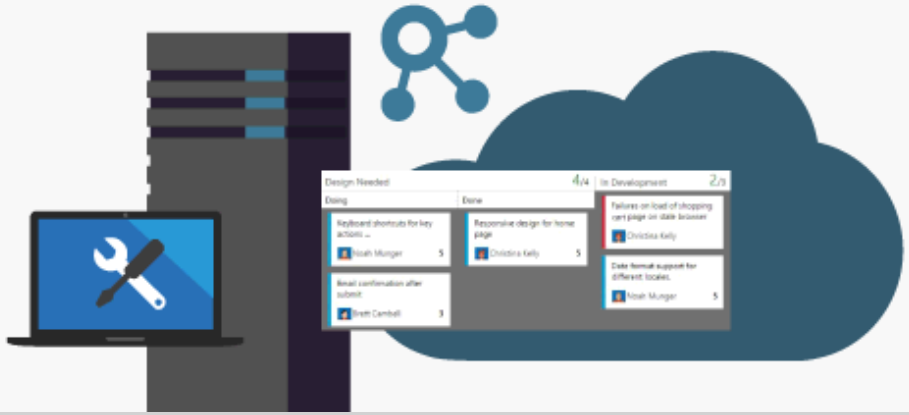
Microsoft

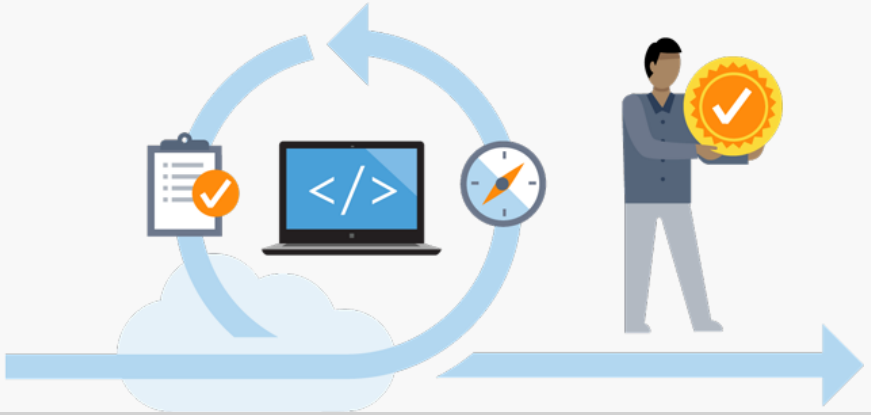
VSTS and Git Basics

Microsoft Services

VSTS and TFS



Team Foundation Server (TFS)

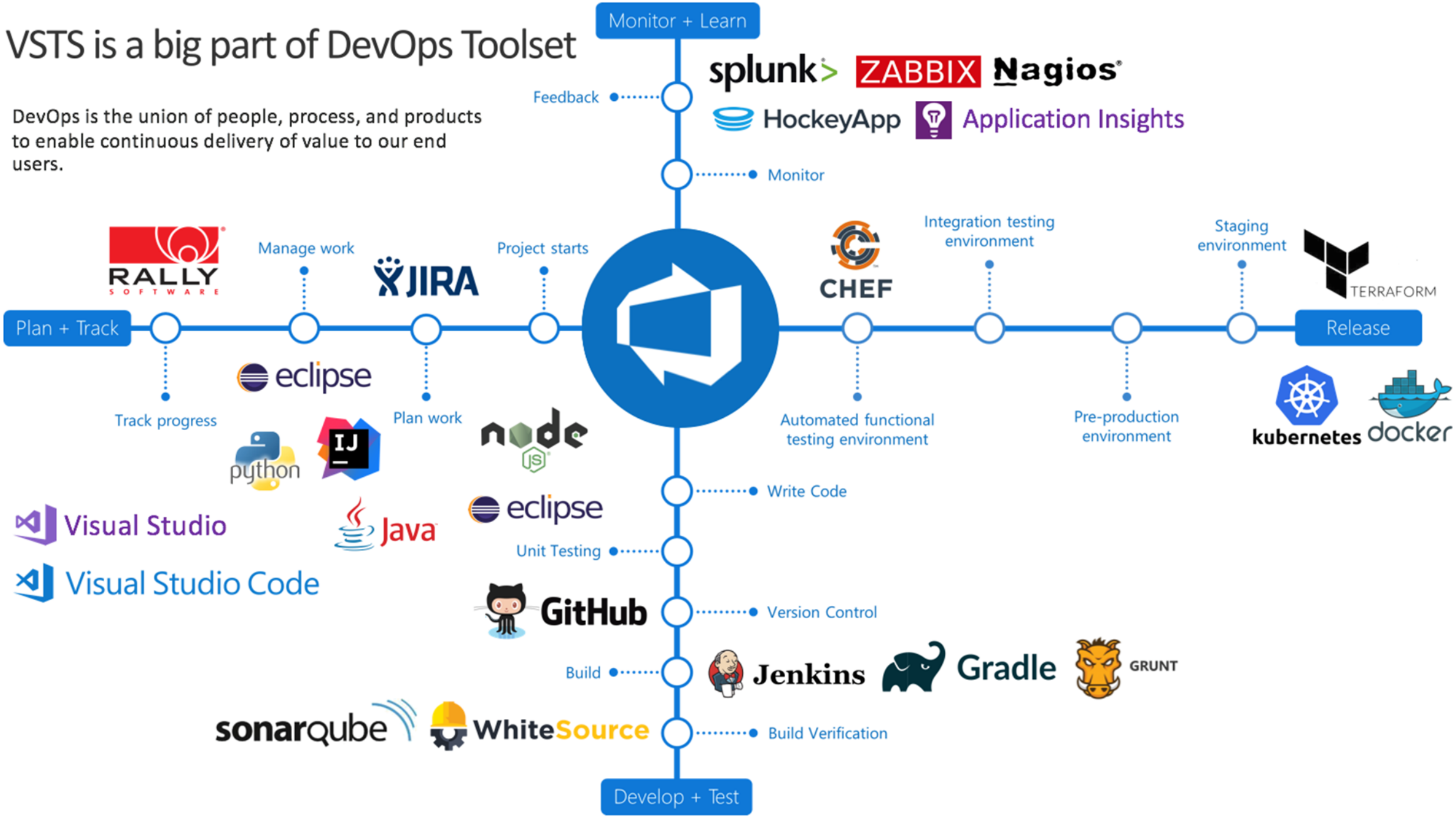


Visual Studio Team Services (VSTS)

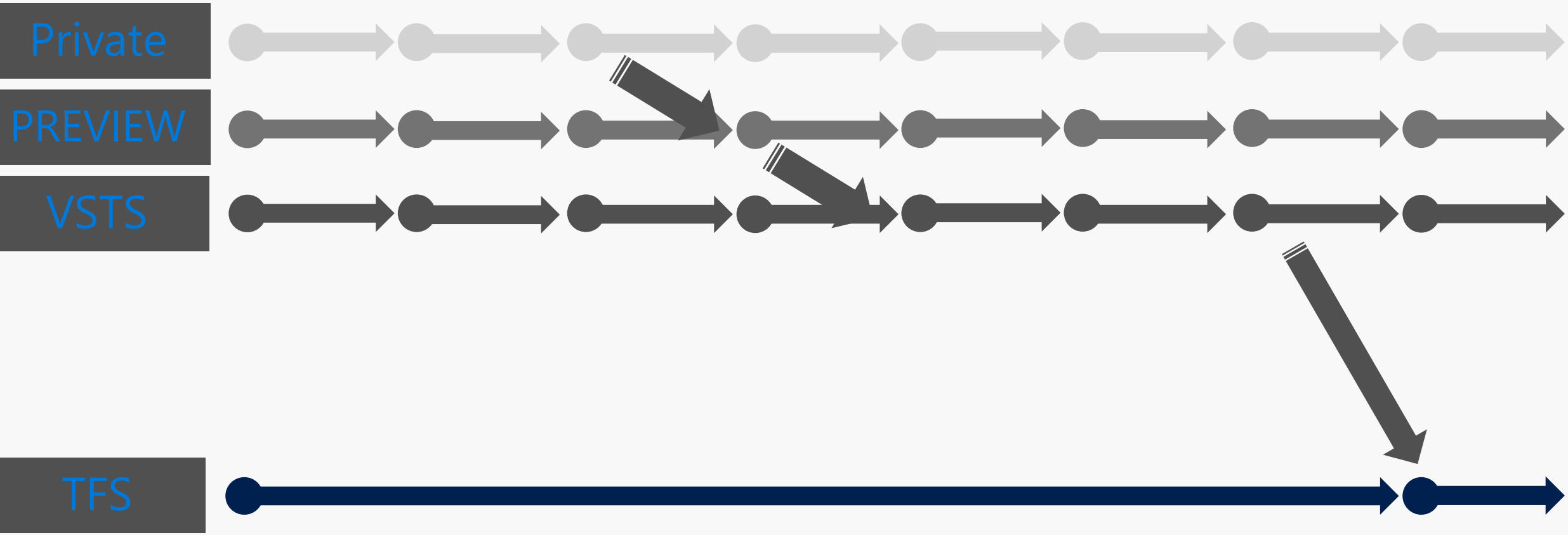
Also known as: Visual Studio Online (VSO)

VSTS is a big part of DevOps Toolset

DevOps is the union of people, process, and products to enable continuous delivery of value to our end users.



Frequent Updates



Demonstration: *Quick Tour of VSTS*

VSTS Capabilities

- Work
- Delivery Plans
- Code
- Dashboard
- Wiki
- Testing

- PREVIEW features

What is the difference between Centralized and Distributed workflows?

Centralized

Based on the idea that there is a single "central" copy of your project on a server.

Each programmer "commits" or "checks-in" their changes to this one true central copy. Client-server approach.

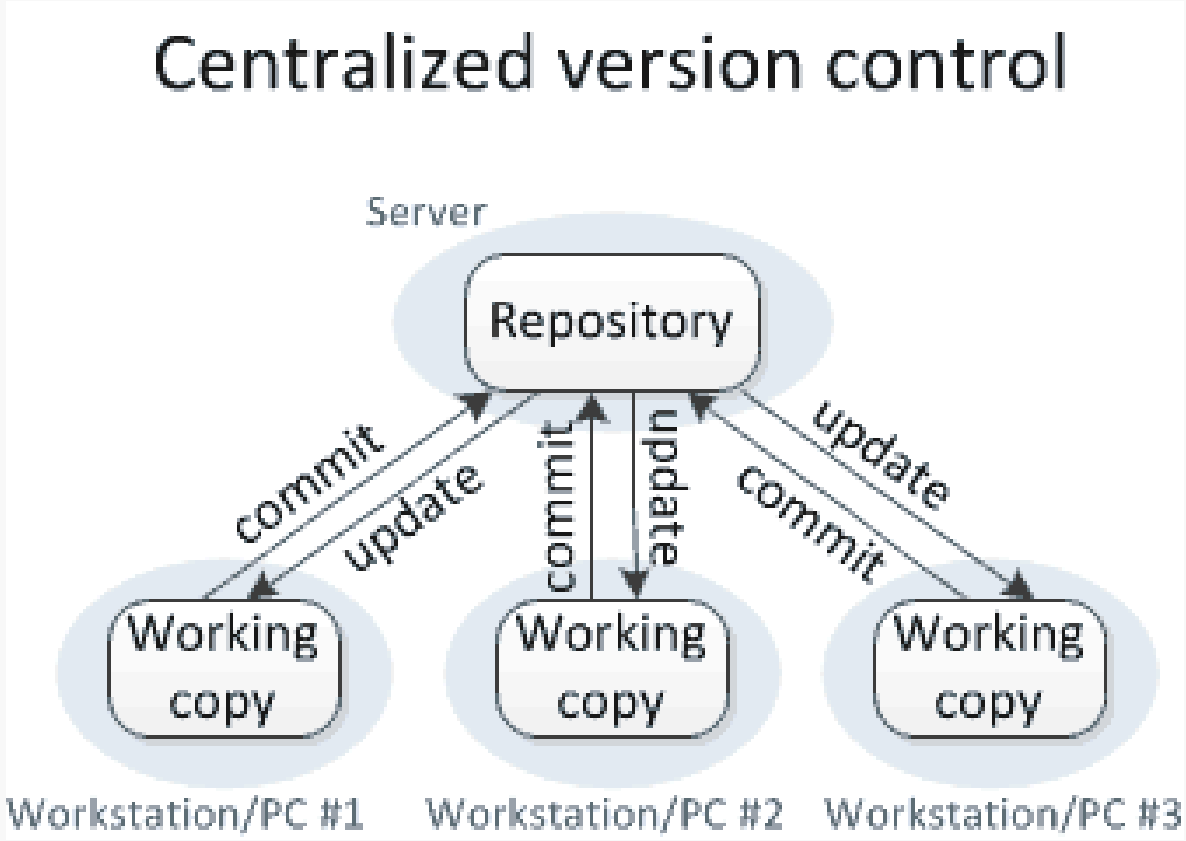
Concurrent Versions System (CVS)

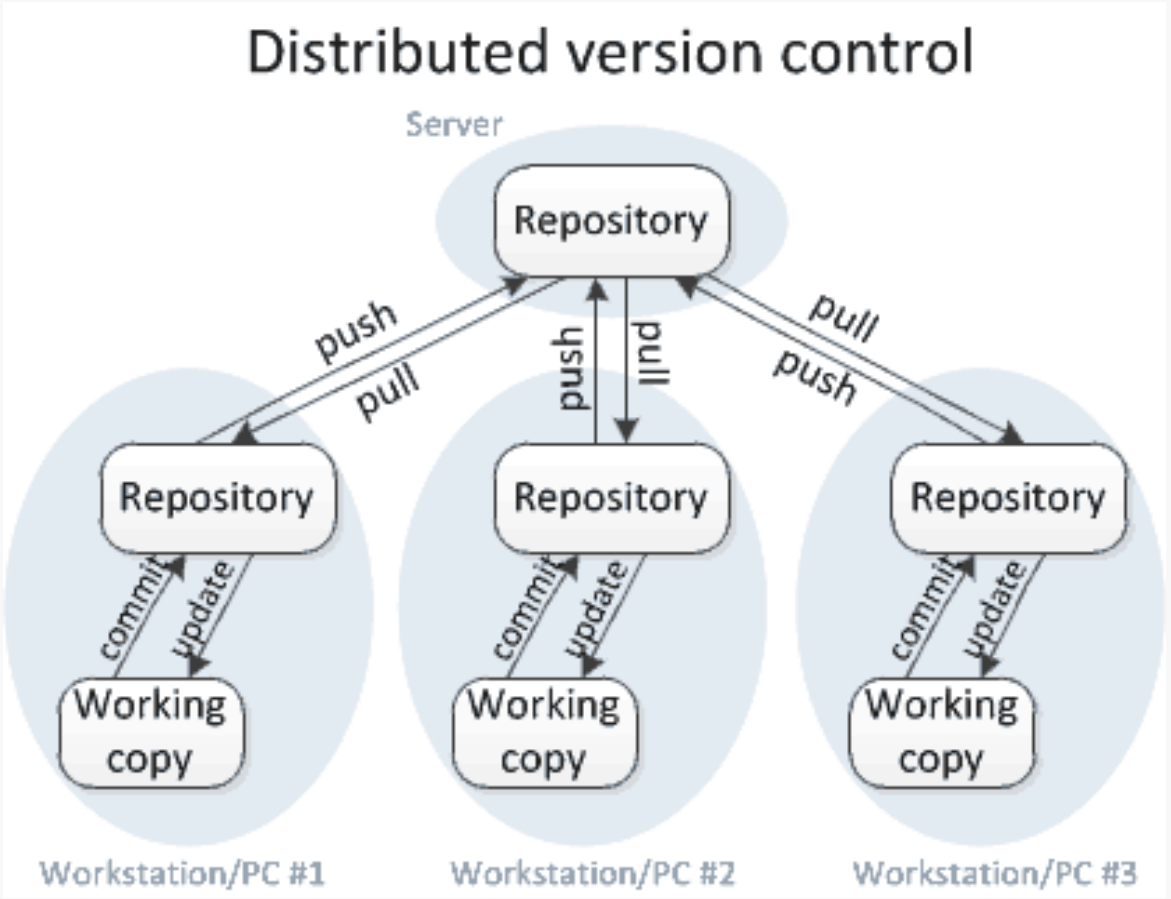
Distributed

No canonical, reference copy of the codebase exists *by default*; only working copies.

Every developer "clones" a copy of a repository and has the **full** history of the project on their own hard drive. This copy (or "clone") has *all* of the metadata of the original.

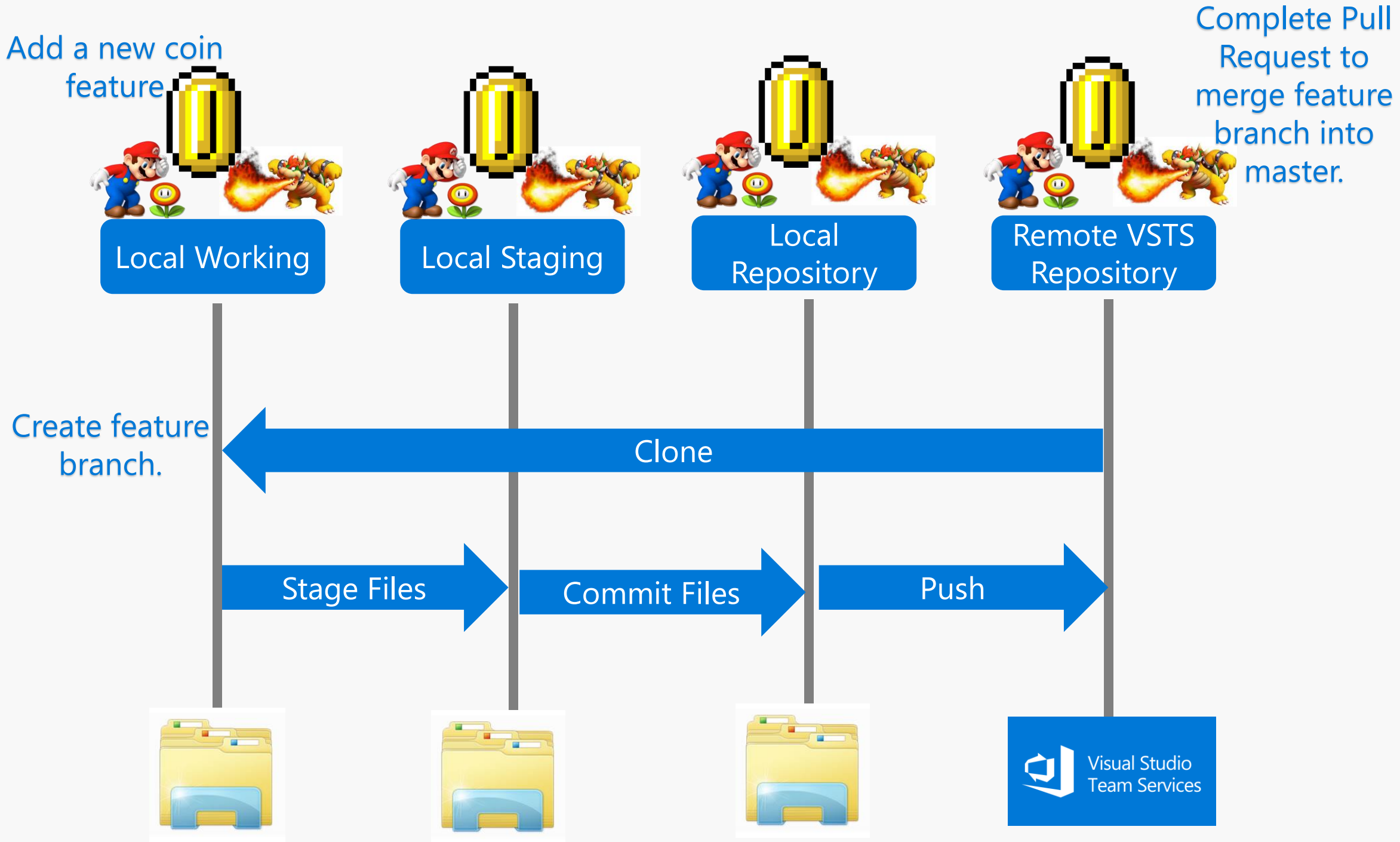
Centralized vs Distributed Source Control





Centralized vs. Distributed Comparison

	Strengths	Best for	Disadvantages
Centralized Version Control (CVC)	• Fine level permission control • Allows usage monitoring • Easy setup	• Large integrated codebases, long history, many binary files • Control and auditability down to the file level	• Single point of failure • Remote commits are slow • Merging can be difficult • Offline is a challenge. Committing / viewing history requires repo access
Distributed Version Control (DVCS)	• Fast offline experience. Complete repository with portable history • Pull Requests model for reviewing code • Branching and merging is easy and extremely fast	• Modular codebases • Open source projects • Highly distributed teams • On the go teams, everything can be done without Internet except push/pull	• Many large binary files could impact performance • Long history 100k+ changesets could take a lot of time and disk space, performance hit • Learning curve to adopt


Add a new coin feature
Local Working
Local Staging
Local Repository
Remote VSTS Repository
Complete Pull Request to merge feature branch into master.
Create feature branch.
Clone
Stage Files
Commit Files
Push
Visual Studio Team Services

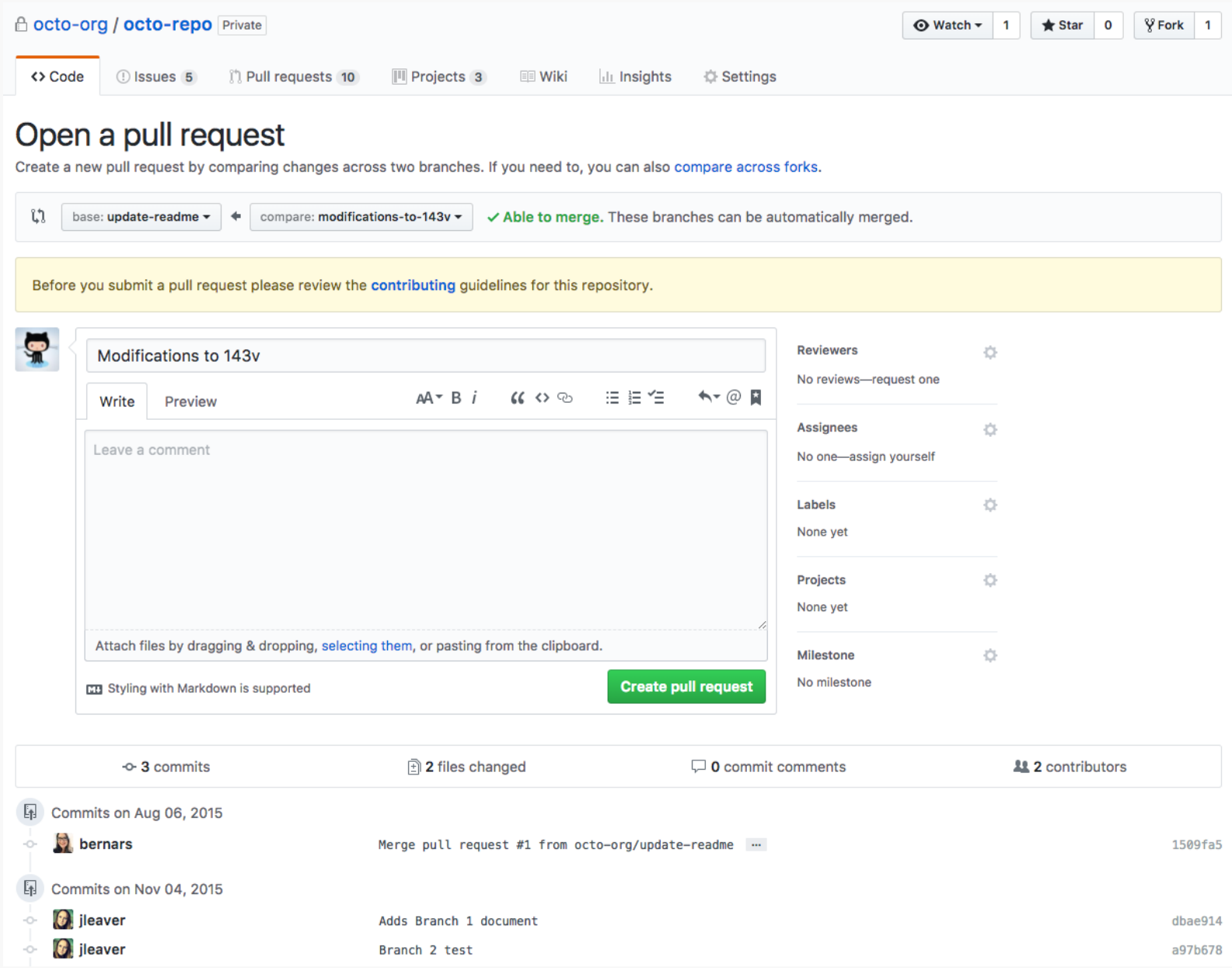
Git Pull Request

Pull Request

Pull requests let you tell others about changes you've pushed to a repository.

It is basically a request to ‘ask to put your feature branch into the main branch’.

Once a **pull request** is sent, interested parties can review the set of changes (code review time!), discuss potential modifications, and even push follow-up commits.



Demonstration: *Git Basics- Adding a Project and Making Changes*

Git Walkthrough on VSTS using VS

Git Walkthrough on VSTS using Git Command Line

Git Terminology Slide

- Fetch – downloads data about commits to your branch, remote branches, and history from the remote repository. Does not merge or affect your code in any way, until you do a ‘pull’.
- Pull – does a fetch and a merge of changes from a remote repository into your current branch.
- Add/Stage – moves a change in the working directory to the staging area
- Commit – records changes to your local repository, doesn’t affect remote copy at all
- Push – transfer commit(s) to a remote server

- Sync – Fetch, Pull, and Push (VS Shortcut)

Git Enterprise Workflows

Centralized Workflow

- **Summary**: Very similar to SVN/TFS. The master branch is the single point of entry and there are no feature branches.
- **Use Case**: Simple transition state from centralized flow. Can work well for 1-3 devs doing off a little side project. Can work for 1 dev in an enterprise maintaining an old project.
- **Pros**: easy to use and learn, very simple non-encumbered fast process. In enterprise, good for 1 dev maintaining an old project, since feature branches and PR's are overkill. Works just like SVN/TFS, but has the benefit of each dev pulling down the full history of the project, committing as often as they like locally, and no central point of failure.
- **Cons**: Not good for large projects, multiple devs, doesn't take advantage of branching/PR's.

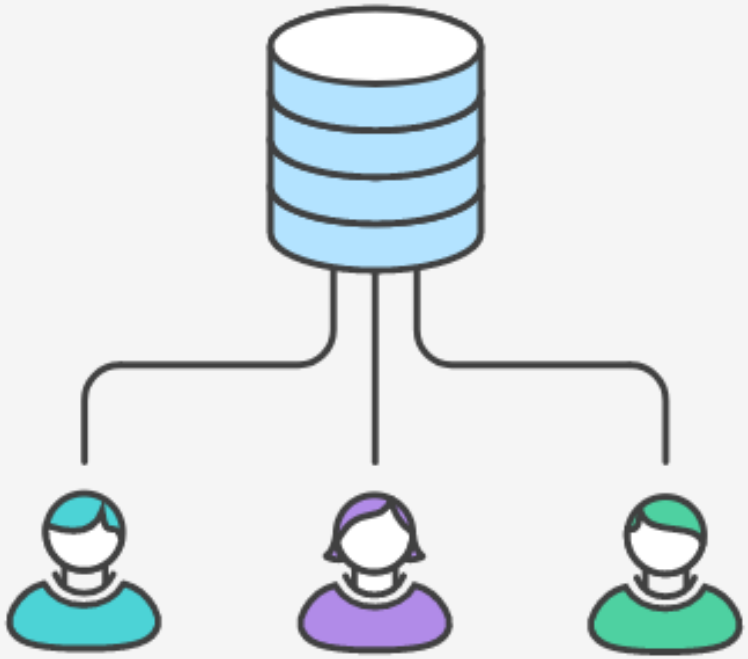
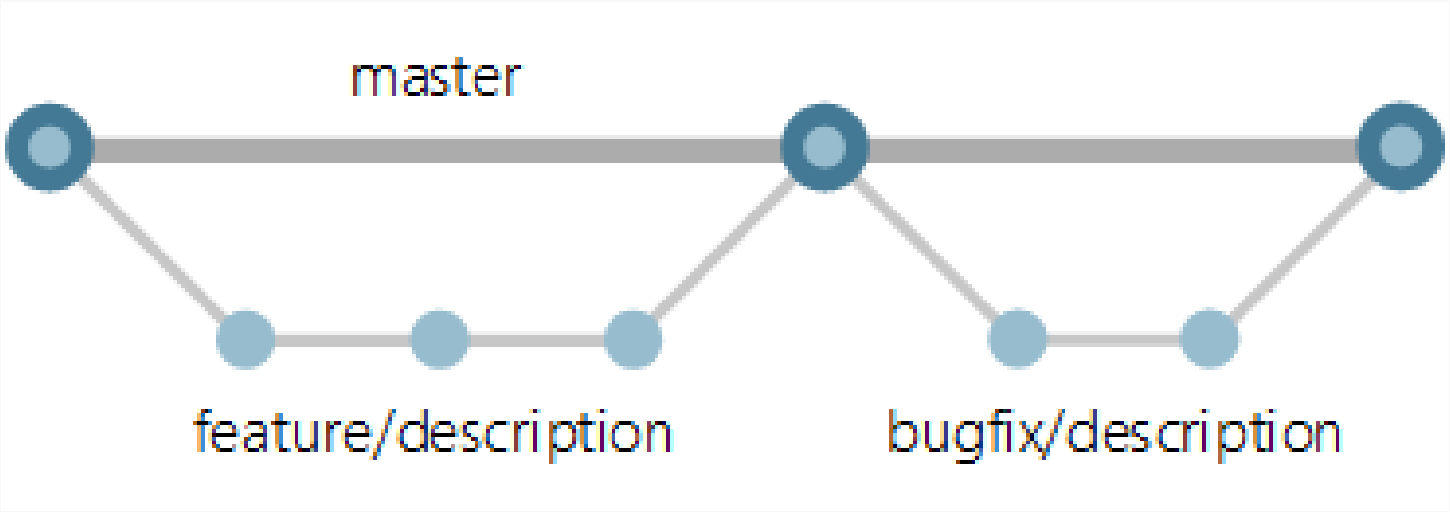


Image reference:
https://www.atlassian.com/git/tutorials/comparing-workflows

Feature Branch Workflow

- **Summary**: All dev should take place in a dedicated feature branch instead of the master branch. Leverages PR's. The official master branch should never have broken code. This workflow is a flexible guideline and can be incorporated into larger workflows with different code base and CI/CD environments.
- **Use Case**: Robust flexible framework for large projects with large teams.
- **Pros**: Flexible and adaptable workflow for additional workflows/environments/CI/CD processes. Goes really well with Agile, CI/CD processes, and suitable for large projects with large teams. Also okay if you have partial Agile (or non-Agile) and bleed bugs into the next sprint.
- **Cons**: Learning curve, overkill for maintenance of older projects with 1 dev,



Gitflow Workflow

- **Summary**: Strict branching model based around project release. Involves feature branches + PRs, and additionally individual branches for prepping, maintaining, and recording releases.
- Uses 2 branches to record project history. Master branch has official releases. Develop branch is the integration point for all the feature branches.
- When develop has enough features or a deadline is hit, fork a release branch off develop, fix bugs, and then merge into master + tag with version number. Merge back into develop also.

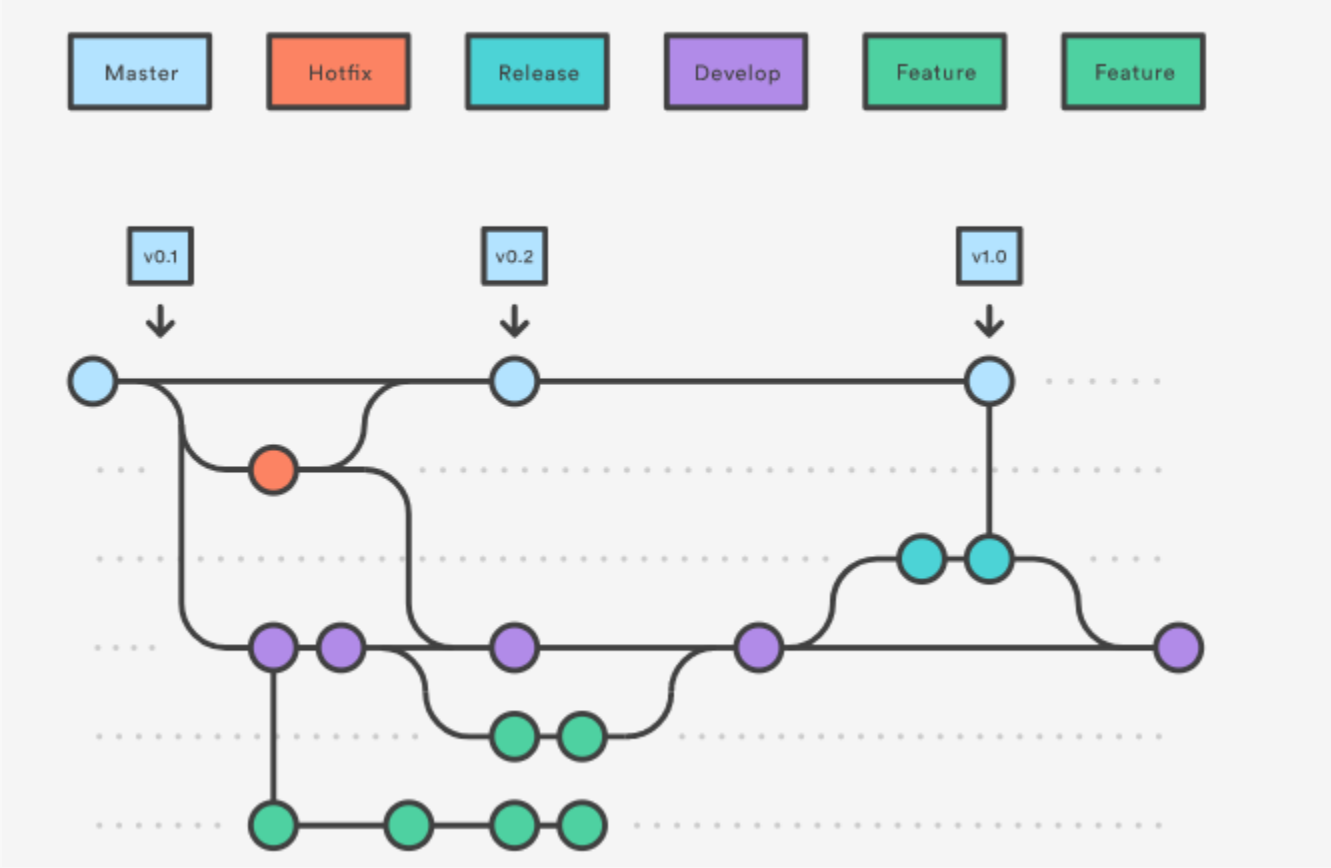


Image reference: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

Forking Workflow

- **Summary**: A contributor will “fork” (clone into their own remote server) a repo. Then, they will work on that remote copy and push to it from their own local copy. When the contributor is ready to integrate their changes, they do a PR and the owner of the original will have to accept for it to be merged in.
- **Use Case**: Public open source projects where it is open to anyone to make contributions. Very common on GitHub. Can also use with 3rd party integrations to enterprise apps.
- **Pros**: Prevents random folk from pushing in code into your open source project w/out permission.
- **Cons**: Merging / rebasing can become difficult once enough time has gone by and the ‘original’ branch has changed a lot. Unnecessary extra step/process/permissions for day to day dev work in house, easier to just publish feature branches for online backup.

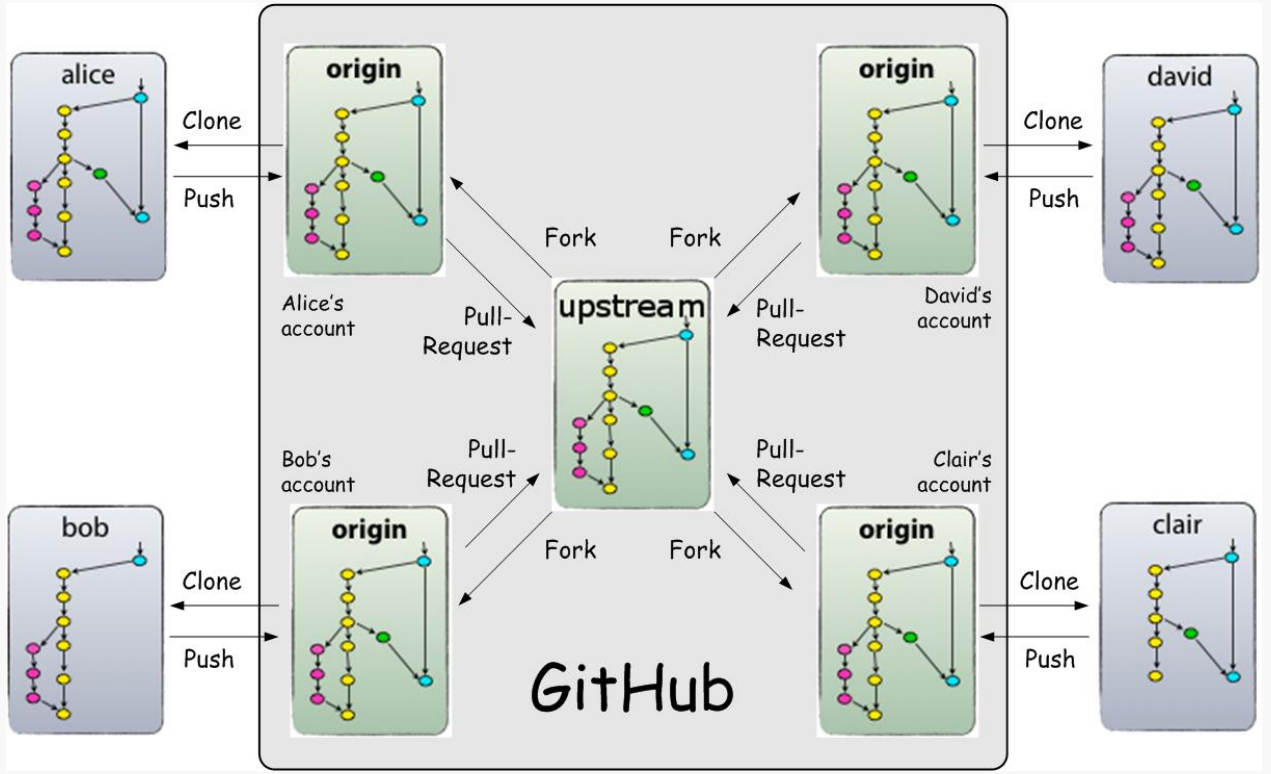


Image reference: http://www.dalescott.net/using-gitflow-with-githubs-fork-pull-model/
https://www.atlassian.com/git/tutorials/comparing-workflows

tl;dr Summary – Git

- **Code reviews** are built into the Git process
- Git is easy to make **different workflows** that fit your needs, from small one person projects to large enterprise projects with many teams sharing one codebase.
- Git is typically much faster and more performant than traditional centralized workflows.
- Every branch has the entire history, so no need to be online to get work done.

Questions?

Thank you!