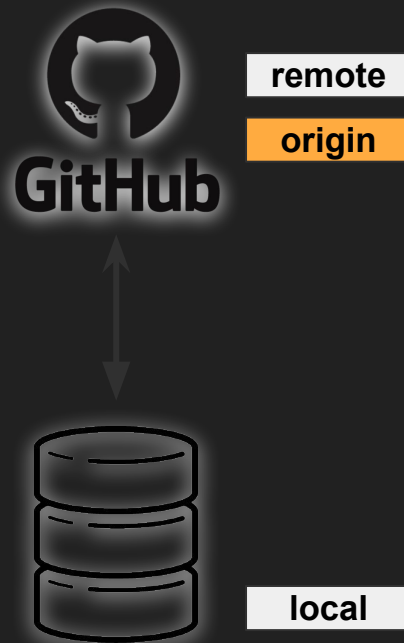
# Collaborating

## Remotes

Enables sharing code with other locations

- Versions of the project hosted on the internet or somewhere else
- Enabled through the `git remote` command

```
# Adding a remote  
$ git remote add origin git@github.com:ansible/ansible.git
```



# Collaborating

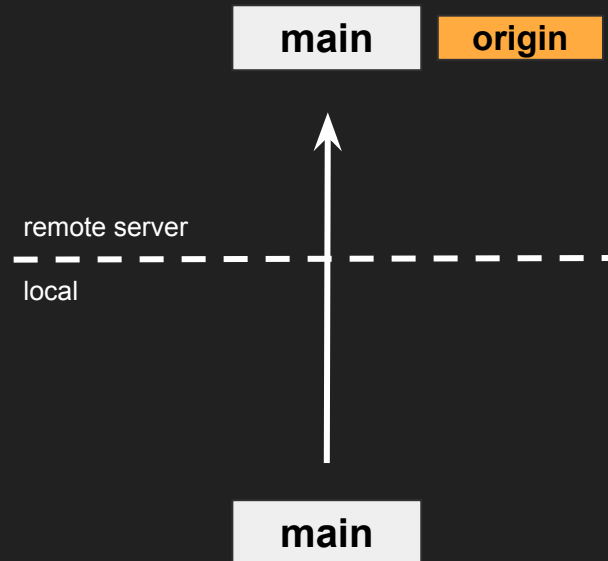
## Pushing

Upload code from local repository to remote repository

- Merge local changes into the remote branch
- An existing remote must be defined

```
$ git push <remote> <branch>
```

- **branch** is **main**





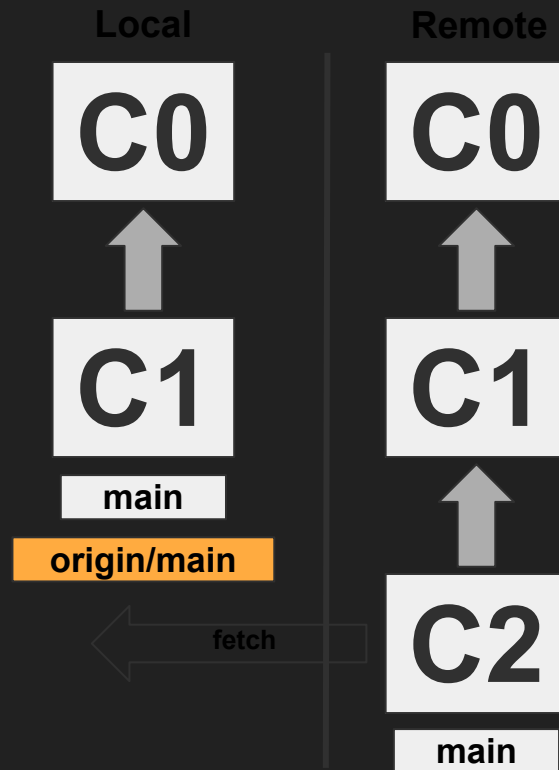
# Collaborating

## Fetching

Retrieve changes from remote repositories

- Brings in changes from other contributors
  - Changes are retrieved, but does not modify local workspace
  - Creates new branch locally (<remote>/<branch>)
  - Afterward, a merge needs to occur to integrate
- 
- Single branch of all branches can be retrieved

```
$ git fetch <remote> <branch>
```



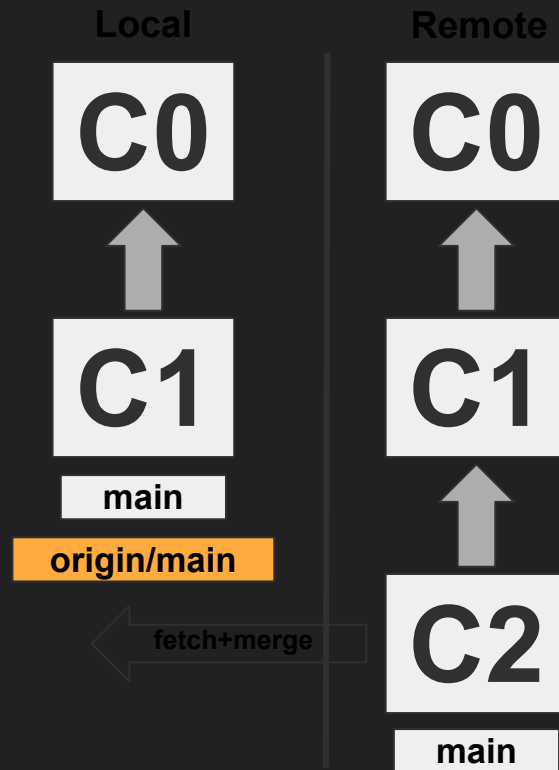
# Collaborating

## Pulling

Streamlined method of retrieving changes from remote repositories

- Performs a fetch and merge

```
$ git pull <remote> <branch>
```



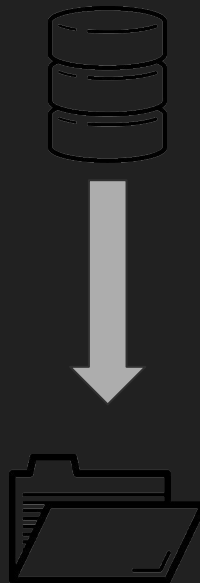
# Collaborating

## Cloning

Retrieves a full copy of a remote repository

- Unlike other VCS tools, the entire content is retrieved

```
$ git clone <url>
```



# Workshop 1 Clone, Commit, Push, Merge

[AzureDevOps]

<https://redhat-adsa.github.io/GitWorkshop/ado/02-advanced.html>

[Bitbucket]

<https://redhat-adsa.github.io/GitWorkshop/bitbucket/02-advanced.html>

[Github]

<https://redhat-adsa.github.io/GitWorkshop/github/02-advanced.html>

[Gitlab]

<https://redhat-adsa.github.io/GitWorkshop/gitlab/02-advanced.html>

# Git Workflows

# Git Workflows



**Workflows in Git are guidelines and not structured rules**

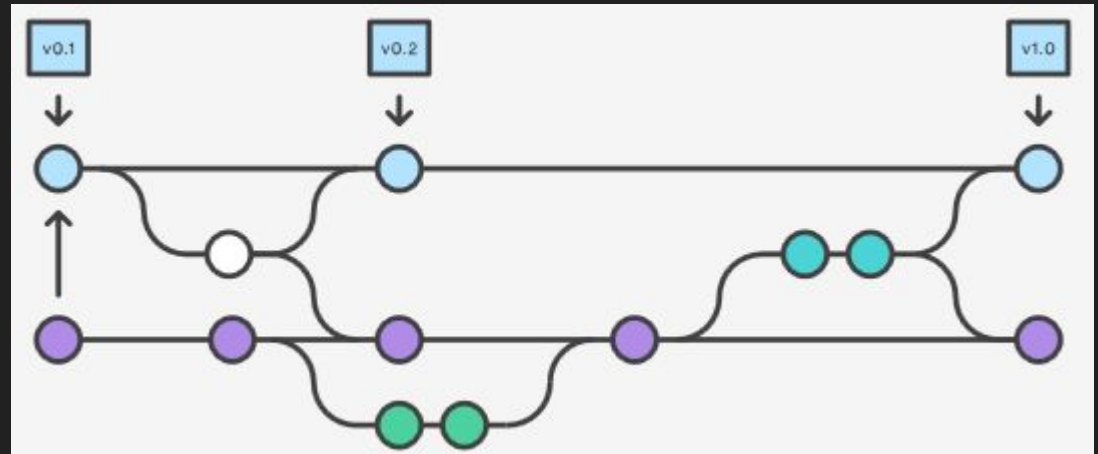
# Git Workflows

## Types of Workflows

Methods for working and collaborating with git

- **Basic/Centralized**
- **Feature Branch**
- **Pull/Fork**

Plus several others:  
[Martin Fowler Blog](#)



# Git Workflows

## Basic/Centralized Workflow

Simplified method where all changes are made against the same branch

- Similar structure as a Subversion repository
- All changes made against the **main** branch
  - SVN equivalent to *trunk*
- Ideal for small teams



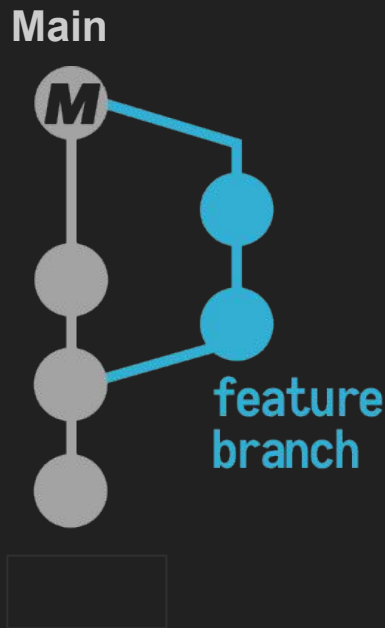


# Git Workflows

## Feature Branch

Separate branches created for each enhancement

- **main** branch represent stable state of repository
- Each developer works on their own separate feature
- Features can be shared without disrupting **main**
- Code review through pull request mechanism prior to integration

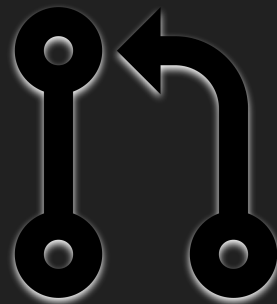


# Git Workflows

## Pull/Fork

Each developer has their own copy (fork) of the repository that they work on

- No centralized repository for pushing changes
- Only project maintainers have access to push to official repository
- Maintainers accept changes from contributors to official repository
  - Pull/Merge request
- Embraces security and distributed nature of Git



This is a common type in GitLab and GitHub based repositories

# Advanced Git

# Pull/Merge Requests

This is how most modern development collaborates on code.

- Fork code to your own namespace
  - Update code and commit back to your branch
  - Perform a pull request to the upstream repository

Links to each type of repository server documentation on how to do a PR:

- [Azure Devops](#)
- [Bitbucket](#)
- [Github](#)
- [Gitlab](#)

# Pull/Merge Requests

This is how most modern development collaborates on code.

- Fork code to your own namespace
  - Update code and commit back to your branch
  - Perform a pull request to the upstream repository

Links to each type of repository server documentation on how to do a PR:

- [Azure Devops](#)
- [Bitbucket](#)
- [Github](#)
- [Gitlab](#)

# Workshop 2 PR, Rebase, Rest, Diff, Stash

[AzureDevOps]

<https://redhat-adsa.github.io/GitWorkshop/ado/02-advanced.html>

[Bitbucket]

<https://redhat-adsa.github.io/GitWorkshop/bitbucket/02-advanced.html>

[Github]

<https://redhat-adsa.github.io/GitWorkshop/github/02-advanced.html>

[Gitlab]

<https://redhat-adsa.github.io/GitWorkshop/gitlab/02-advanced.html>

# Git Ansible and K8

# Git and Ansible

## Branches by Environment

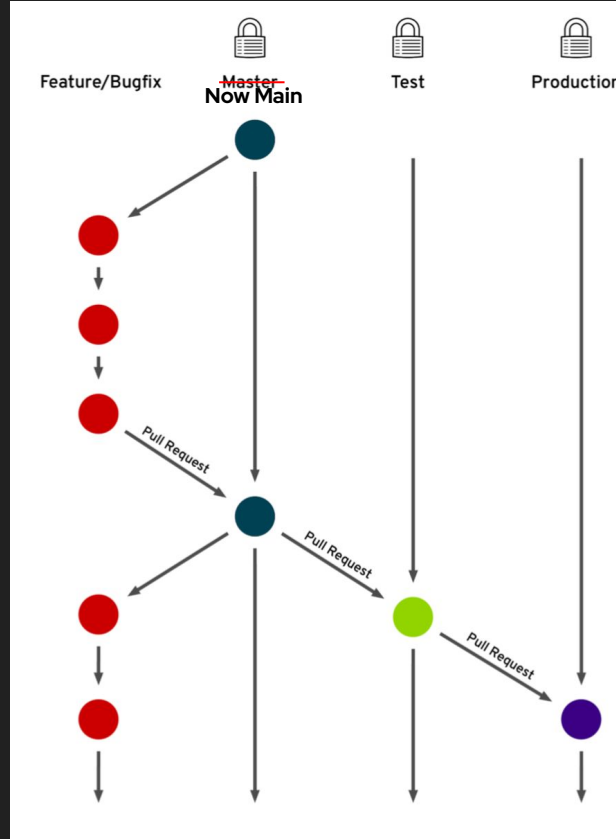
Each branch within a repository represents a deployment environment

- Branches such as dev, test, prod in addition to main
- main branch is starting point. Changes are promoted to upper level environments
  - Bug fixes can be made against each branch as necessary
- Extends concepts emphasized by multiple git workflows
  - Feature branch
  - Pull/Fork
- Example: Ansible Tower projects created per branch. Job templates target each project



# Git Ansible and k8

## Branches by Environment











# Git and Ansible

## Typical Repository Structure

```
playbooks/
├── group_vars
│   ├── all.yml
│   ├── dev.yml
│   ├── prod.yml
│   └── web.yml
├── inventory
├── library
├── roles
│   └── requirements.yml
├── .gitignore
├── ansible.cfg
├── apache.yml
├── deploy-app.yml
├── install-updates.yml
└── site.yml
```

# Git and Kubernetes

## Typical S2I (source to image) Repository Structure

	root
	s2i/bin
	test
	Dockerfile
	Dockerfile.rhel7
	README.md
	cccp.yml
	content_sets.yml