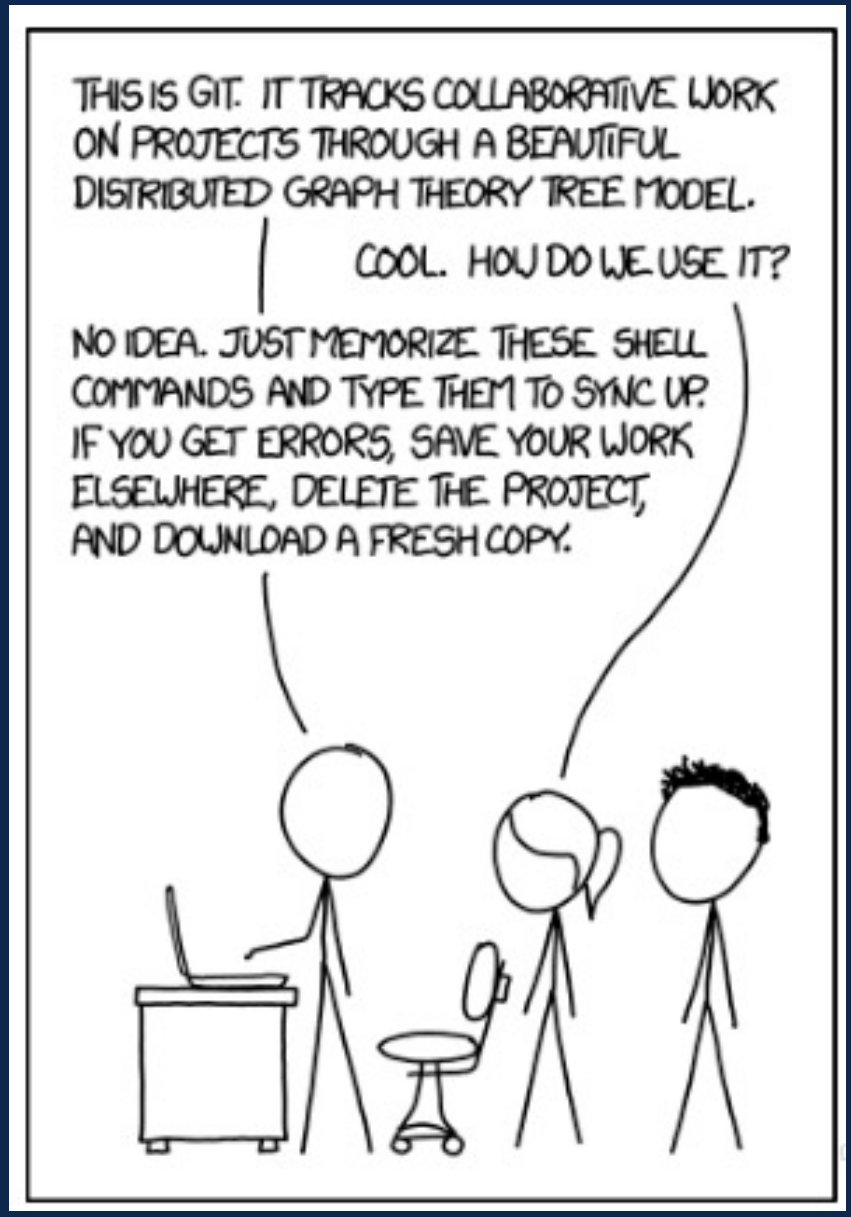
Git: Use Case & Introduction

20.4.2022

Ed Moore

Customer Success Specialist

Why are we here?



How to get started

Installation

Follow install instructions documented in the git-scm site for your chosen operating system: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Basic Config

After Installation/validation set your username and email:

```
git config --global user.name  my_username
git config --global user.email my@email.com
```

IDE Integration (Optional)

VS Code is the most popular tool with integrated SCM; additionally, there are many extensions to make life easier - a personal favorite is Git Graph.

https://code.visualstudio.com/docs/introvideos/basics

Before we begin.

- Introduction – 5 mins
- Overview to Git – 25 minutes
- Getting Started – 5 minutes
  - Hands on (Breakout)– 15 minutes
- Branching & Issue Resolution – 10 minutes
  - Hands on (Breakout)– 25 minutes
- Basic GitHub Walkthrough & Additional topics

Learning Labs

https://developer.cisco.com/learning/labs/git-intro/introduction/

If you're like me and struggle with non-practical learning, we've created a Learning Lab to nail down these basics to repeat at anytime or share with colleagues.

Reference Guide

https://git-scm.com/docs

Easily one of my most visited pages which comes in useful when you're in a mess or have forgotten a command.

Problem Statement

What do we want?

To efficiently create new functionality and manage existing functionality in a collaborative manner.

What makes this difficult?

As solutions have become more complex, collaboration with legacy tools is time consuming, error prone and slow.

What can we do about it?

- Outsource change tracking to a tool.
- Define policy-based operational rules.
- Enhance capabilities through integrations.

Why Git?

Free & Open Source

- GPLv2 guarantee
- Commercial platform offerings
- Industry leader

Performance

- Speed
- Scalability
- Reliability

Common Integrations

- Pipelines: Actions/Runners
- Comms: Teams/Email etc..
- Validation: Linters, pre-commit

Primary Personas

Individual developer

Starting to utilize some of the functionality of version-controlled systems, which may include common integrations and validations. Tools such as VS Code provide a low barrier to entry with a wide array of free solutions.

Infrastructure-Oriented

The state of the infrastructure is expressed with readable configuration and a standardized, automated pipeline.

Configuration needs to be versioned and meet specific requirements to ensure the security and performance of underlying applications.

Code-Oriented

The IT team are geographically spread across EMEA; they have a joint interest in managing a full-stack spanning multiple datacenters, branches and a campus estate.

Infrastructure as Code

- A representation of the desired end-state configuration
  - Structured Data like YANG and YAML
  - Human and machine readable
- Tracked in Version Control system
  - Change tracking and rollback
- Deployed via automation or orchestration tools
  - Ansible, Terraform, CloudFormation, NSO, etc.

```
---
- hosts: dnac_servers
  vars_files:
    - credentials.yml
  gather_facts: no
  tasks:

  - name: Find discovery
    cisco.dnac.discovery_range_info:
      dnac_host: "{{dnac_host}}"
      dnac_username: "{{dnac_username}}"
      dnac_password: "{{dnac_password}}"
      dnac_verify: "{{dnac_verify}}"
      dnac_debug: "{{dnac_debug}}"
      startIndex: 1
      recordsToReturn: 500
    register: discovery_range_result

  - name: Set fact of filtered discoveries
    set_fact:
      discoveries_filtered:
"{{discovery_range_result.dnac_response.response |
selectattr('name', 'equalto', 'start_discovery_test2')}}"
    when:
      - discovery_range_result is defined
      - discovery_range_result.dnac_response is defined
      - discovery_range_result.dnac_response.response is
defined
    register: discovery_filtered_results
```

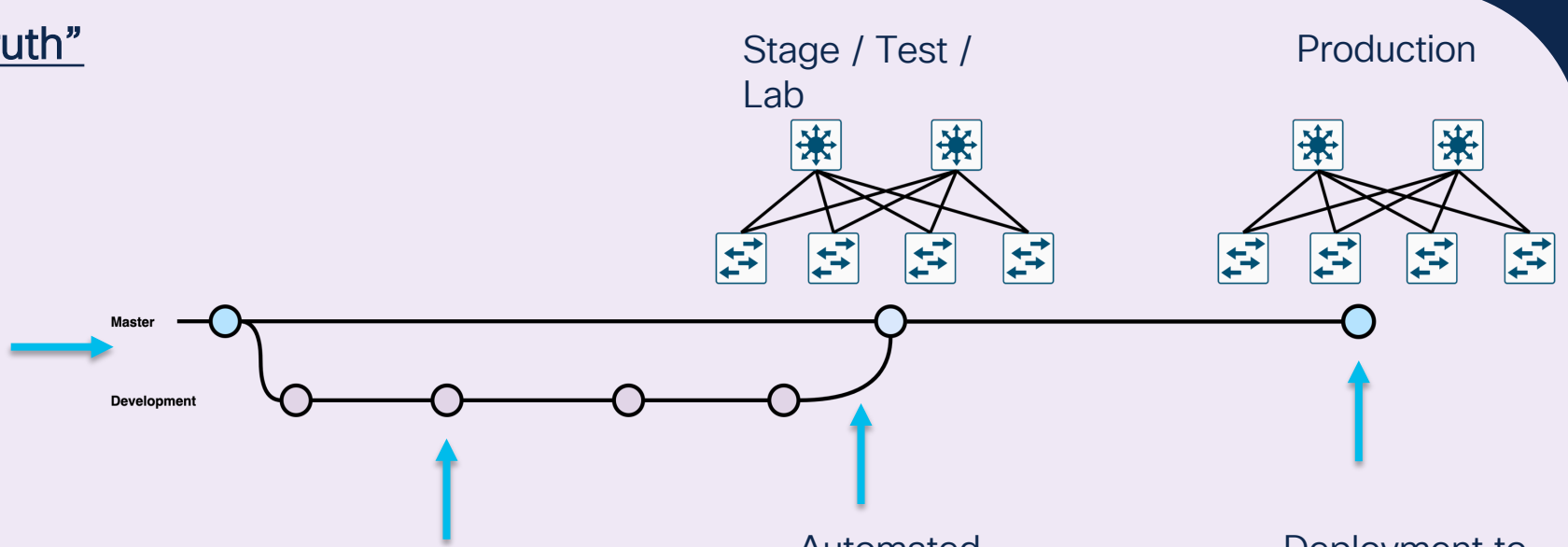
Key Concepts: Infra as Code & CI/CD

The “Source Of Truth”

```
---
- hosts: dnac_servers
  vars_files:
    - credentials.yml
  gather_facts: no
  tasks:

  - name: Find discovery
    cisco.dnac.discovery_range_info:
      dnac_host: "{{dnac_host}}"
      dnac_username: "{{dnac_username}}"
      dnac_password: "{{dnac_password}}"
      dnac_verify: "{{dnac_verify}}"
      dnac_debug: "{{dnac_debug}}"
      startIndex: 1
      recordsToReturn: 500
    register: discovery_range_result

  - name: Set fact of filtered discoveries
    set_fact:
      discoveries_filtered:
"{{discovery_range_result.dnac_response.response |
selectattr('name', 'equalto', 'start_discovery_test2')}}"
    when:
      - discovery_range_result is defined
      - discovery_range_result.dnac_response is defined
      - discovery_range_result.dnac_response.response is
defined
    register: discovery_filtered_results
```



Configuration modeled and stored as IaC

Version control & code branching

Automated deployment to Stage for validation

Deployment to Production

Continuous Integration (CI)

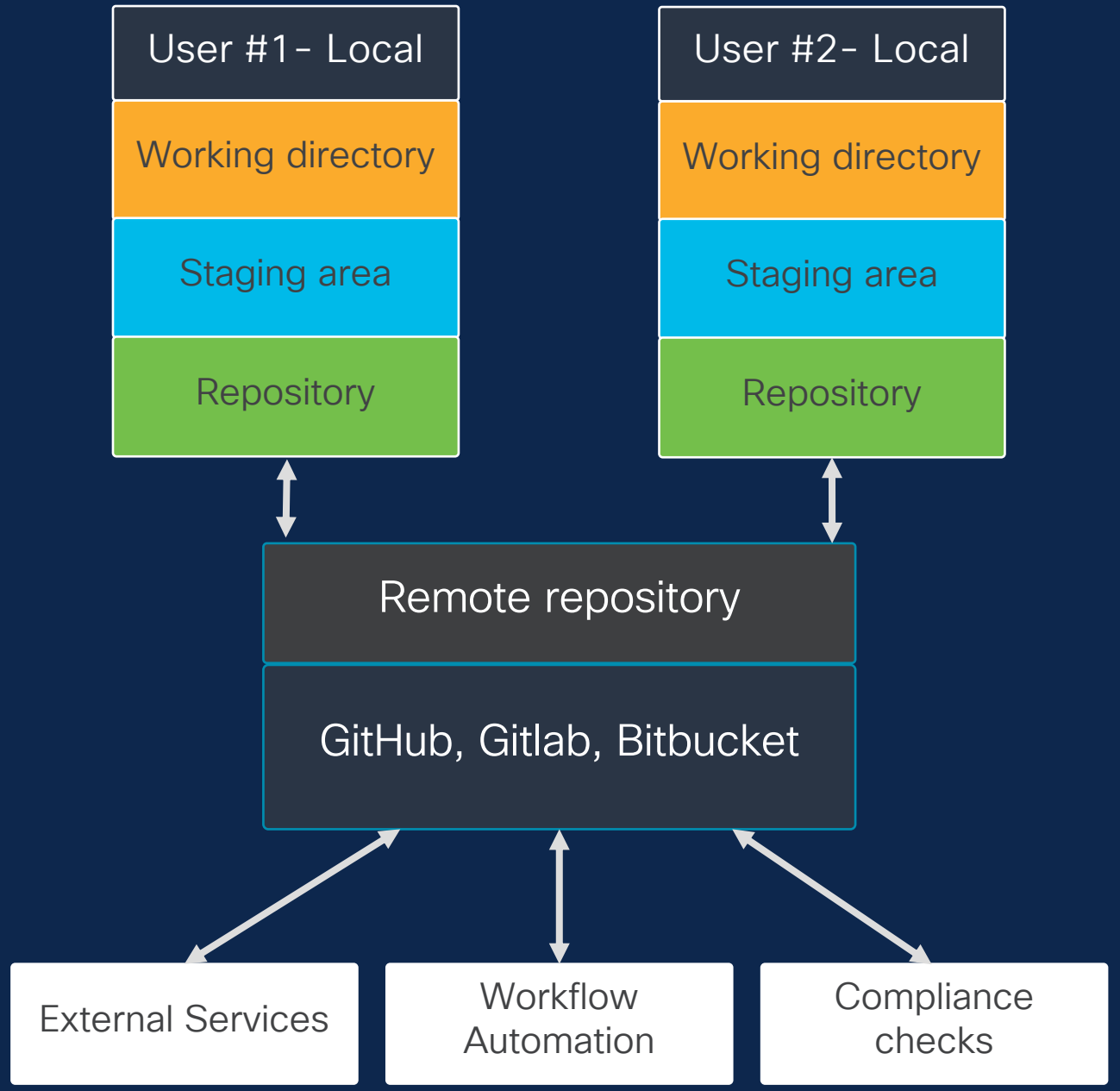
Continuous Deployment (CD)

Component Architecture

Typical functions

Two users are working on a project together,

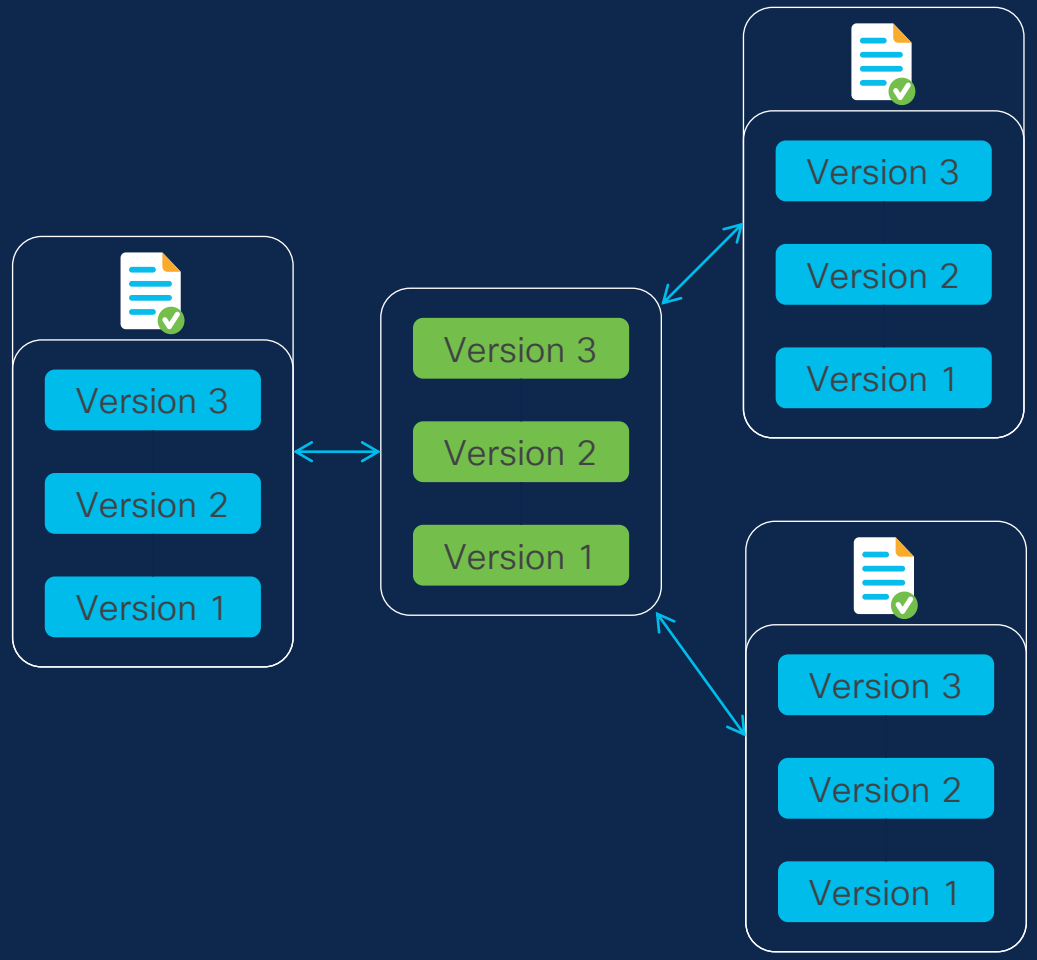
1. User #1 starts working on a project on their machine.
2. The pair decide to start working together and so User #1 They decide
3. User #2 begins to work with User #1 and so clones a copy of the remote repository to their machine.
4. Additional functionality/integrations are added as required.



**CRUD – Create, Read, Update, Delete*

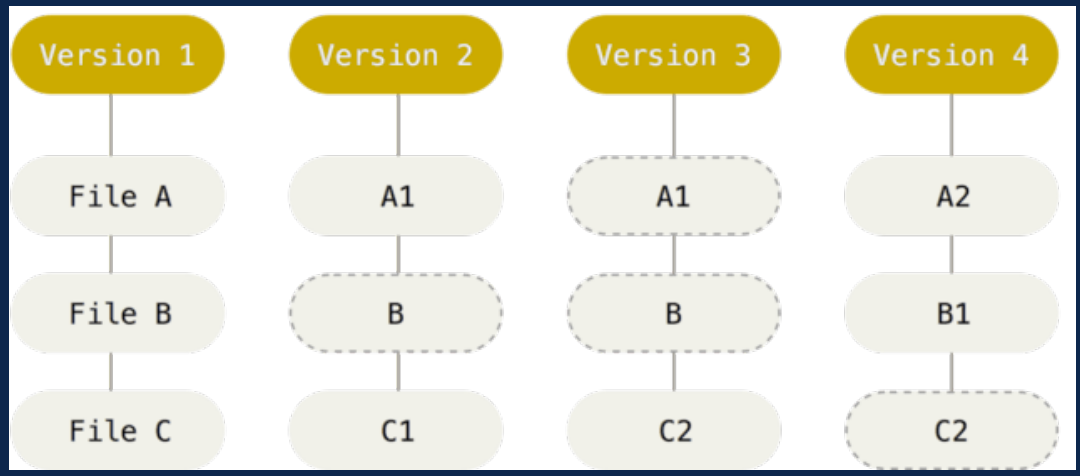
Distributed version control

- Peer-to-Peer approach to version control, as opposed to the client-server model of centralized systems
- Each user has a full working copy along with the full change history
- Working copies essentially function as remote backups, which helps avoid relying on one single source as a point of failure
- Most common are Git, Mercurial, and Bazaar

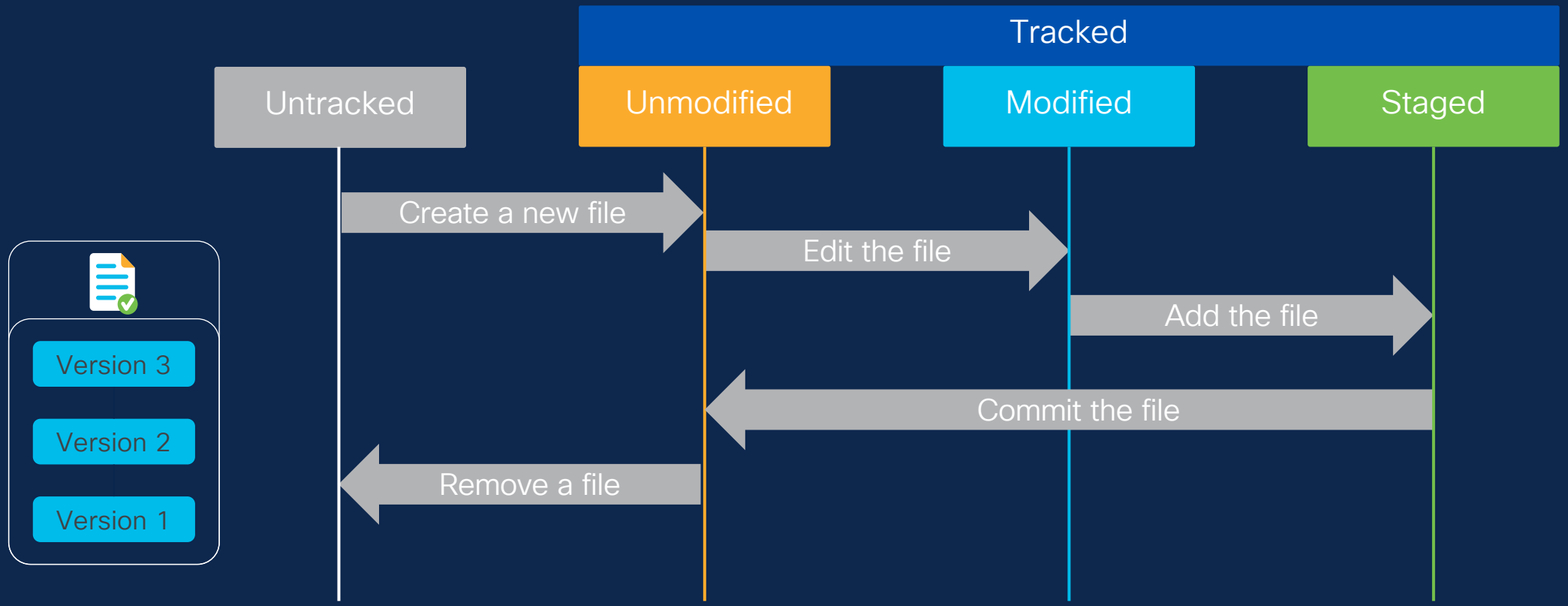


Introduction
Key Characteristics

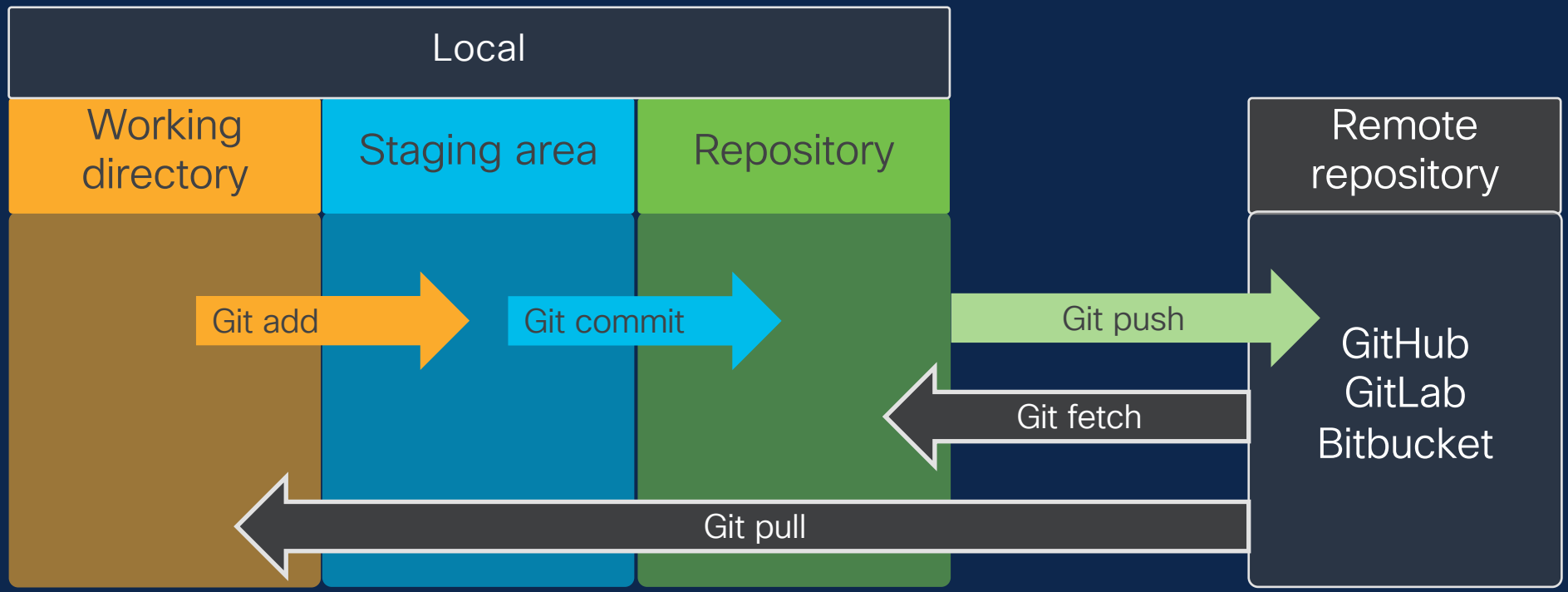
- Git is a **distributed VCS**
- Git stores **snapshots, not differences**
- Git performs nearly every operation **locally**
  - Most operations need only local files, so they can be performed offline
- Git has **Integrity**
  - Everything in Git is checksummed (SHA1) before it is stored and is then referred to by that checksum



Lifecycle: File



Moving files between the main areas of Git and remote repositories

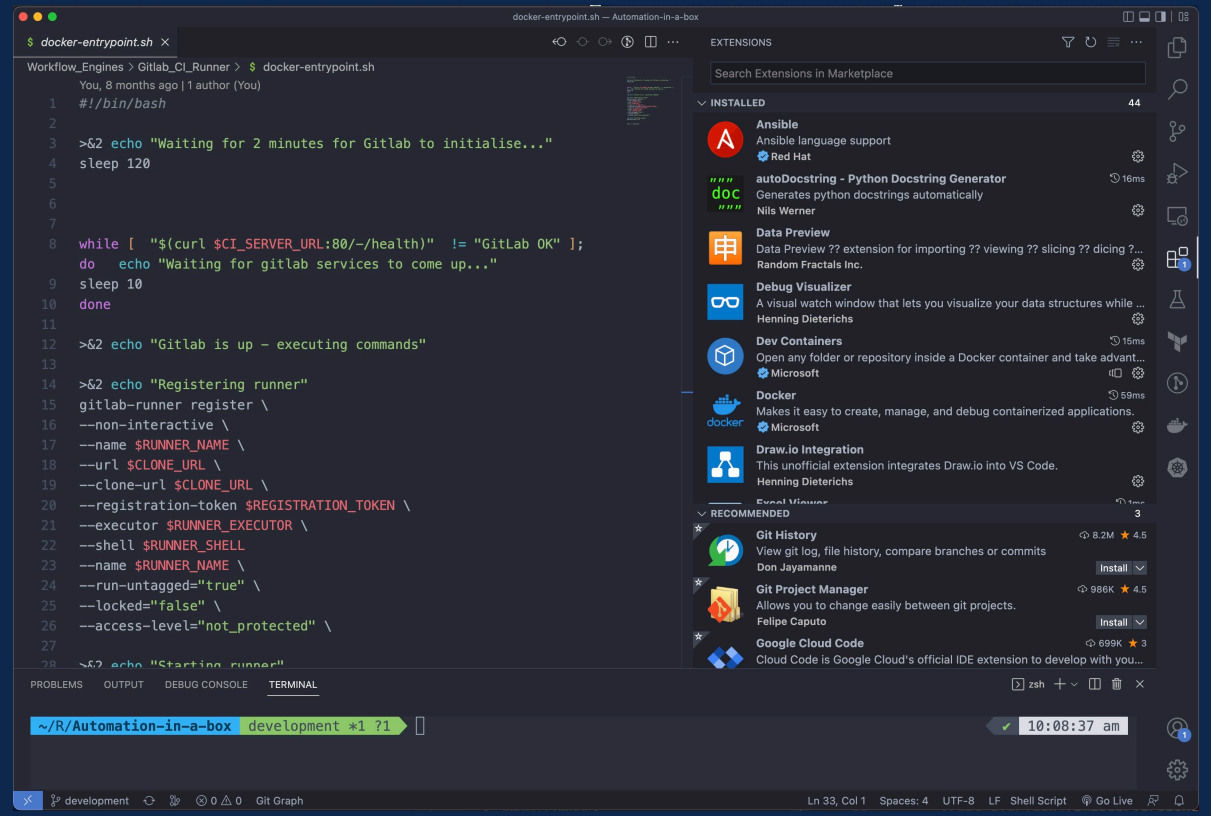


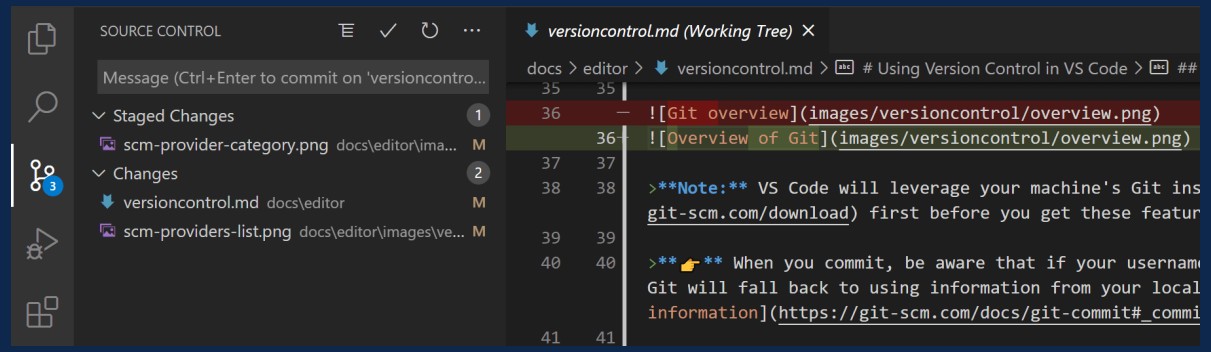
Visual Studio Code

Popular IDE with lots of Integrations

***Visual** Studio **Code** is a **code** editor redefined and optimized for building and debugging modern web and cloud applications. A lightweight version is available by replacing the dotcom GitHub address to dotdev:*

i.e.: https://github.dev/moore-automation/dnac_software_compliance_requests





Getting Started
Task Overview

- Initialise our repository
- Add files to the staging area
- Commit files to the local repository
- Verify changes and state is as expected.
- Experiment with different situations

My first project

Create a new file

```
mkdir example_demo
cd example_demo
touch example.txt
echo “sample text” >> example.txt
```

Check status

One of the most useful commands is:

```
git status
```

It details a summary of gits’ understanding of your repository.

Useful tools & IDEs

VS Code is the most popular tool with integrated SCM; additionally, there are many extensions to make life easier - a personal favourite is Git Graph.

Initialize our repository

init vs clone

Init is used to initialize a local repository and create the .git subdirectory. Git clone first initializes the local directory then copies the remote repository into that folder.

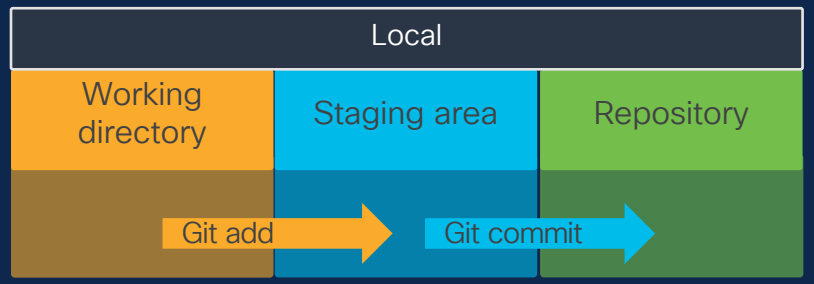
Clone existing repository

```
git clone https://github.com/repo_name...
```

Create a new Repository

```
cd /path/to/your/existing/code
git init
```

Add/Remove Files



Add file(s) to staging area

Files can be added to the index using the add command; a snapshot of the added content is then incorporated into future commits. Similarly, they can be removed using the rm command from either the index or working directory.

Remove file(s) from index

Add specific file to the index

```
git add <filename>
```

Add all files in directory (except ignored)

```
git add .
```

Remove file(s) from index

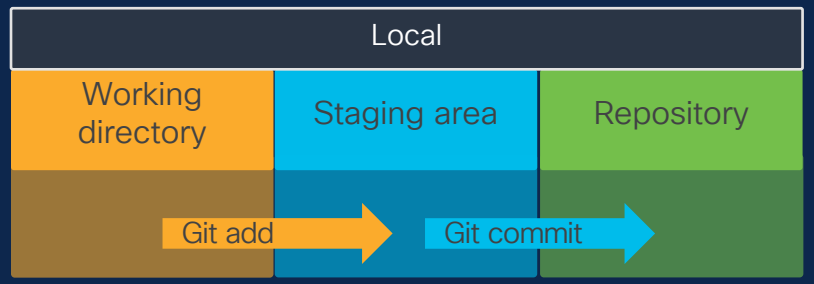
Remove specific file from index

```
git rm --cached <filename>
```

Remove specific file from directory and index.

```
git rm -f <filename>
```

Commit Changes



Commit

Snapshot of the entire repository at a given time, where all modifications to Staging Area are saved in the local repo. Files not added to Staging Area are not included.

```
git commit -a
```

Clone existing repository

Messages are required to indicate the purpose of each commit, there are many popular standards but general is to describe intent in a concise manner.

```
git commit -am ”this is commit message"
```

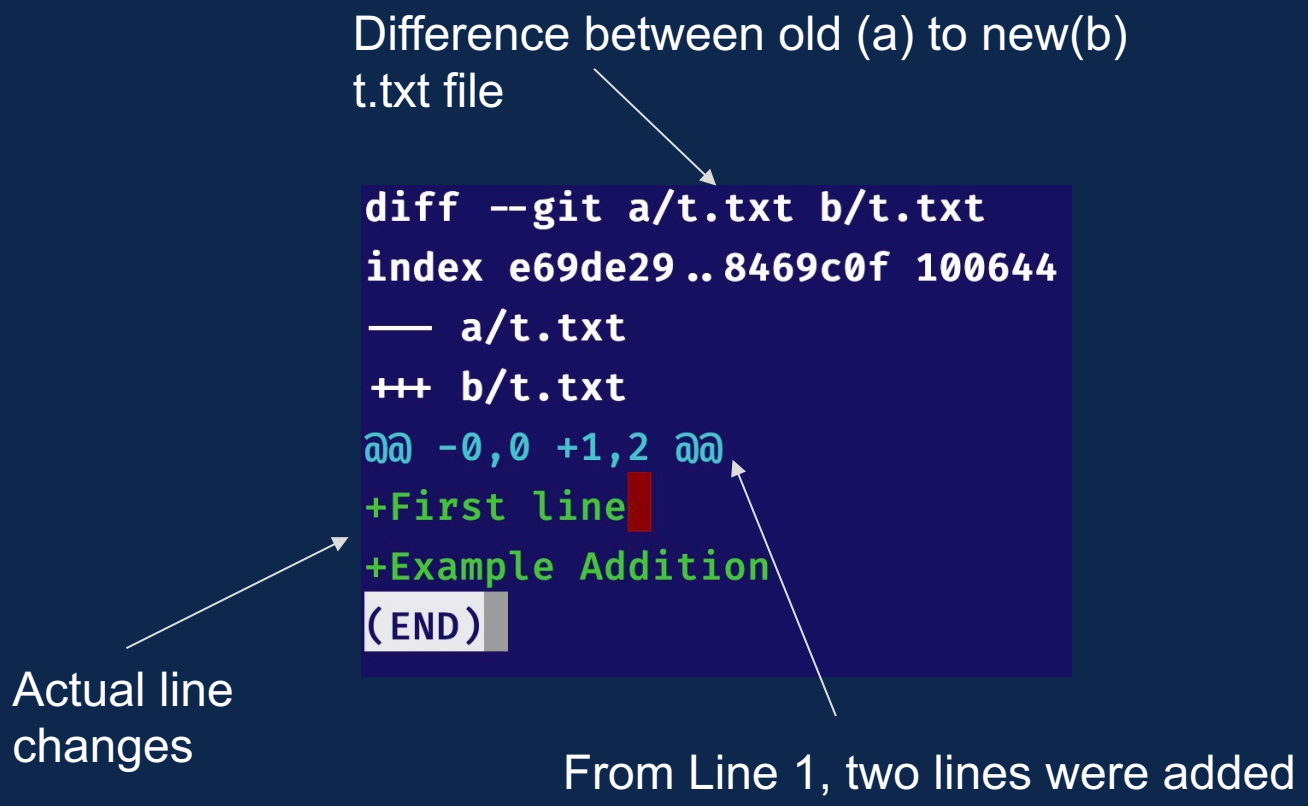
SHA Hashing

Each commit has unique SHA1 hash (40 chars), for example:

```
b4a450b50d9bf229dfb483be612506b813bb4fa2
```

hash(parent commit + date + author + message)

Differences



Verify changes

Delta of the change between Repository and Staging Area

```
git status -short
```

State: (M) Modified,(A) Added, (D) Deleted, (?) Untracked

diff

Diff provides a view of the differences found between tracked files prior to a commit.

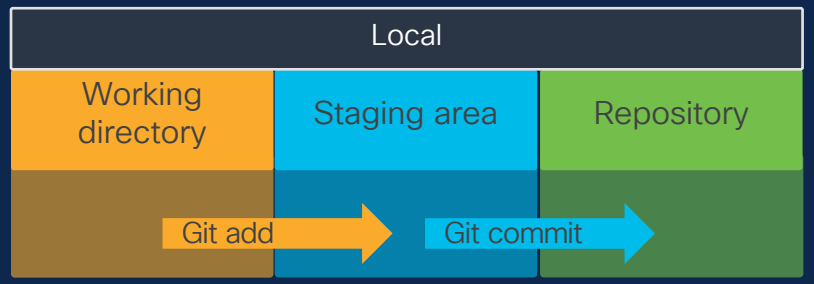
```
git diff
```

Difftool

Difftool is a useful appliance to compare the file differences and can be used with useful tools to make identifying differences simpler.

```
git config diff.tool vscode
```

What's going on?



Logs

Git log provides a useful overview of the history of commits made within the repository. There are many options that provide slightly different views, two are included below.

```
git log
```

Git Tags

A tag is a user-friendly pointer (label) to a given commit which is statically assigned to the commit.

```
git tag <tag_name>
```

```
git tag <tag_name> 98ae100
```

Alias

You can create git specific aliases or as standard within bash/zsh etc..

```
git config --global alias.history "log --oneline --graph --all"
```

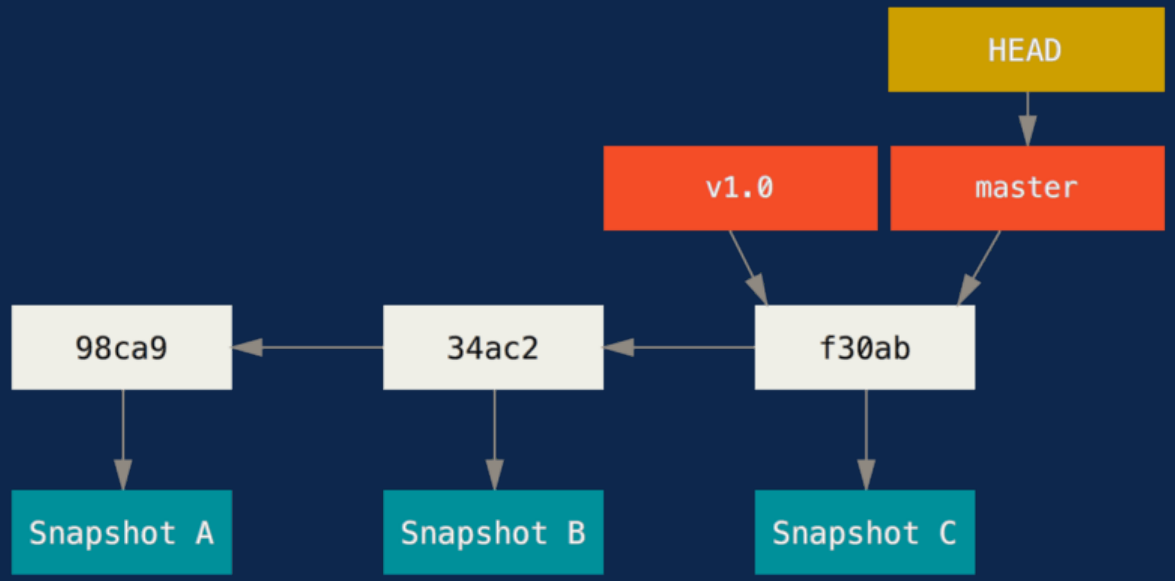
Branching & Issue Resolution

Task Overview

- Create a new branch
- Make updates to our new ‘feature’
- Compare differences between branches
- Verify changes and state is as expected.
- Experiment with different situations

Git Branches

Creating a branch



What is a Branch?

A branch in Git is simply a lightweight movable pointer to a commit. To list the created branches you can use the below command.

```
git branch
```

Swapping between branches is achieved by using the branch command with – specified branch name.

```
git branch <name>
```

Creating a Branch

Creating a branch can be done in two ways: *git branch* creates the branch but does not swap you into it, whereas *git checkout –b* combines the two processes.

```
git branch <name>
```

```
git checkout -b <name>
```

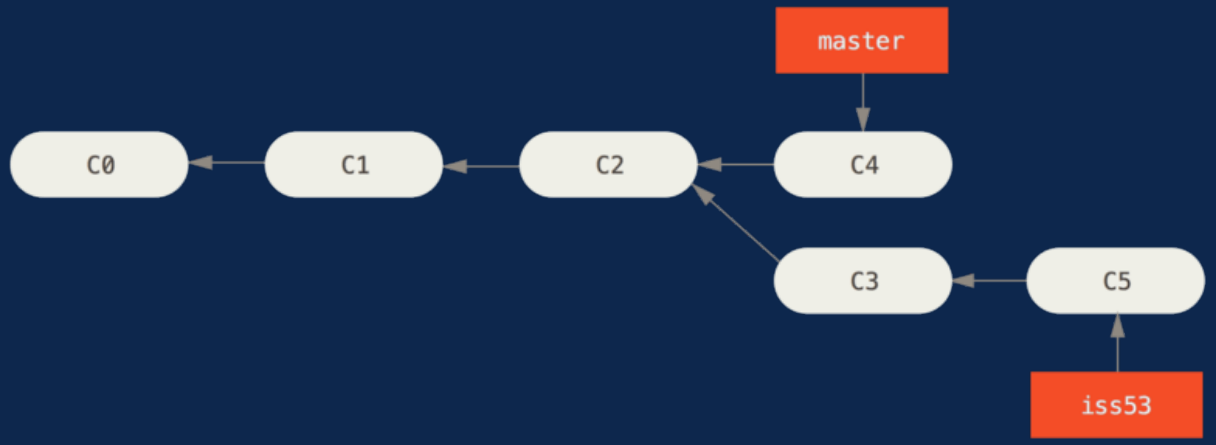
Delete a Branch

Branches can be deleted simply by using the –d option.

```
git branch -d <name>
```

Git Branches

Merging a branch



Merging

A branch in Git is simply a lightweight movable pointer to a commit.

```
git branch
```

Merging Process

In order to merge changes into a desired branch you need to checkout that branch and merge from the branch you've added too.

```
git branch <target>
```

```
git checkout -b <source>
```

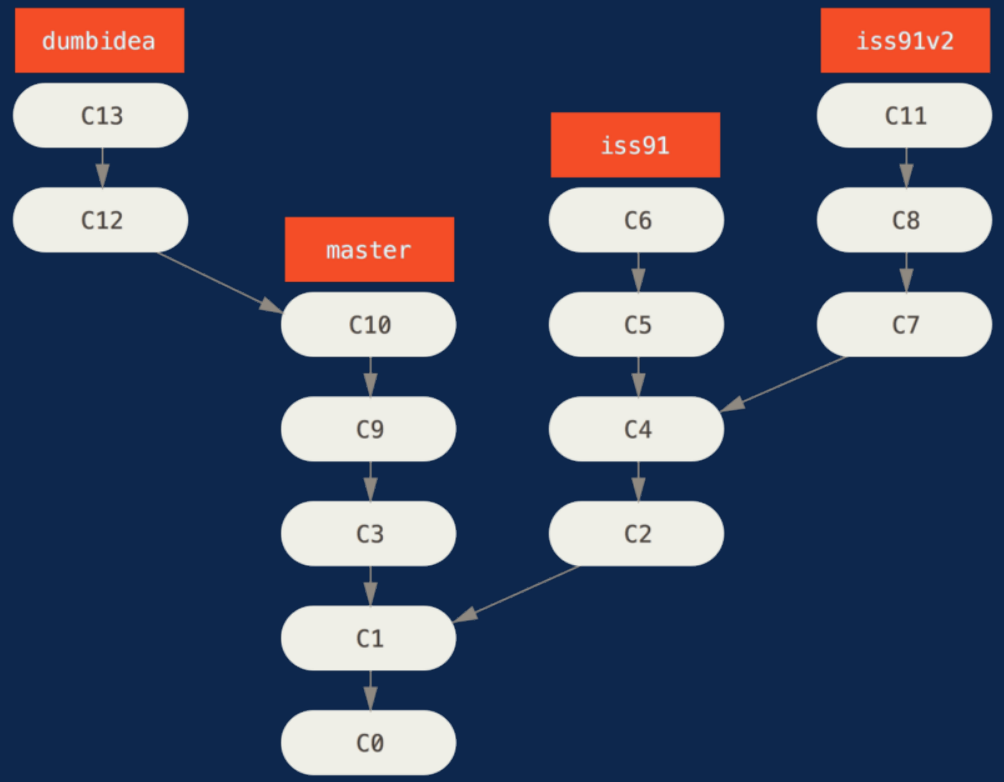
Merging Strategies

- Fast Forward vs Non Fast Forward
- Alternative : Rebase (discuss later)
- Pull request & Branch Policies

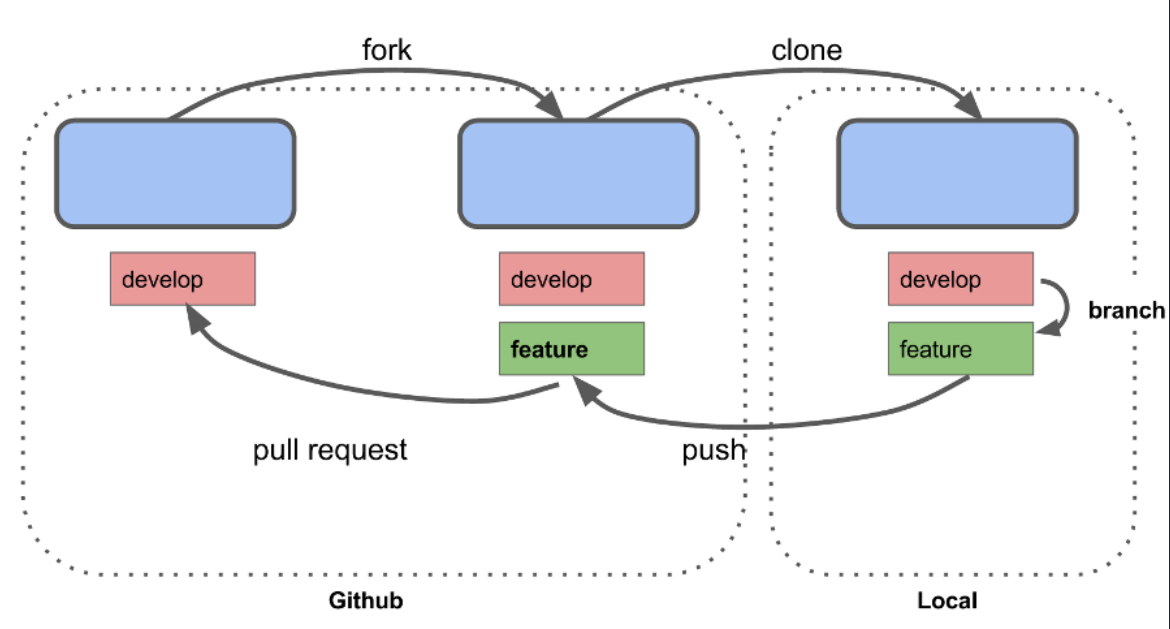
Git Branches

General Recommendations

- Use branches to implement new features or fix bugs.
- Try to keep the amount of changes in a branch small
  - e.g., single feature, bug fix, change request
- Give your branches a meaningful name
  - e.g., bug ID, change request ID
- The more you change and the longer you maintain a branch, the more your code will diverge from master.



Remote processes



Fetch

Fetch is used to request all remote branches with their associated commits and files to the /refs/remotes directory. It doesn't affect local branches and can therefore be used to compare remote vs local changes.

```
git fetch
```

Pull

Pull concatenates fetch and merge, where the remote branches and heads execute a local merge commit.

```
git pull <remote>
```

Push

As an inverse of pull, push uploads the commits from the local repository to the remote. Conflicts can be common when working in collaboration, so general practice is to fetch before a push and preferably to use seperate branches and pull requests.

```
git push origin -u
```

Issue Resolution

“I shouldn’t have added this”

```
› git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   private.key
        new file:   public.key

~/example_demo  main +2
```

Reset

Git Reset can be used to move both the HEAD pointer and the current branch pointer to a chosen point.

```
git reset <reference_pointer>
```

gitignore

Gitignore is a file located in the root directory which contains patterns with which git will ignore.
Common examples include build artefacts or private keys etc..

Resolution

```
git reset HEAD^
echo ’private.key’ >> .gitignore
git add .
git commit -m “Adding the public key”
```

“I should not have deleted that!”

Working/Stage Issues

Similar to the previous example reset can be used to rectify unmade changes by resetting to HEAD or an earlier time.

Checkout

We used checkout to swap branches in the previous section, it can be used for a different purpose to undo a change where the working directory is ‘swapped’ to the previous commit. This is useful for troubleshooting or rectification.

Resolution

```
git reset --hard HEAD
```

```
git checkout .
```

“Urgh, this is a mess”

Commit message update

Quite often you’ll encounter unclear or inaccurate commit messages; these can be amended in the following manner:

git commit –amend –m |”better message”

Git Clean

Clean essentially removes all untracked files from our Working directory.

```
git clean –f
```

In combination with the –nd option, it can indicate all untracked files & directories.

```
git clean –dn
```

Git Revert

Revert essentially acts as an undo operation though as historical integrity is important it calculates the inverse of a previous commit.

```
git revert HEAD
```

“What do you mean there’s a conflict?”

Merge conflict

When the same line is modified in two branche, this usually happens when more than 1 person works on the project. Git is unable to resolve which changes it should incorporate automatically.

Aborting a Merge

Merges can be aborted simply using the --abort option.

```
Git Merge --abort
```

```
(master) $  git merge feature
Auto-merging blah.file

CONFLICT (content): Merge conflict in
blah.file
Automatic merge failed; fix conflicts
and then commit the result.
```

What now?

Pro Git eBook (git-scm)

https://github.com/progit/progit2/releases/download/2.1.386/progit.pdf

VS Code – Source Control Tutorials

https://code.visualstudio.com/docs/sourcecontrol/overview

Automation for Engineers – White paper

https://www.cisco.com/c/dam/en/us/solutions/collateral/executive-perspectives/technology-perspectives/automation-driving-network-eng-skills-trans.pdf

The bridge to possible