

# Get the best out of Live Sessions HOW?



## Check your Internet Connection

**Log in 10 mins before**, and check your internet connection to avoid any network issues during the LIVE session.



## Speak with the Instructor

By default, you will be on mute to avoid any background noise. However, if required you will be **unmuted by instructor**.



## Clear Your Doubts

Feel free to clear your doubts. Use the “**Questions**” tab on your webinar tool to interact with the instructor at any point during the class.



## Let us know if you liked our content

Please share feedback after each class. It will help us to enhance your learning experience.



edureka!



# DevOps Certification Training

# COURSE OUTLINE

## MODULE 03

1. Introduction to DevOps

2. Version Control with Git

3. Git and Jenkins

4. Continuous Integration with Jenkins

5. Configuration Management using Ansible

6. Containerization using Docker Part - I

7. Containerization using Docker Part - II

8. Container Orchestration Using  
Kubernetes Part-I

9. Container Orchestration Using  
Kubernetes Part-II

10. Monitoring Using Prometheus and  
Grafana

11. Provisioning Infrastructure using  
Terraform Part - I

12. Provisioning Infrastructure using  
Terraform Part - II



**edureka!**



# Module 3 – Git And Jenkins



**edureka!**

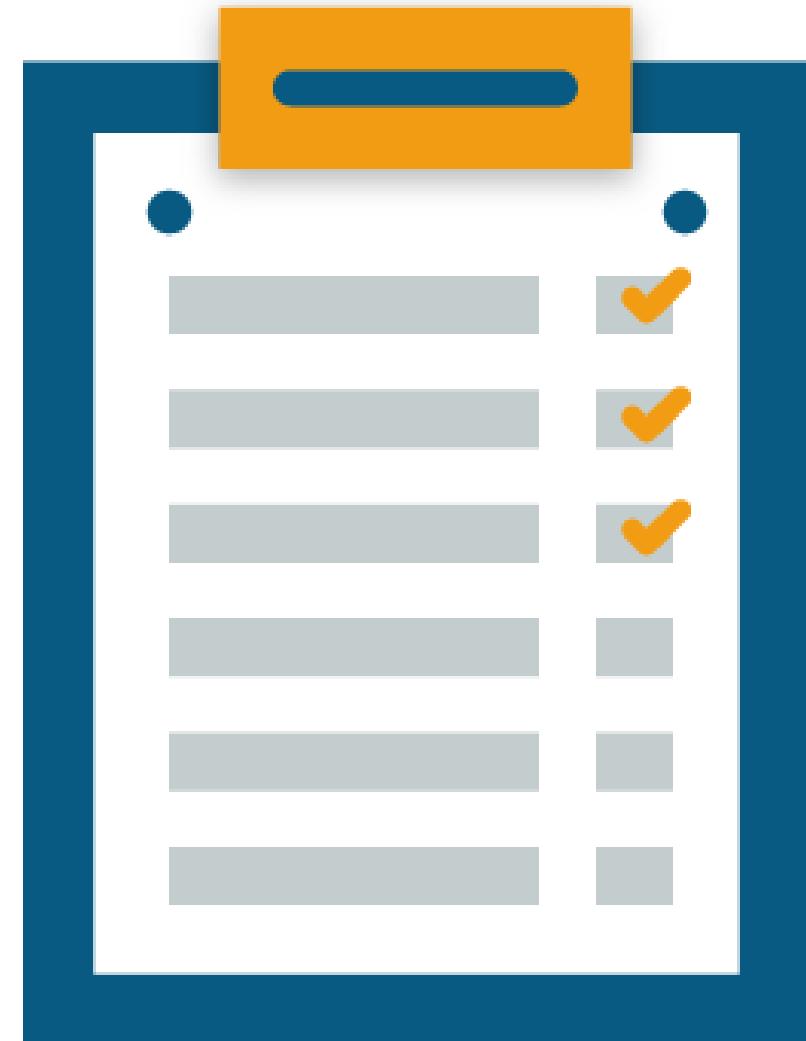
Copyright © edureka and/or its affiliates. All rights reserved.

# Topics

---

Following are the topics covered in this module:

- Branching and merging in Git
- Merge Conflicts
- Stashing, Rebasing, Reverting and Resetting
- Git Workflows
- Introduction to Maven
- Maven Architecture
- Introduction to Continuous Integration
- Introduction to Jenkins



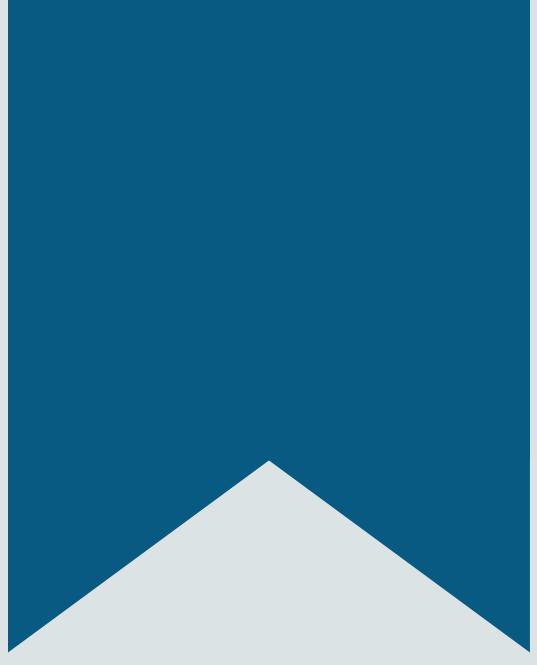
# Objectives

---

After completing this module, you should be able to:

- Execute branching and merging operations
- Perform various Git commands
- Introduction to Maven
- Understand Maven Architecture and dependencies
- Learn about Continuous Integration & its importance
- Understand Jenkins and its features

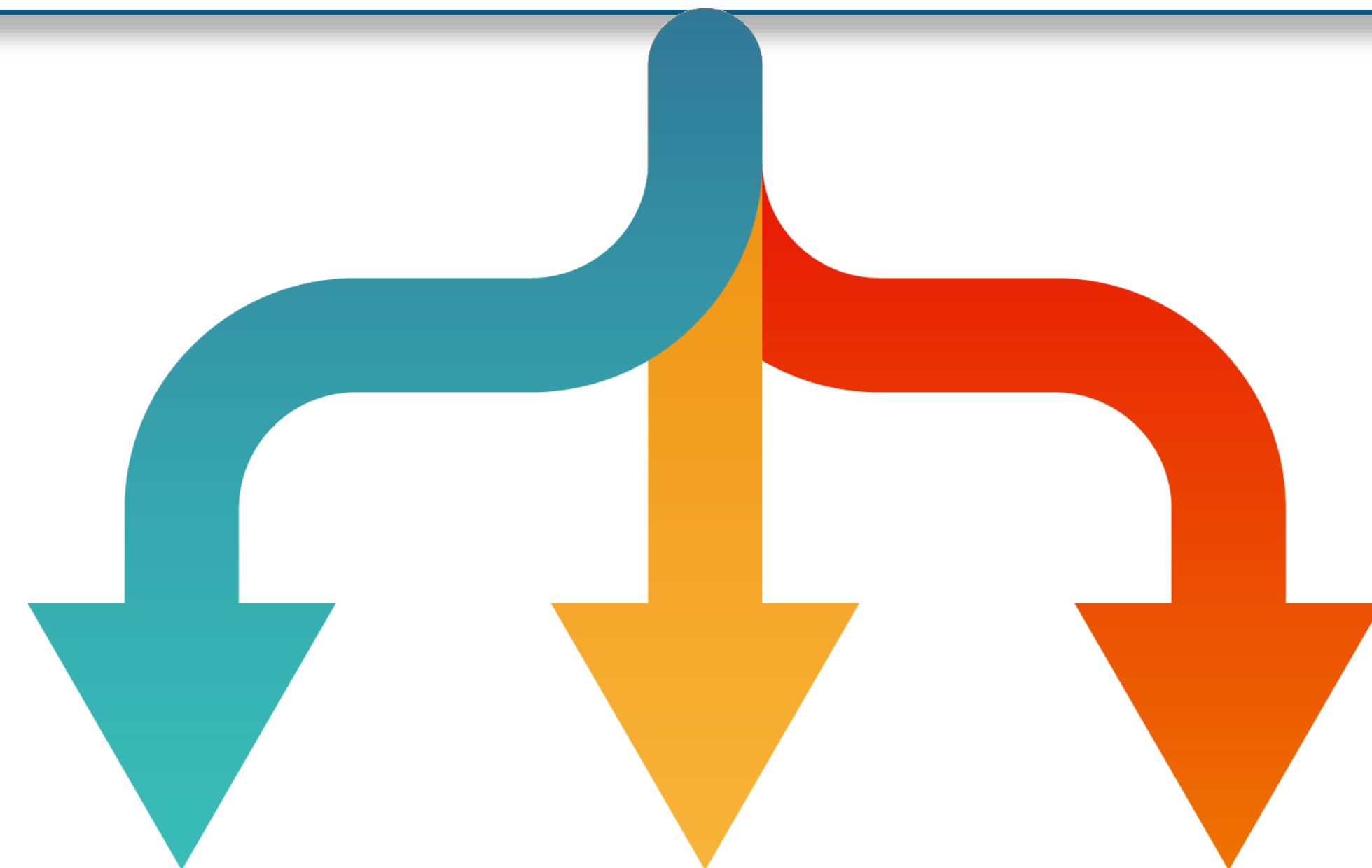




# Branching And Merging

# Branching

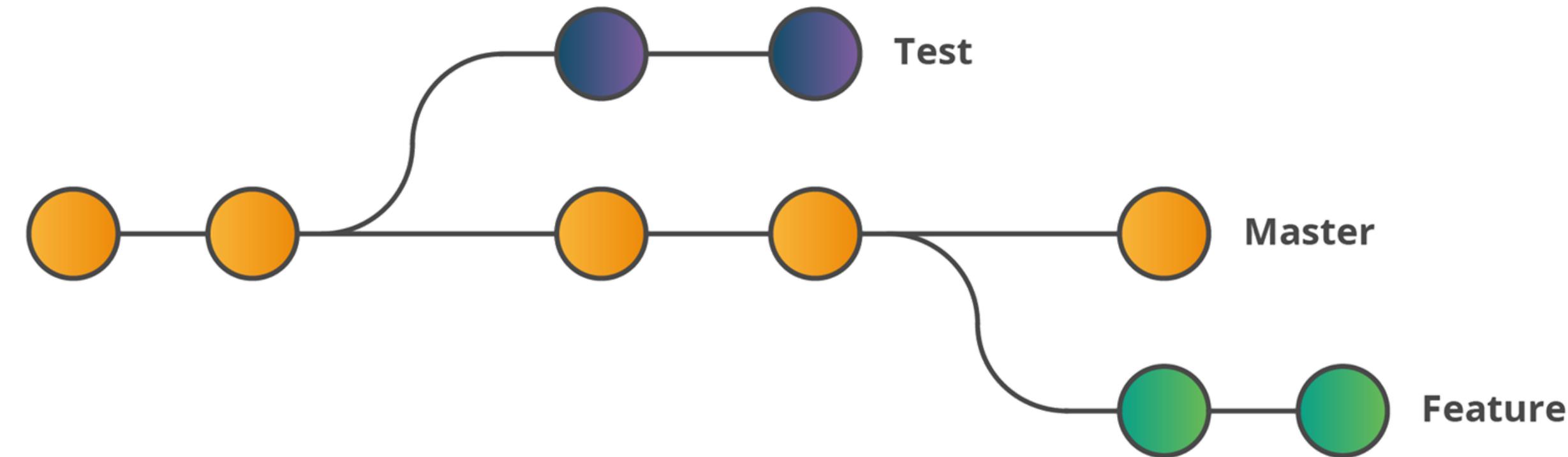
A project in its development could take multiple different paths to achieve its goal. Branching helps us take these different directions, test them out and in the end achieve the required goal.



# Branching In Git

---

- Branching is an integral part of any Version Control(VC) System
- Unlike other version control systems, Git **does not** create a copy of existing files for a new branch
- It **points** to a snapshot of the changes you have made in the system



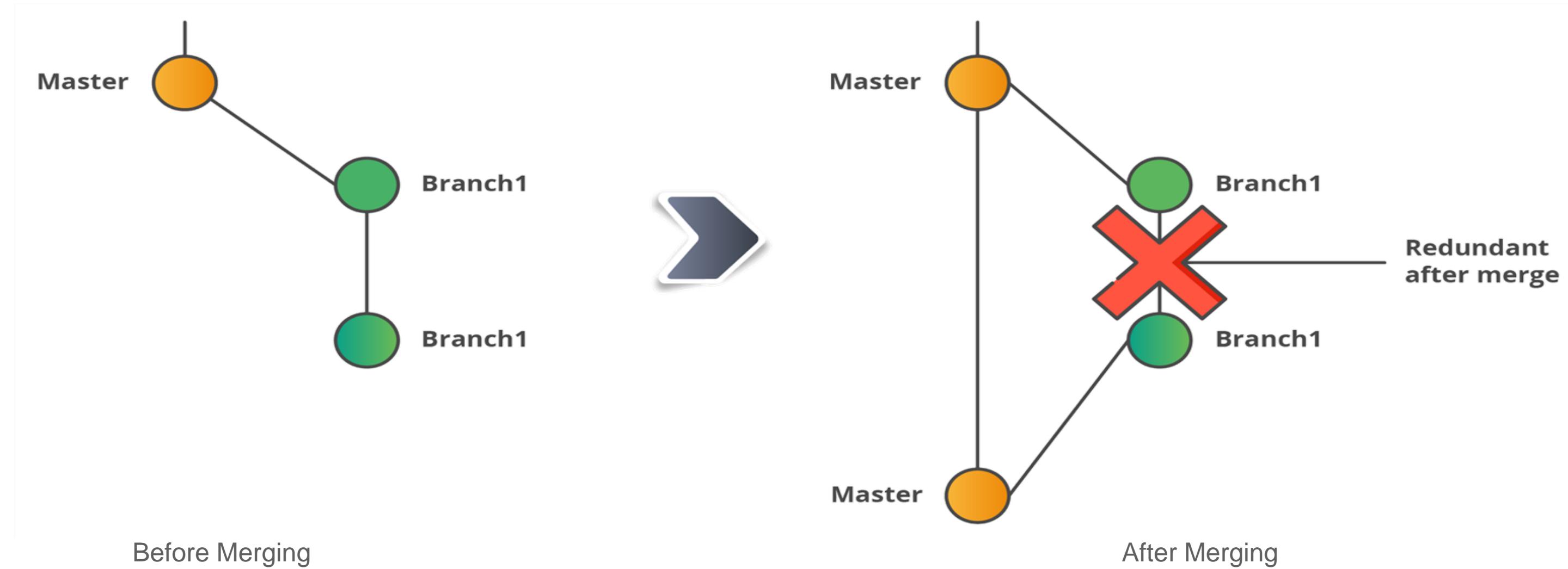
# Merging In Git

---

Merging integrates the changes made in different branches into one single branch



# Merging

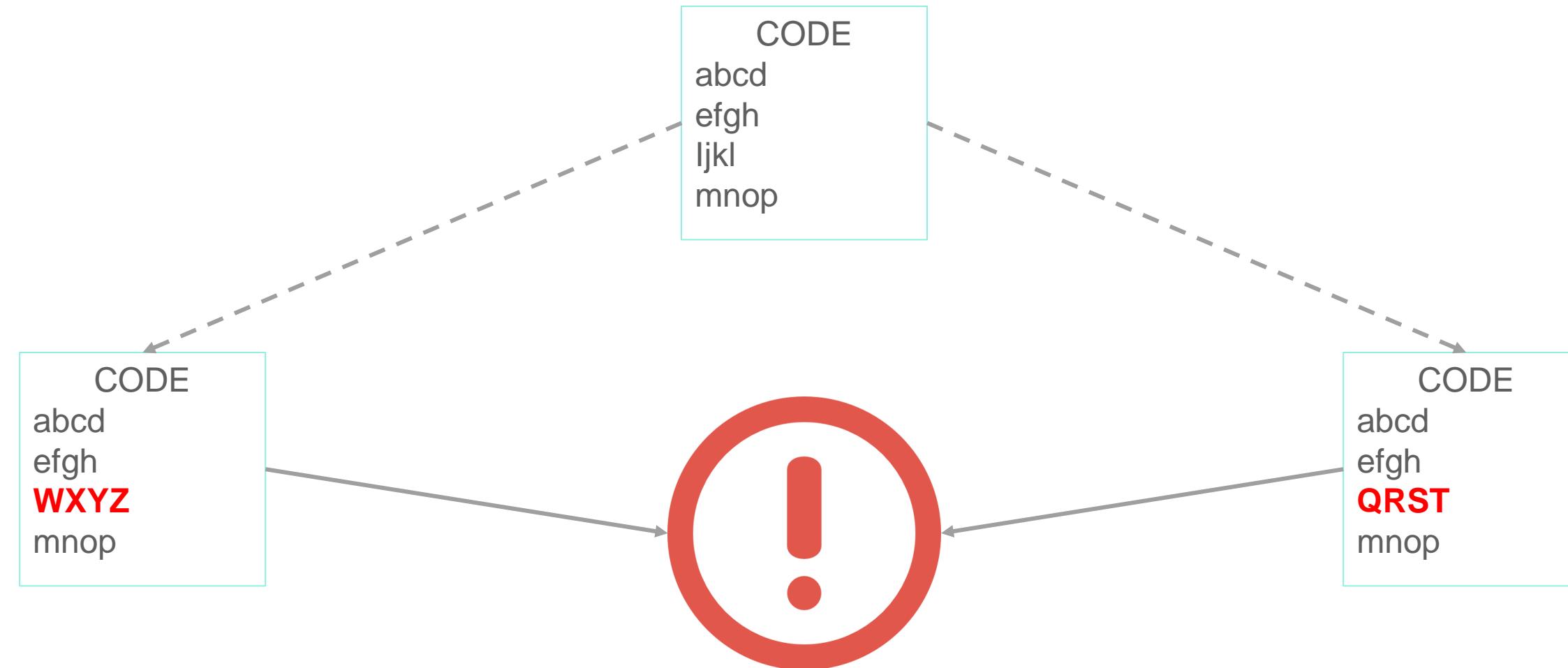


- All the changes made in **Branch1** after merging are available in the Merged branch(Master)
- Branch1 becomes redundant after merging, hence it can be deleted

# Merge Conflict

---

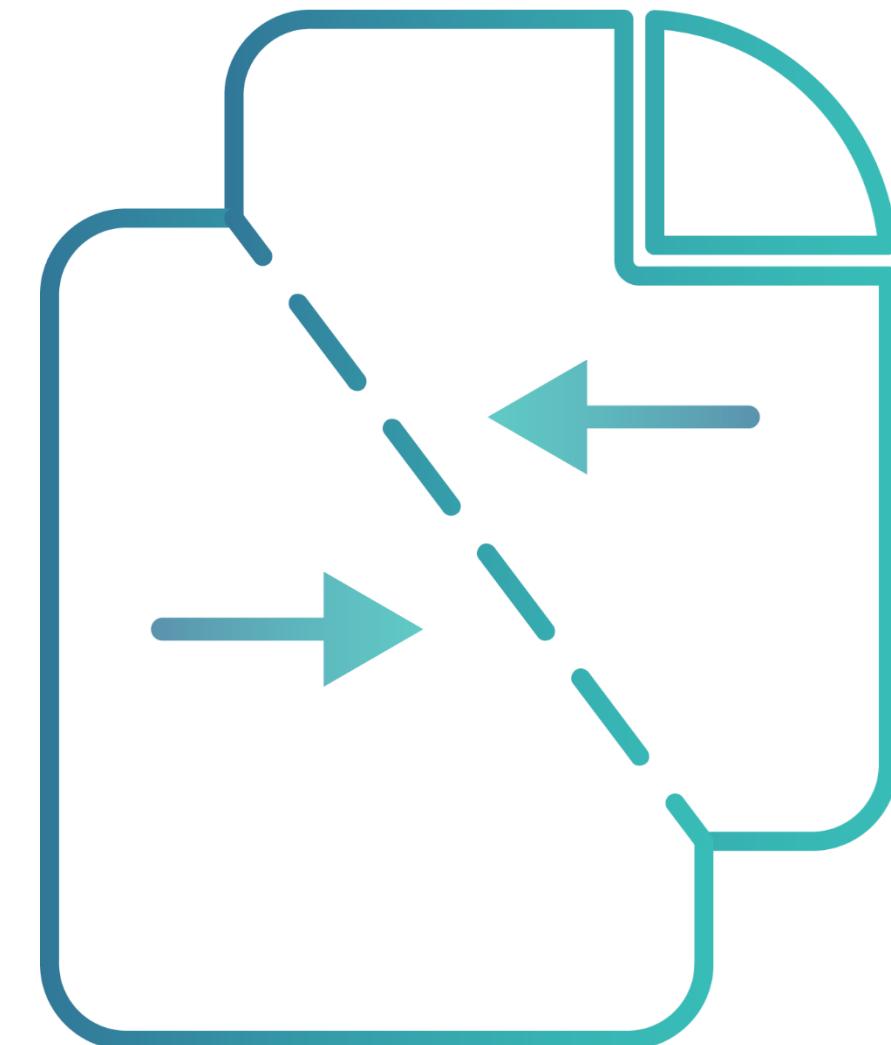
- Merge conflicts arise when two files having same content modified are merged
- Merge conflicts can occur on merging branches or when merging forked history together



# Resolving Merge Conflict

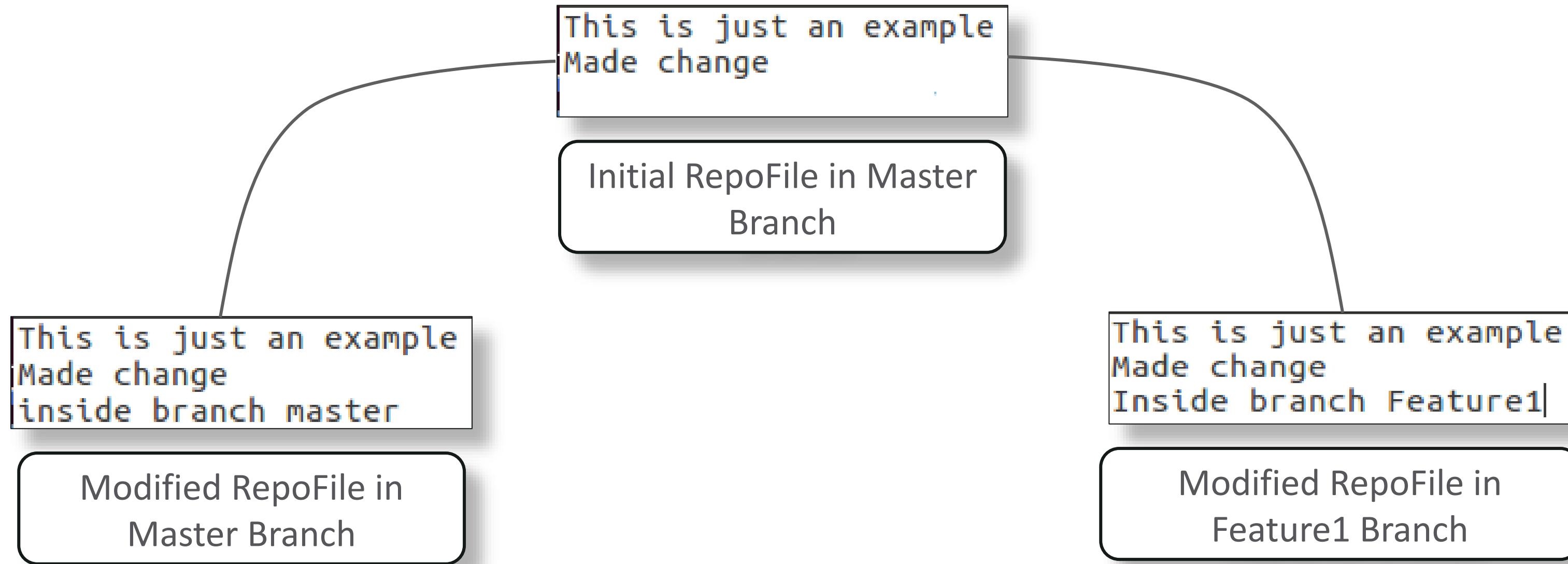
---

- Merge Conflicts are resolved manually by users
- Git provides different Merge-Tools to compare and choose the required changes
- User can also use third party Merge-Tools with Git



# Merge Conflict

---

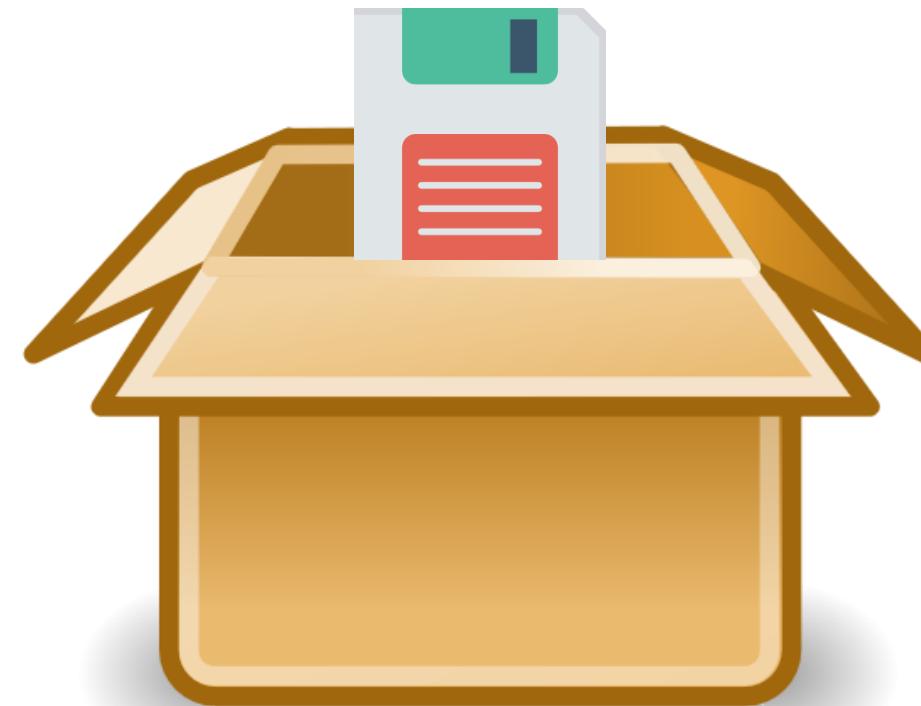




# Stashing, Rebasing, Reverting and Resetting

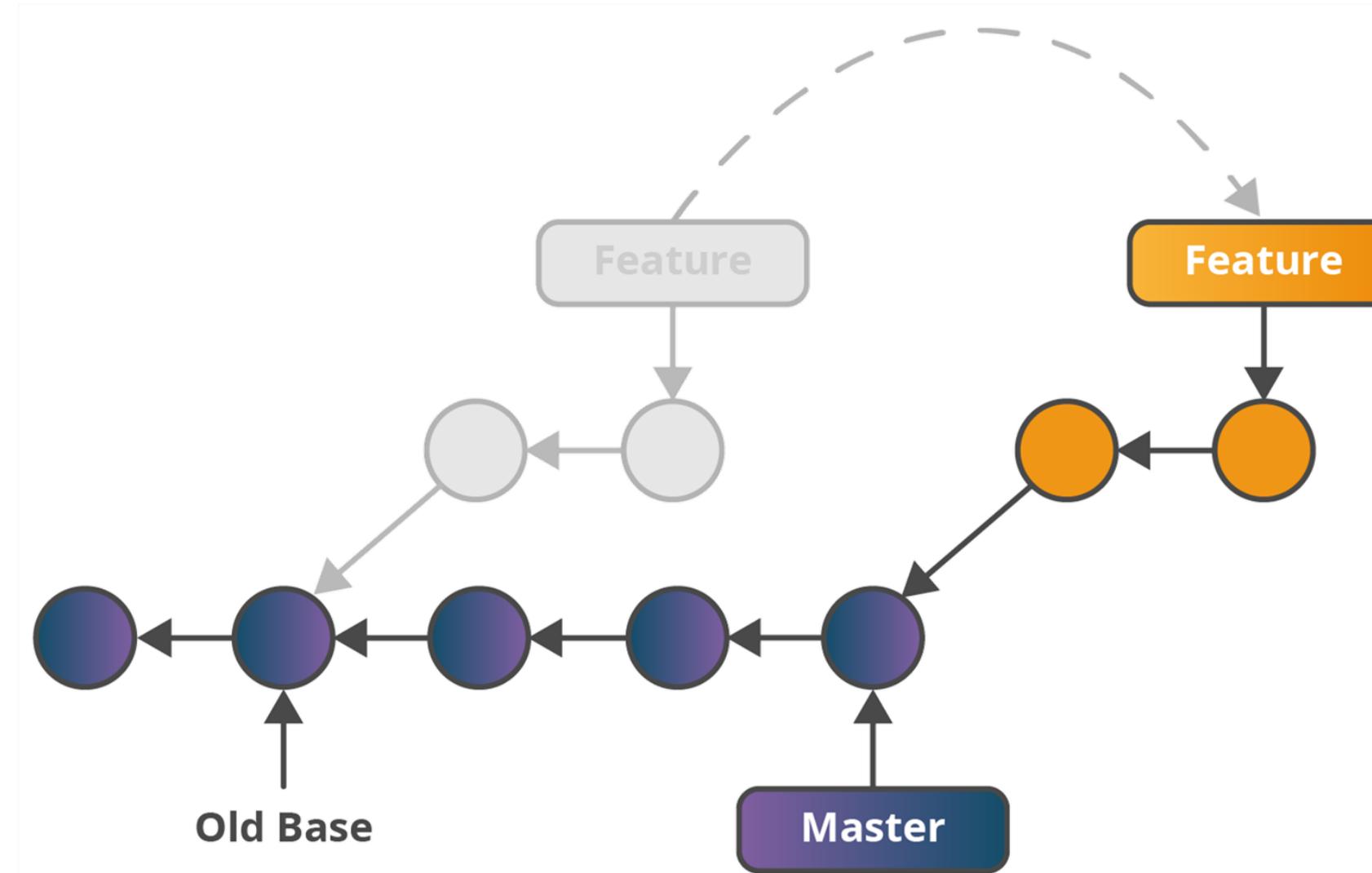
# Git Stashing

Git Stashing is a way of creating a checkpoint for non-committed changes. It saves all the changes to a temporary location so the user can perform other tasks such as switching branches, reverting etc. These changes can then be reapplied anywhere.



# Branch Rebasing

Git rebasing is used, when changes made in one branch needs to be reflected in another branch



# Rebasing

---

To Rebase a branch:

Syntax: git rebase <rebase branch>

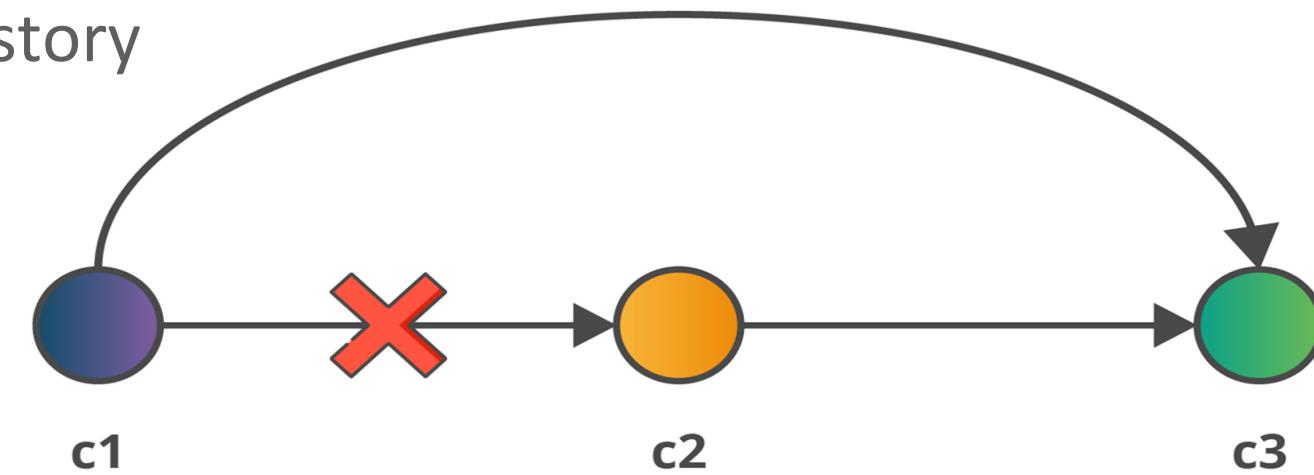
```
amr@amr-VirtualBox:~/Documents/demo$ git rebase master
First, rewinding head to replay your work on top of it...
Fast-forwarded feat2 to master.
amr@amr-VirtualBox:~/Documents/demo$
```

# Revert

- Revert or **undo** the changes made in the previous commit
- New commit is created without the changes made in the other commit

Syntax: git revert <commit id>

- Old commit still resides in the history



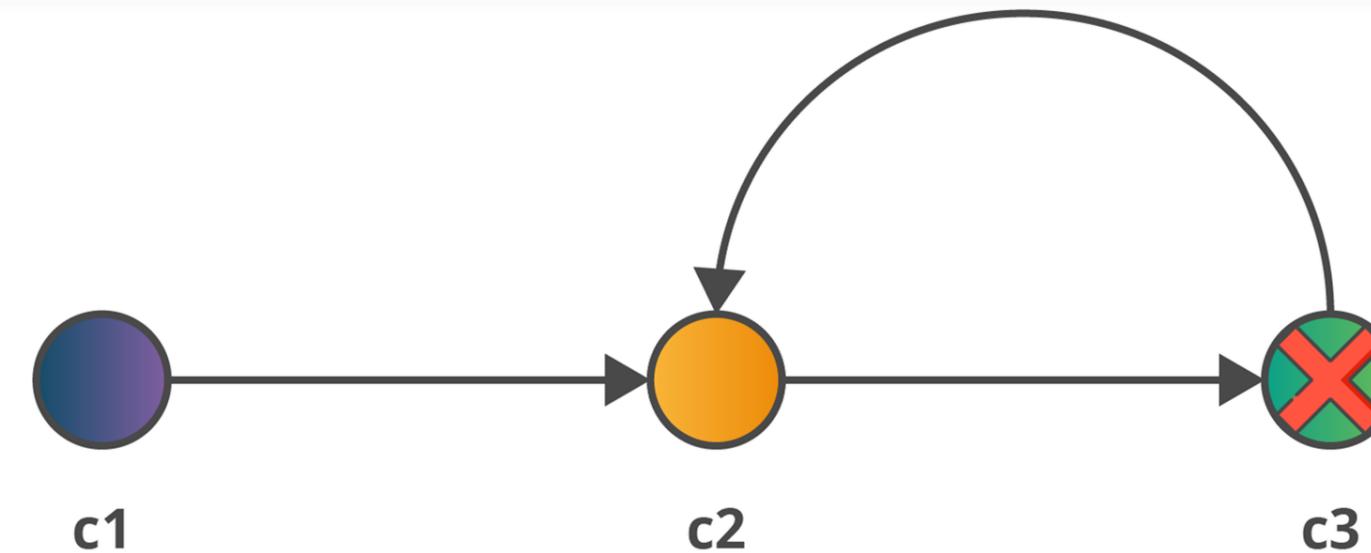
HEAD refers to the latest commit

```
amr@amr-VirtualBox:~/Documents/demo$ git revert HEAD  
[feat2 a50d898] Revert "revert edit"  
1 file changed, 1 deletion(-)  
amr@amr-VirtualBox:~/Documents/demo$
```

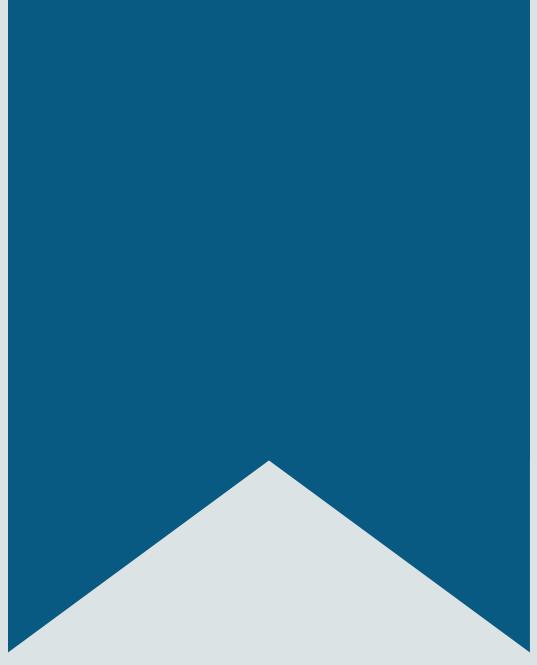
# Reset

- Reset command can be used to undo changes at different levels
- Modifiers like --hard, --soft and --mixed can be used to decide the degree to which to reset

Syntax: git reset <modifier> <commit id>



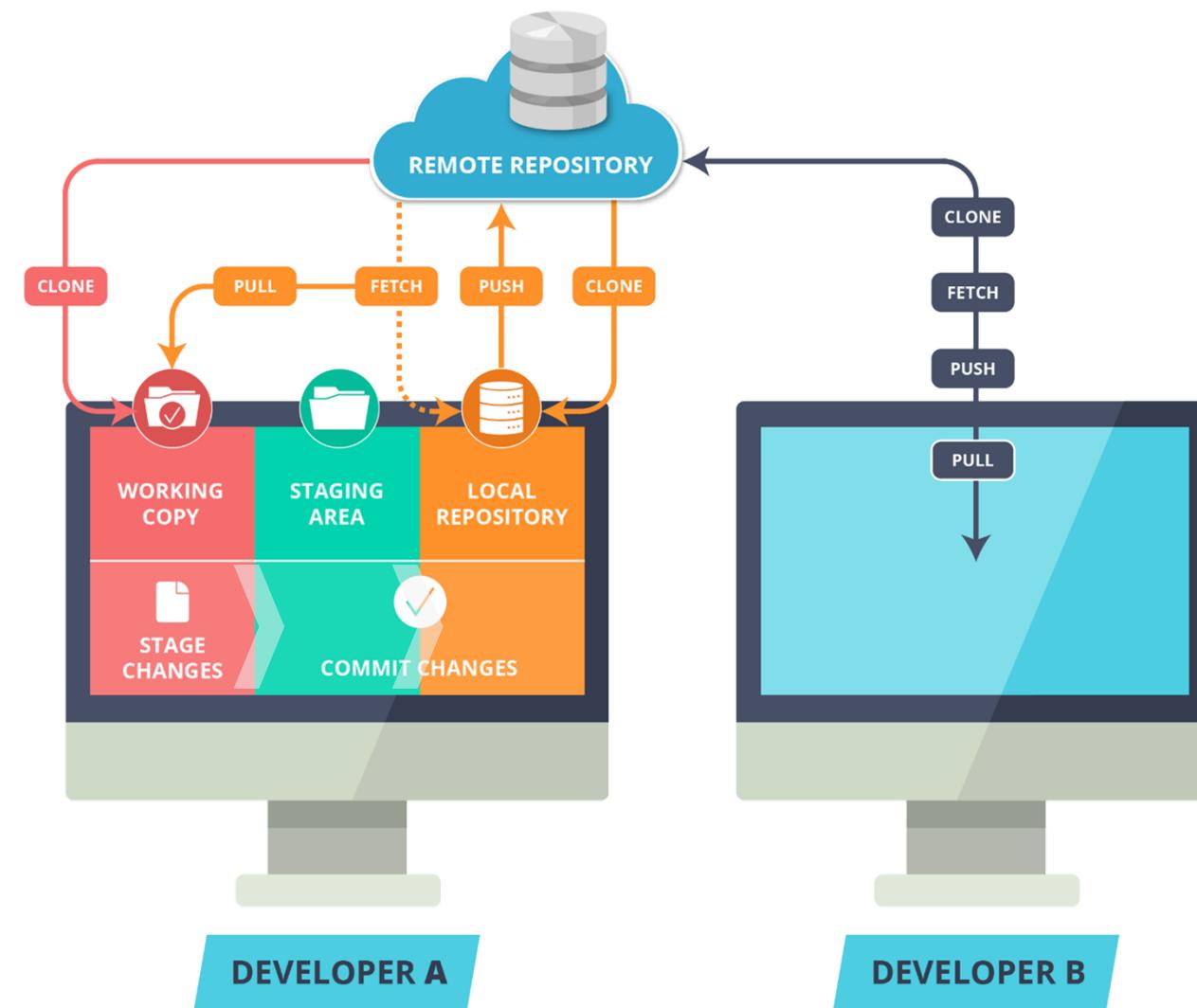
```
amr@amr-VirtualBox:~/Documents/demo$ git reset v1.5
Unstaged changes after reset:
M      RepoFile
amr@amr-VirtualBox:~/Documents/demo$
```



# Git Workflows

# Git Workflows

Depending upon the collaborators or the organization using git different workflows could be adopted to maximize productivity and consistency.

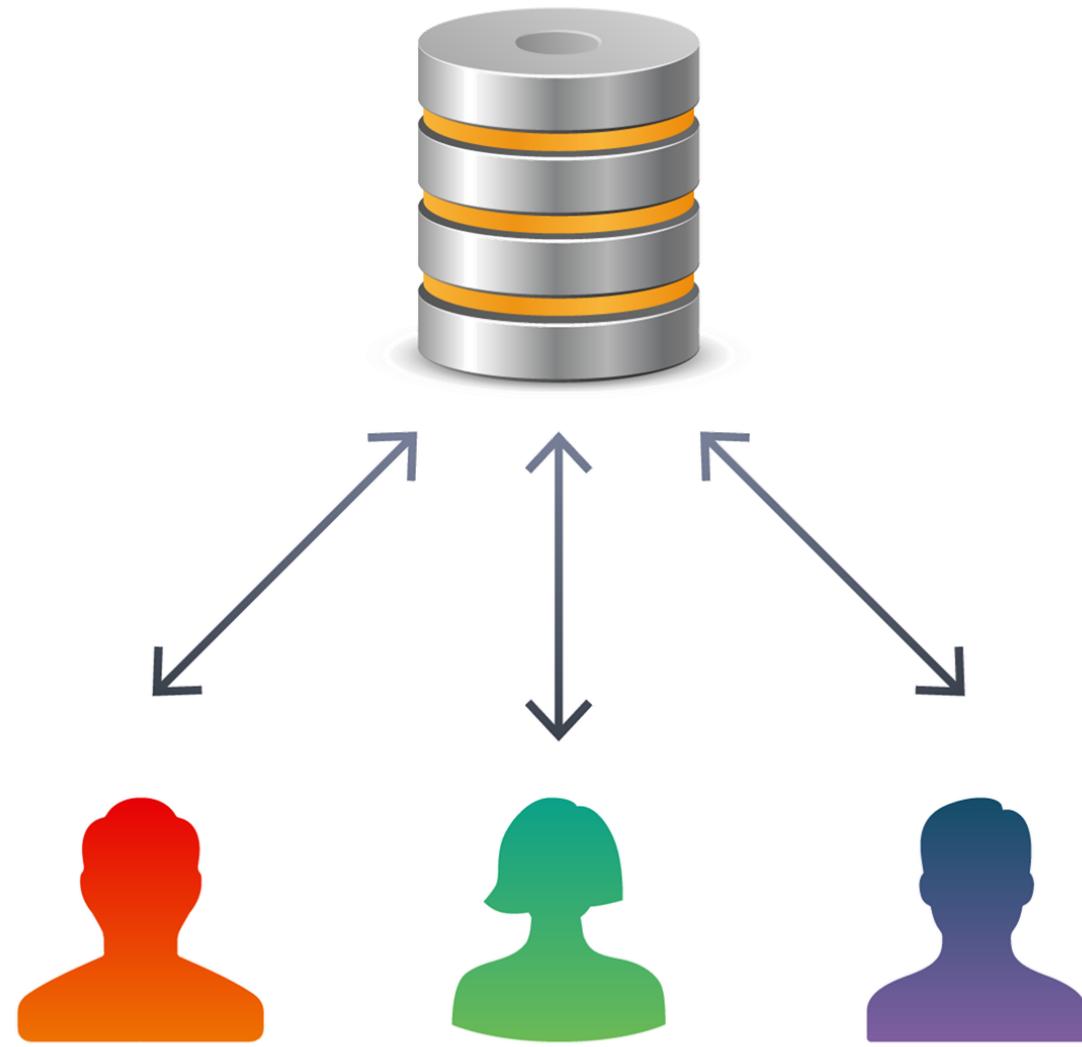


# Git Workflows

---

## Centralized Workflow

- A central repository is created on a remote server
- All the Developers clone this central repository

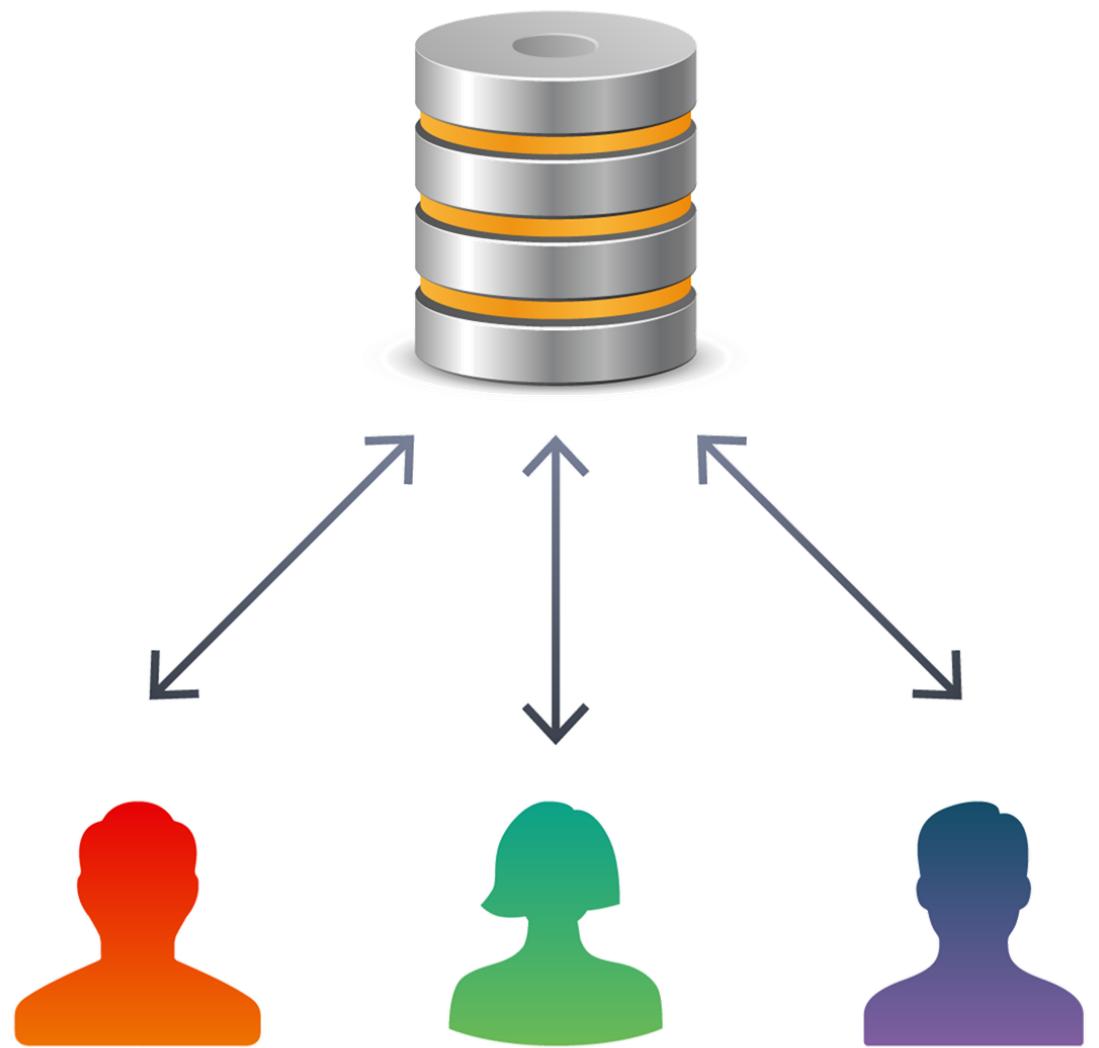


# Git Workflows

---

## Centralized Workflow

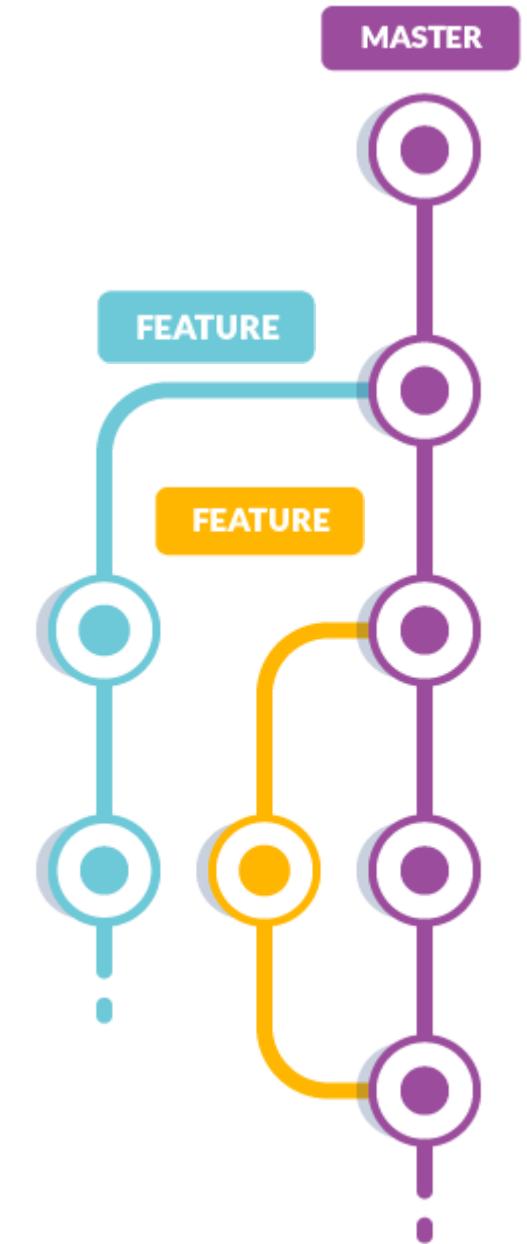
- All the changes are made locally on every users computers
- Isolated environment enables devs to work on their own and then push changes to remote repository
- Suitable for smaller enterprises where teams are small



# Git Workflows

# Feature Branching

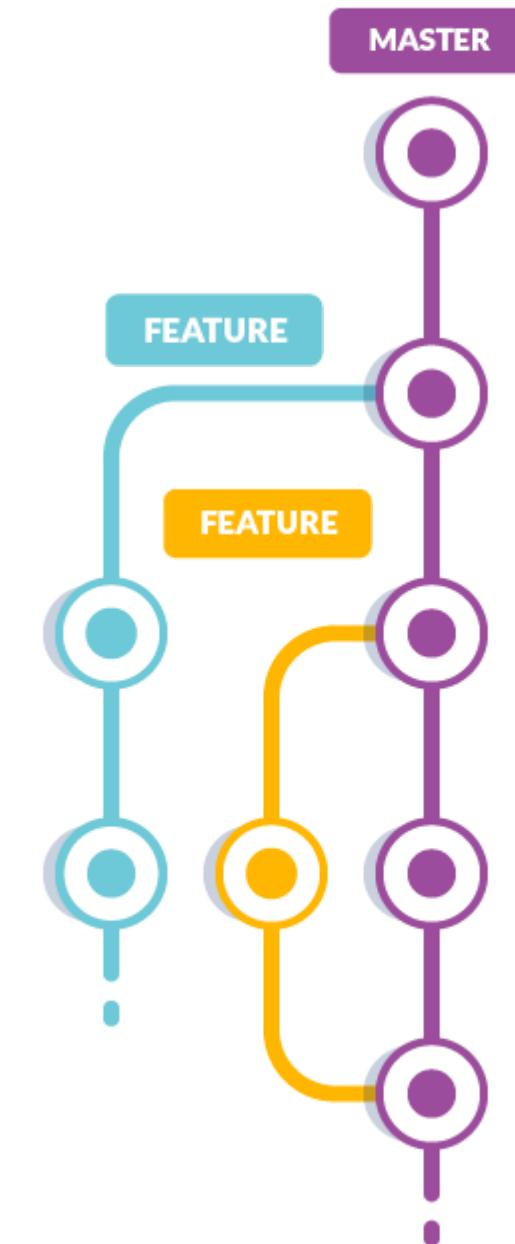
- All the development is done as features on separate branches
  - The master branch remains untouched unless all developers agree
  - Back and forth of pull requests enable developers to collaborate easily



# Git Workflows

# Feature Branching

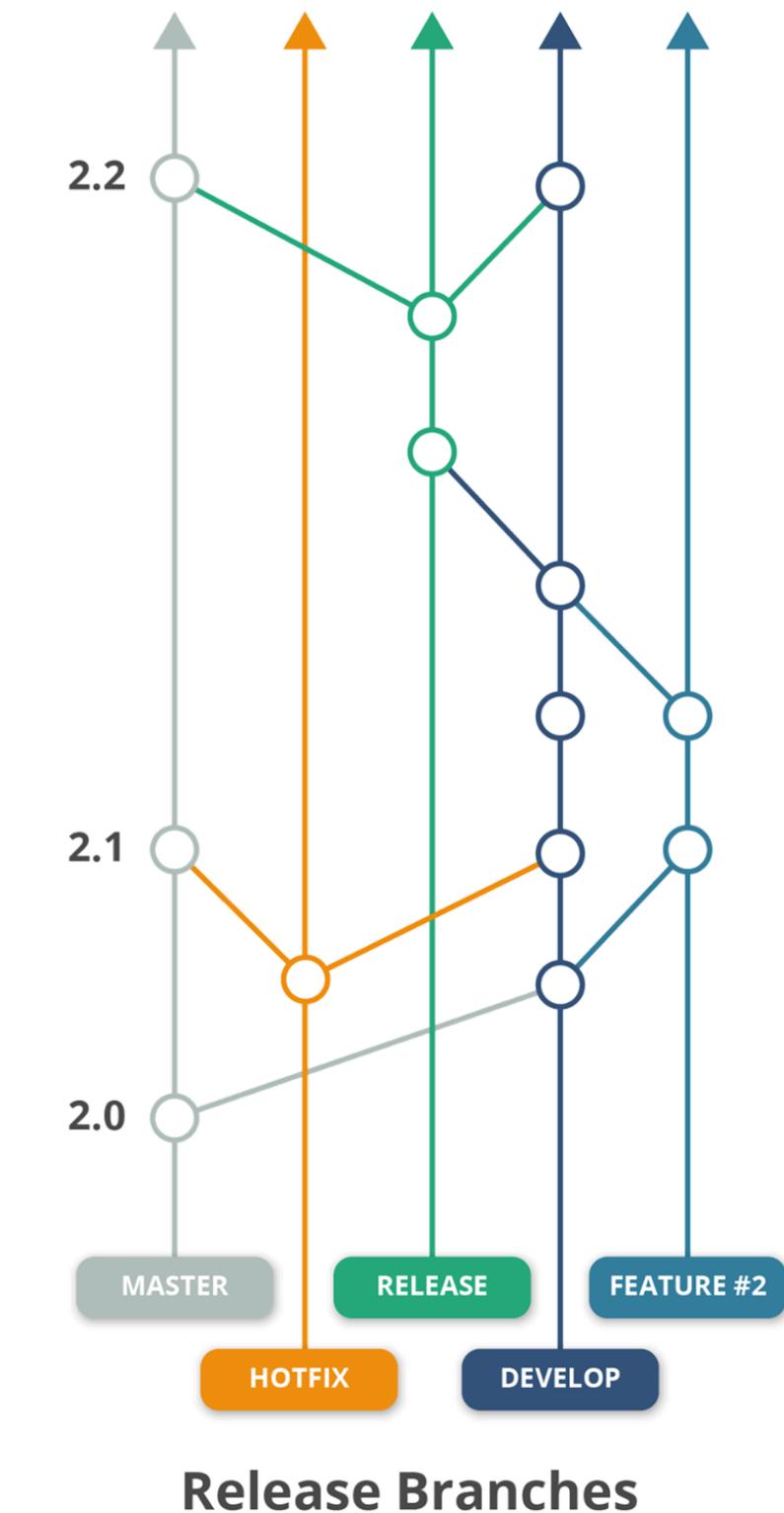
- Pull requests enable feedback as well as acknowledgement from other devs
  - Master branch never gets incomplete or broken code



# Git Workflows

## Gitflow Workflow

- The Gitflow Workflow works as an extension to the feature branching model
- In addition to feature branches, separate branches for bugfixes, maintenance, development etc are maintained
- All other features of feature branching workflow are also present

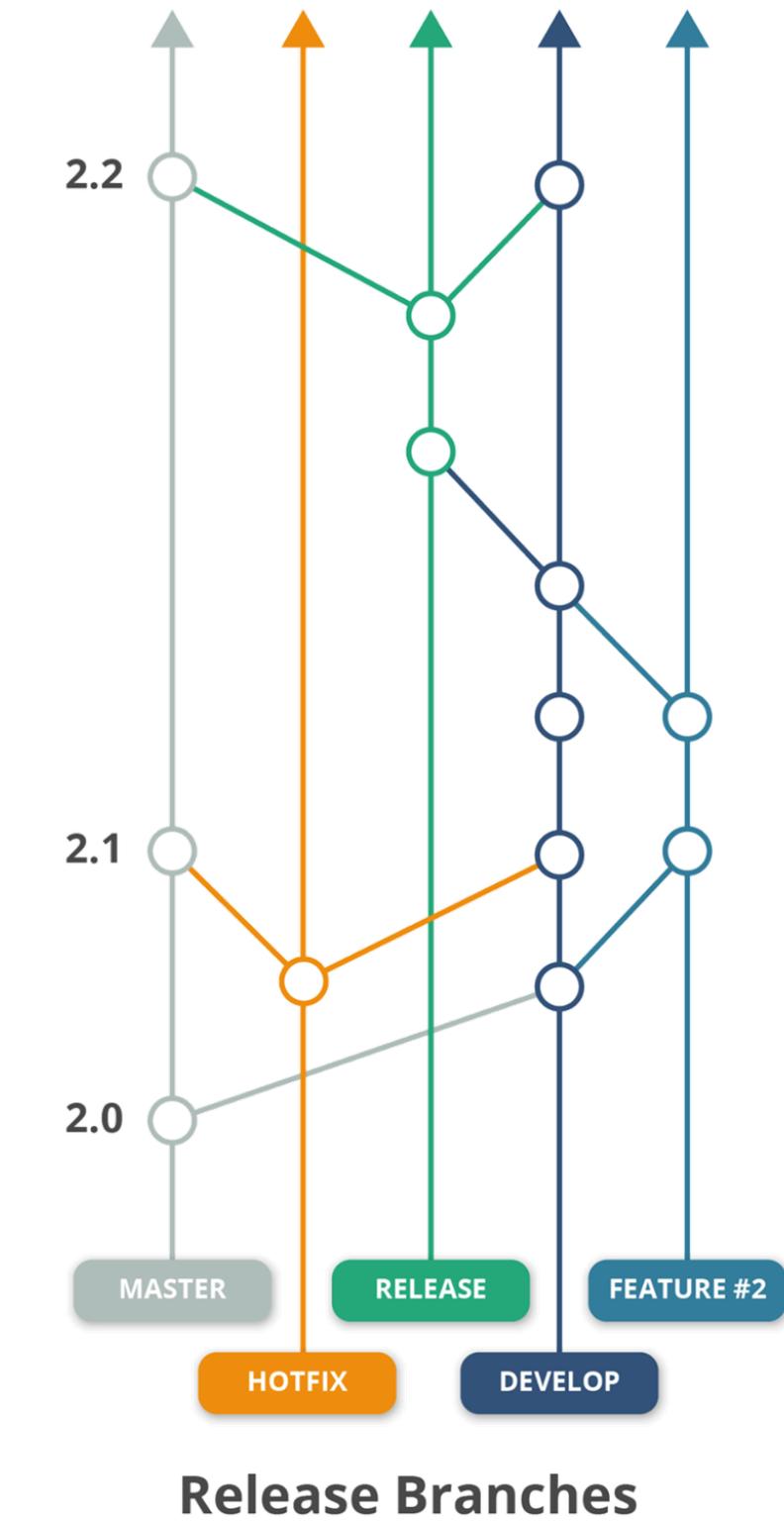


Release Branches

# Git Workflows

## Gitflow Workflow

This workflow is suitable for large organizations with emphasis on rapid software releases

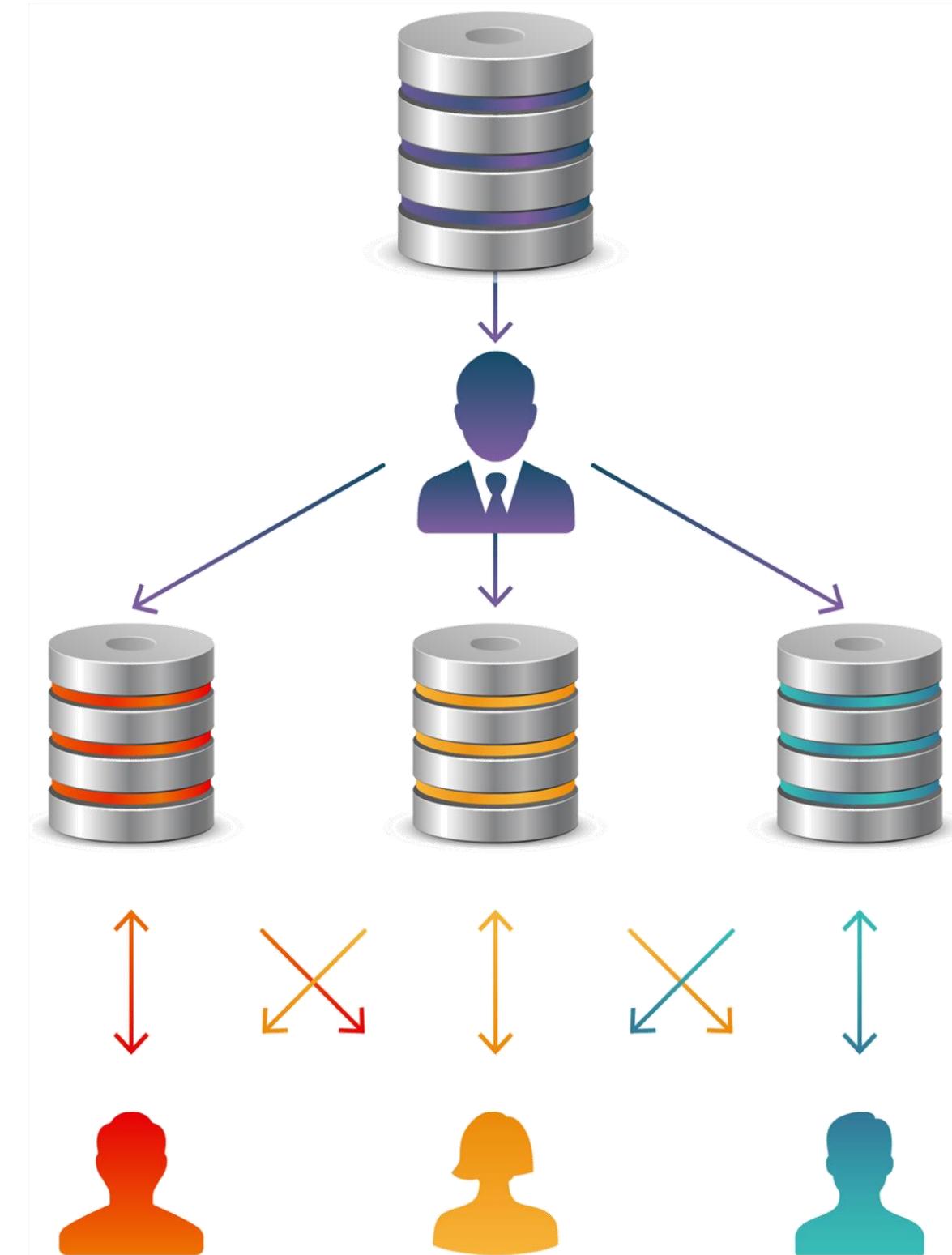


Release Branches

# Git Workflows

## Forking Workflow

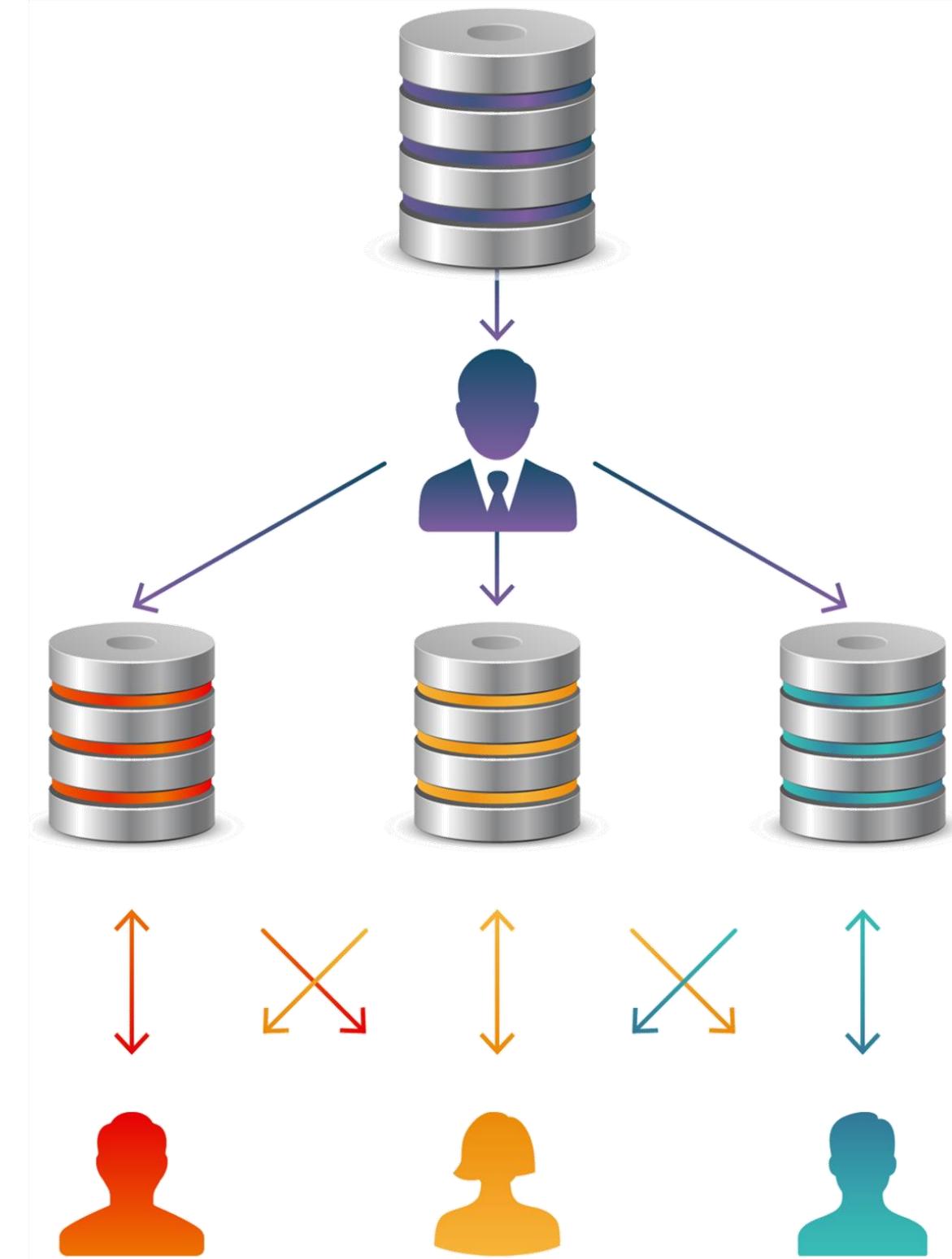
- A server side public repository is maintained by a project lead
- Every collaborator gets his own private repository(clone of public)
- Developers work on their own private repositories

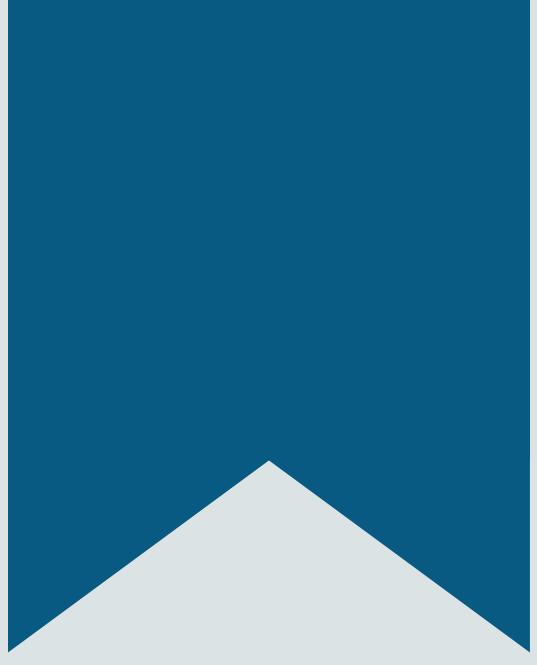


# Git Workflows

## Forking Workflow

- The project maintainer chooses and accepts code from other collaborators
- Only the maintainer can upload changes to the public repository





# Introduction To Maven

# What is Maven?

---

“Maven is a tool that is used to compile, validate codes, and analyse the test-cases in the code.”

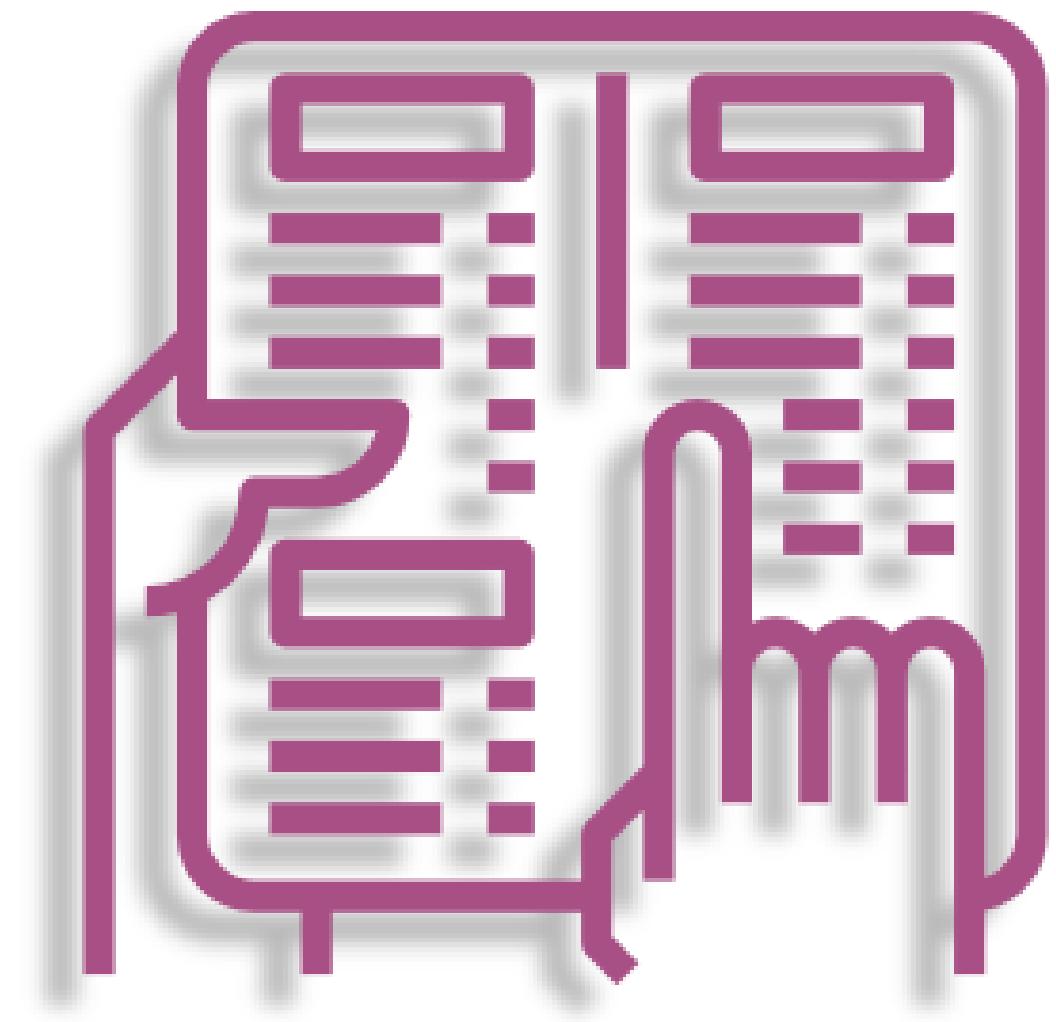
- Manages the building, reporting and documentation from Source Control Management (SCM)
- Maven projects are configured through Project Object Model (POM)
- pom.xml file contains documentation for all of the objects, properties, methods, and events



# What Maven Is Capable of ?

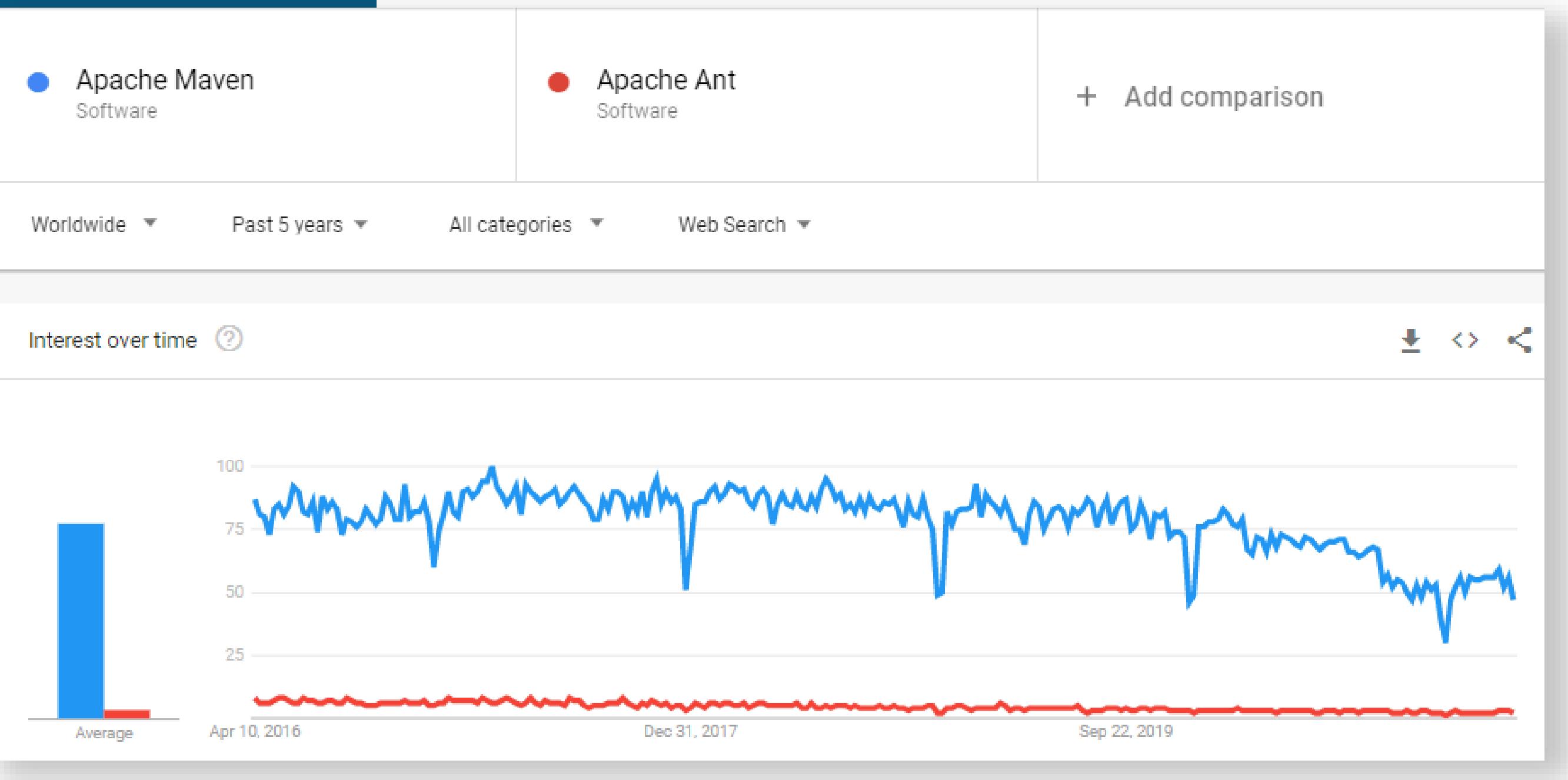
---

- Information of project is centralized through Maven
- Our software project is modeled by it
- Build process is managed
- Data about the software project is gathered and then build itself
- Document the software, and our project
- Deployable artifacts can be generated from source code
- Your source code is compiled, packed, test and distributed
- Reports is created, website is generated for the project



# Apache Maven Vs Apache Ant

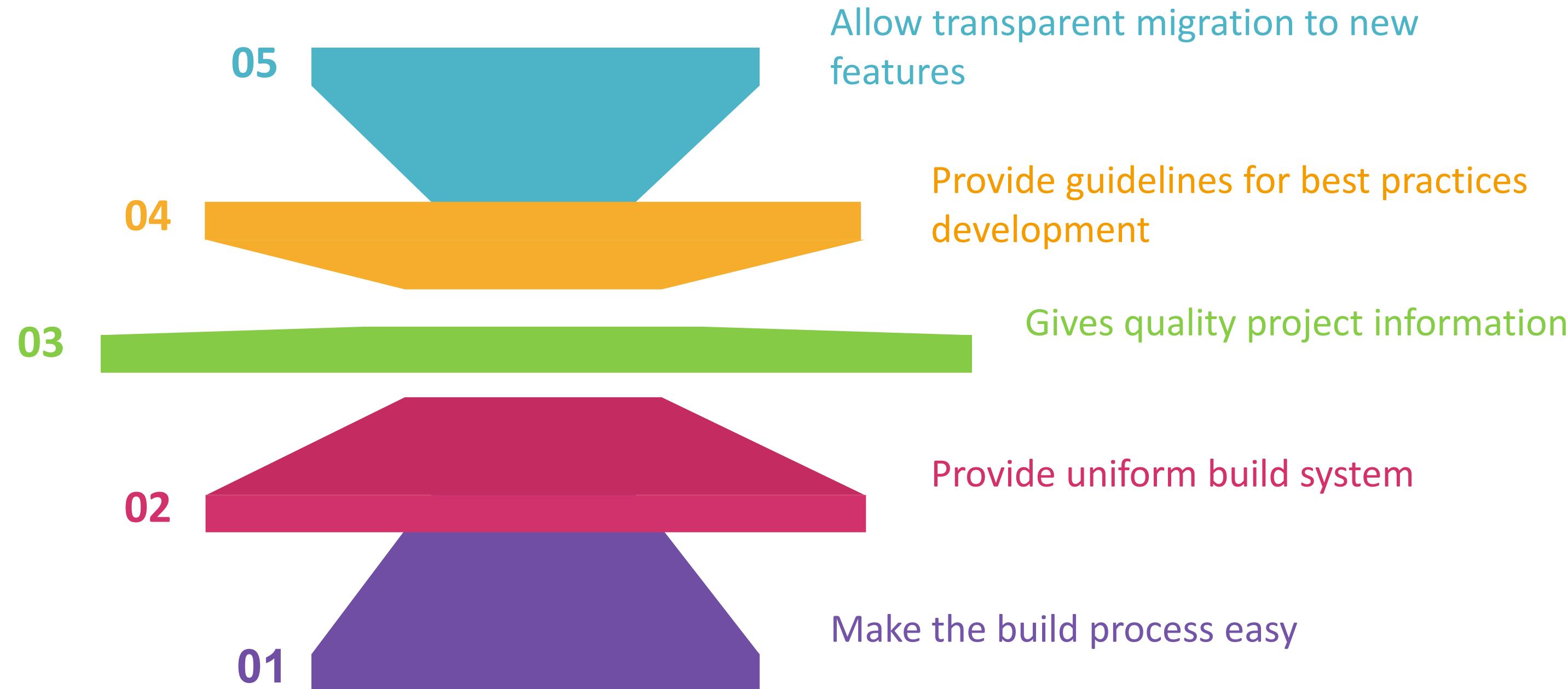
## Google Analytics



It is significant that Apache Maven is far more popular than Apache Ant.

# Objectives Of Maven

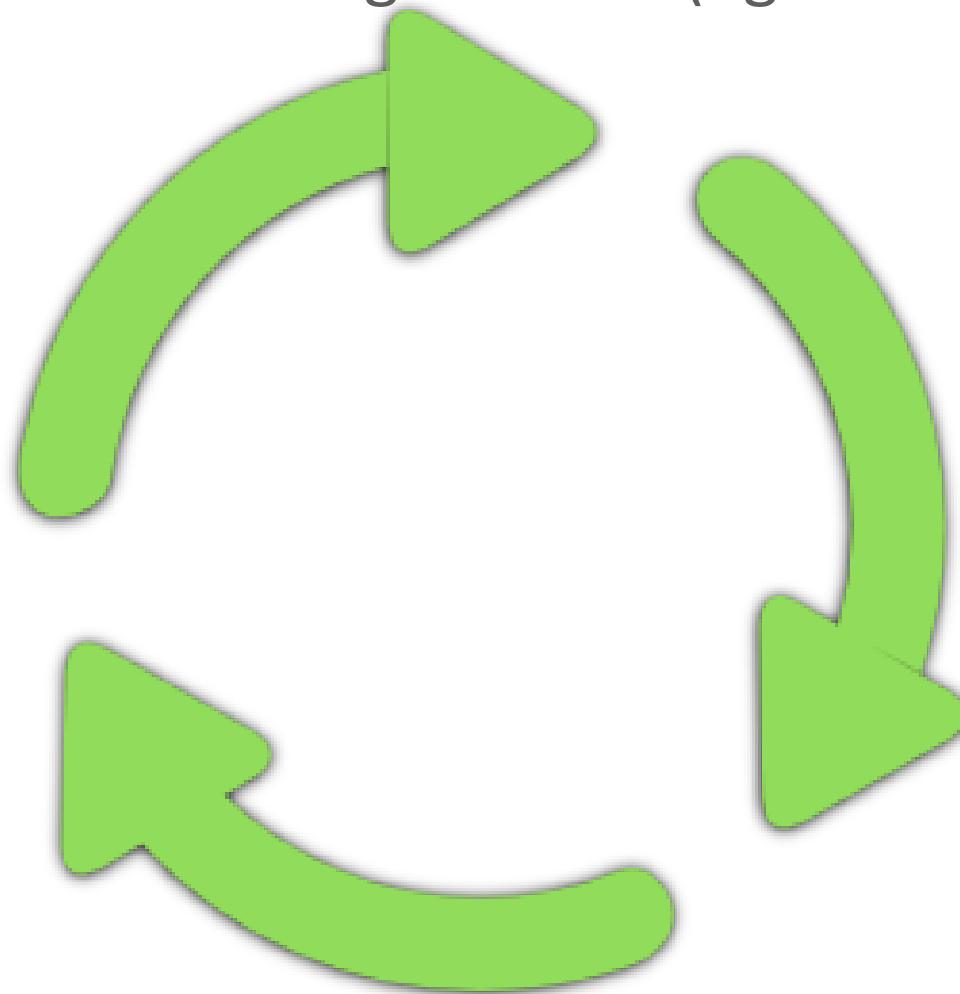
---

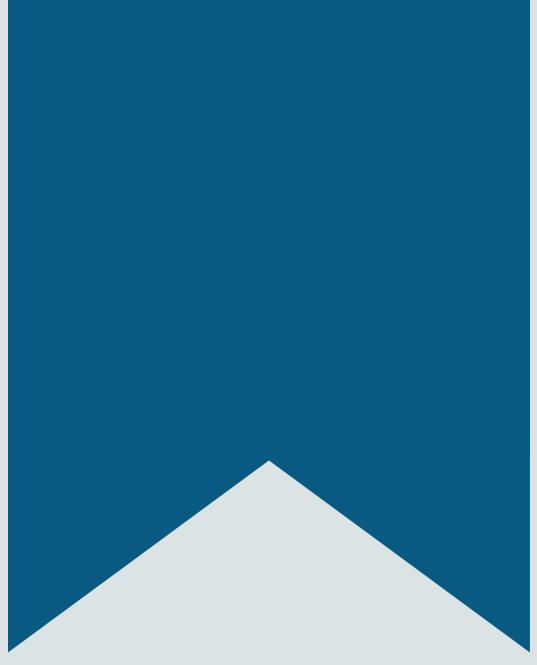


# Maven Build Lifecycle

---

- In Maven, the build is run using a predefined and ordered set of steps to call the build life-cycle
- The build tasks that will be performed during each phase are determined by the configuration in the project file, and in particular the selected packaging
- Maven relies on build lifecycles to define the process of building and distributing artifacts (eg. Jar files, war files)
- There are three built-in build lifecycles
  - Default – handles project building and deployment
  - Clean – handles project cleaning
  - Site – handles project's site generation

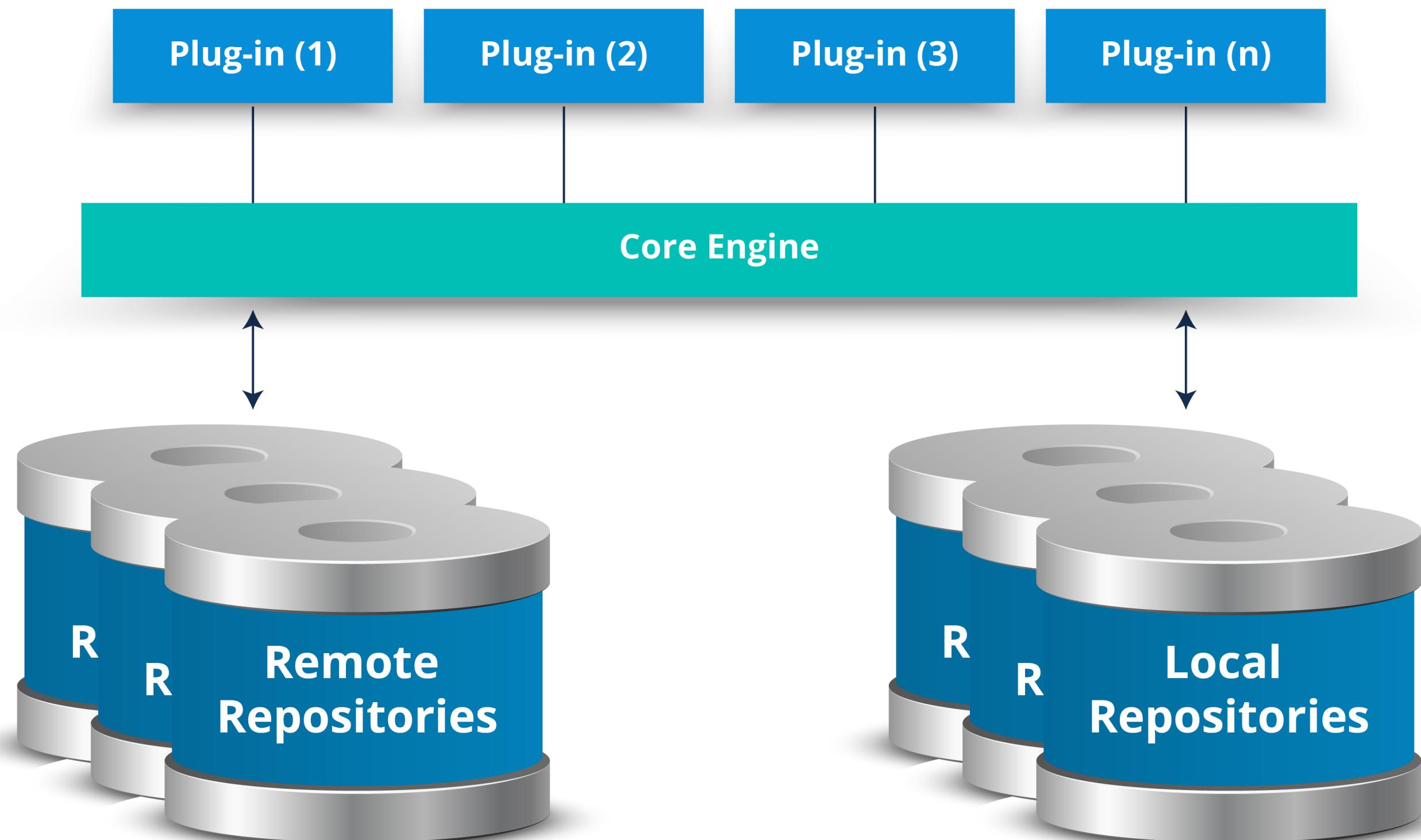




# Maven Architecture

# Maven Architecture

---



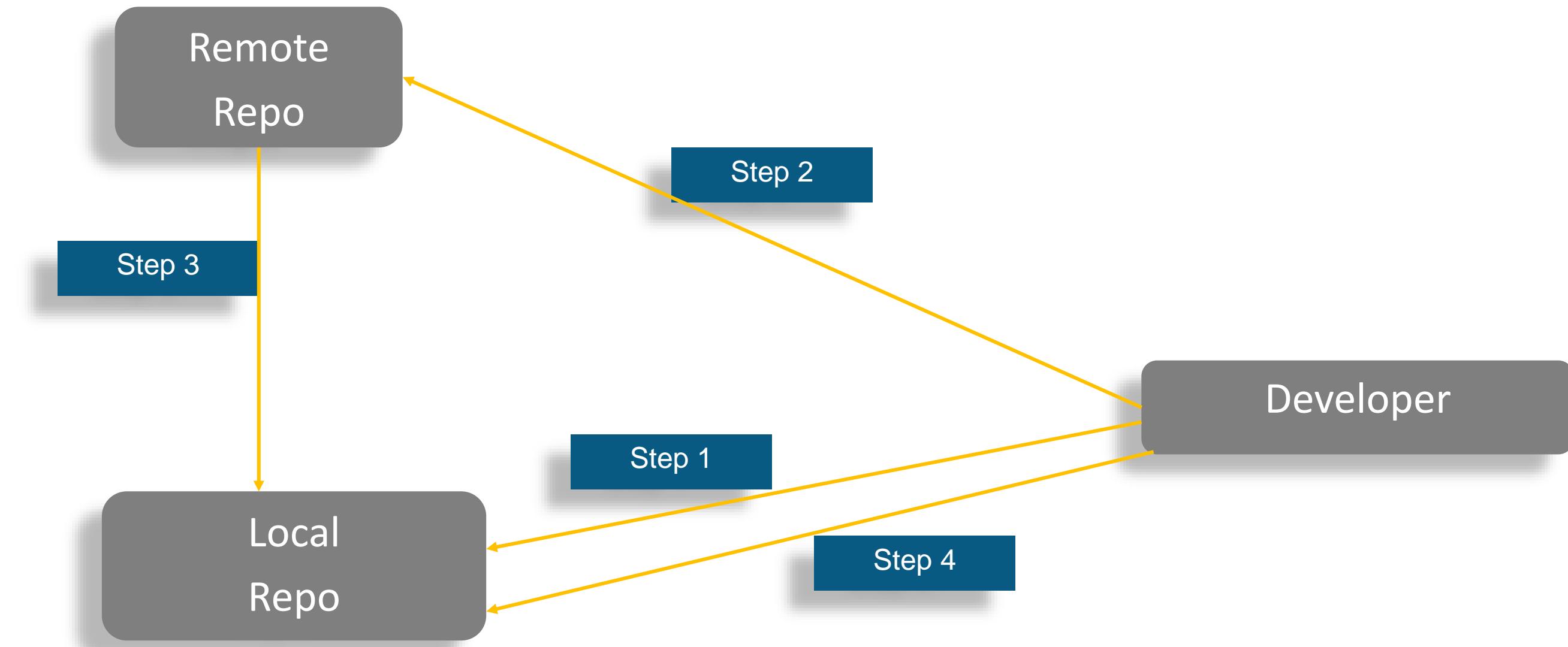
# Maven Artifacts

---

- An artifact is a file resulting from packaging a project
- Can be a jar, war, ear, .xml file, for example
- Artifacts are deployed in repositories, so they can be used, as dependencies, by other projects
- Artifacts are identified by three components:
  - Group Id → An unique identifier for a group of related artifacts. Usually named like Java packages (Ex: pt.jpereira.mobile)
  - Artifact Id → An unique identifier, within the context of GroupId, that identifies the artifact (project). (Ex: puzzle)
  - Version → Also called artifact coordinates

# Maven Repository Flow

---



# Maven Repository

---

Maven repository stores all the versions of all the dependencies

There are three types of maven repository:

- Local : It is located in the developers machine. By default, it is created in the home directory and whenever dependencies are downloaded from remote server, it is saved into the developers machine
- Central Repository : The location of this repository is <http://repo.maven.apache.org/maven2/> . Whenever build job is run, maven try to find it from local repository but when it's not there, they download the dependencies from central repository
- Remote : Sometimes, there are objects and modules which are specific to organization, for which remote repositories are created which are accessible within the organization.



# Project Object Module

---

- POM for short
- XML file located at the root of the project (pom.xml)
- It includes configuration for your project, including:
  - Information about the project
  - Configuration details to build the project
  - Contains default values for most of the projects. Ex: Source dir, target dir
  - Dependencies of the project
  - Configuration about plugins and goals
  - Used repositories

# Dependencies

---

- Maven comes into the picture for dependencies
- Manages hundreds of plug-ins
- Dependencies get cached into the local repository
- Manages the transitive dependencies
- Provides multiple scope for including dependencies

# Declaring Dependencies

---

While declaring the dependencies, coordinates of the artifacts must be provided:

- GroupID : A unique identifier for the group of related artifact
- ArtifactID : An artifact is a file which is made after packaging a project and ArtefactID is a unique identifier within the context of GroupID and it identifies the artifacts
- Version : It is an identifier for the release or build number of project

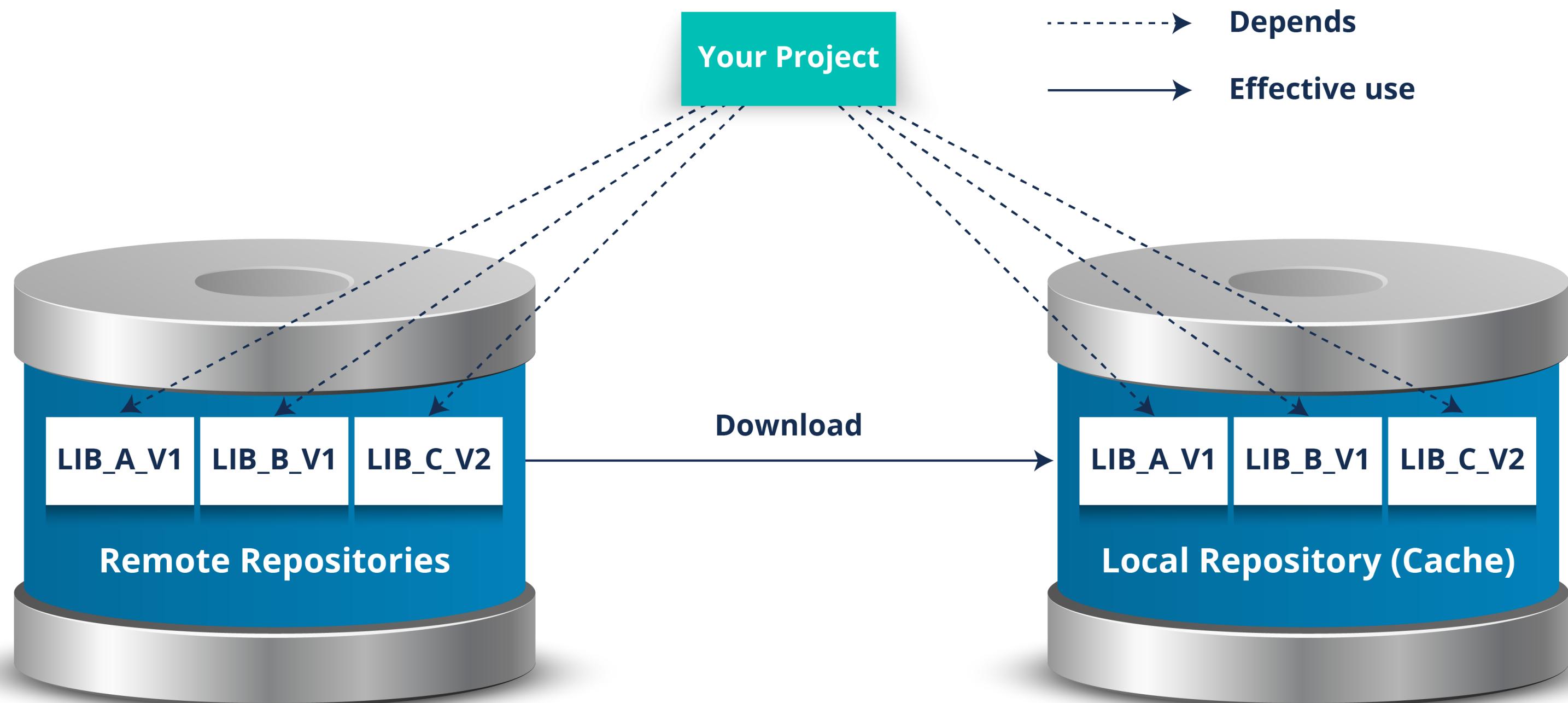


# Dependency Management

---

- It has mechanism to centralize dependency information
- Maven has all information about dependencies in the parent POM (version, exclusions, scope, type)
- Child POMs only have to have a simple reference to the dependencies, excluding version, exclusions, scope type

# Cache Dependencies



# Maven Through Terminal

---

- Install the Maven tar file after verifying the signature
- After installing the Maven, check its version from ubuntu terminal by using the below command
  - \$mvn -version
- Below command will create a project using Maven
  - \$mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
- The generate goal will create a directory having the same name given as the artifactId

# Build The Package In Maven Through Terminal

---

- Below command will build the Maven project
  - \$ mvn package
- The newly compile and build project can be tested by using the below command
  - \$java -cp target/my-app-1.0 proj\_name.jar com.mycompany.app.App
- The above command will print the output of the project

# Maven Lifecycle Phases

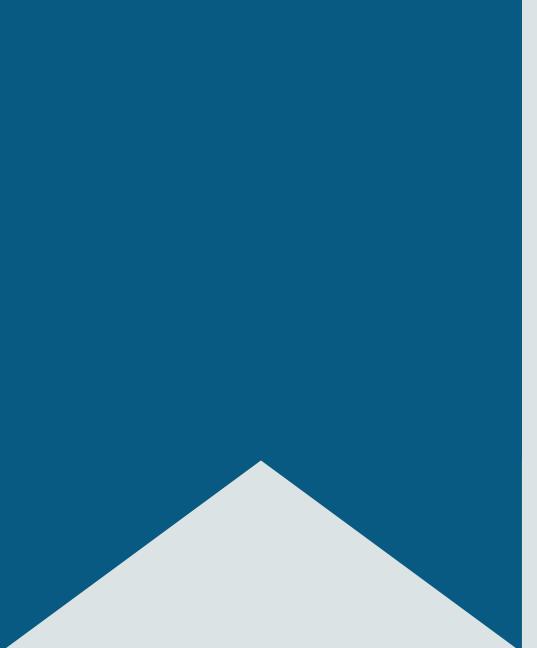
---

- **validate**: check if the project is correct, having all the necessary information
- **compile**: compile the source code of the project
- **test**: test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- **package**: take the compiled code and package it in its distributable format, such as a JAR

# Maven Lifecycle Phases

---

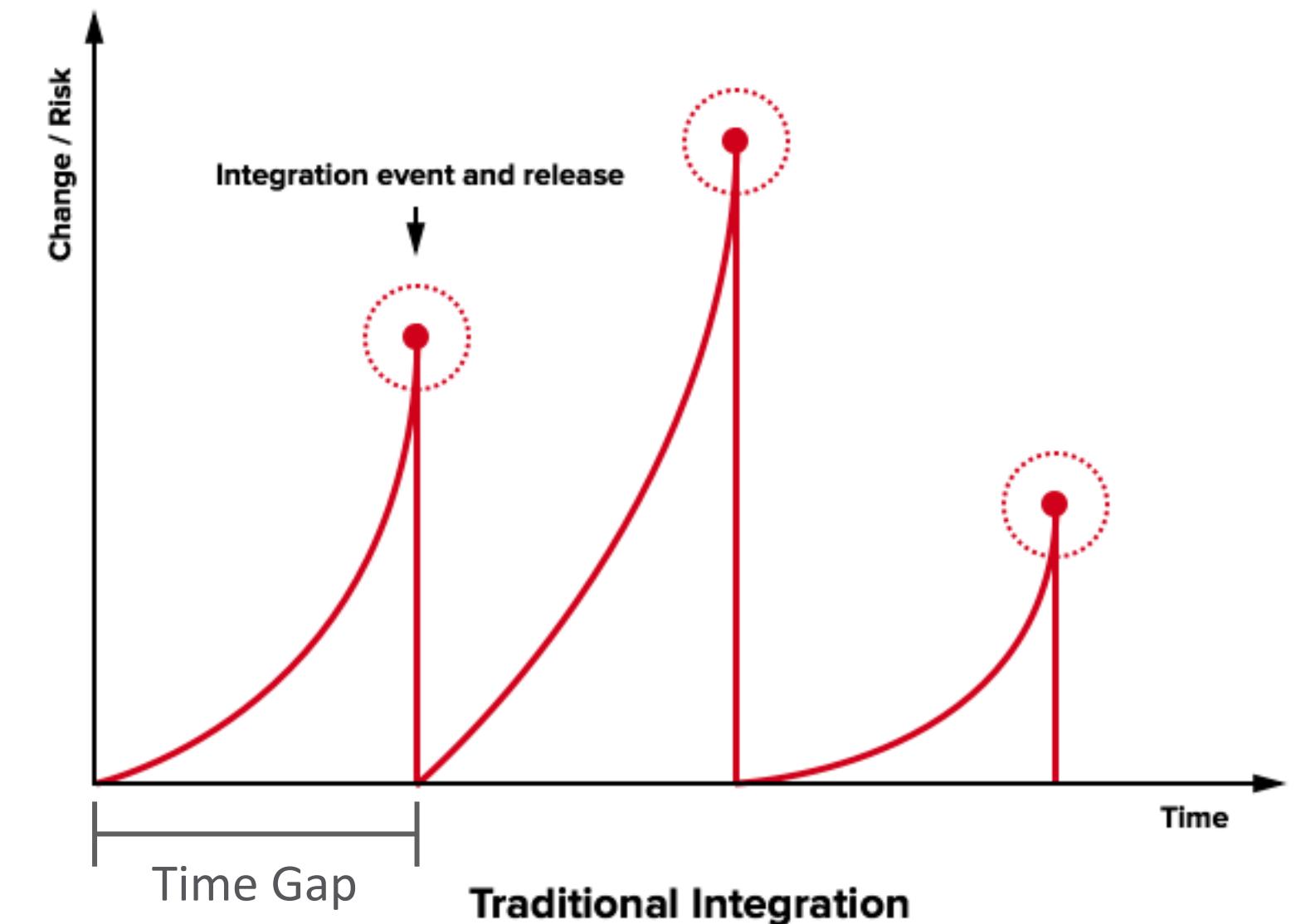
- **integration-test**: process and deploy the package into an environment where integration tests can be run
- **verify**: run any checks to verify the package is valid and meets quality criteria
- **install**: install the package into the local repository, for use as a dependency in other projects locally
- **deploy**: done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.



# Continuous Integration & Its Importance

# Traditional Integration

- Greater time gap in case of Traditional Integration
- Relatively greater risk or change in conflicts



# Problems With Traditional Integration

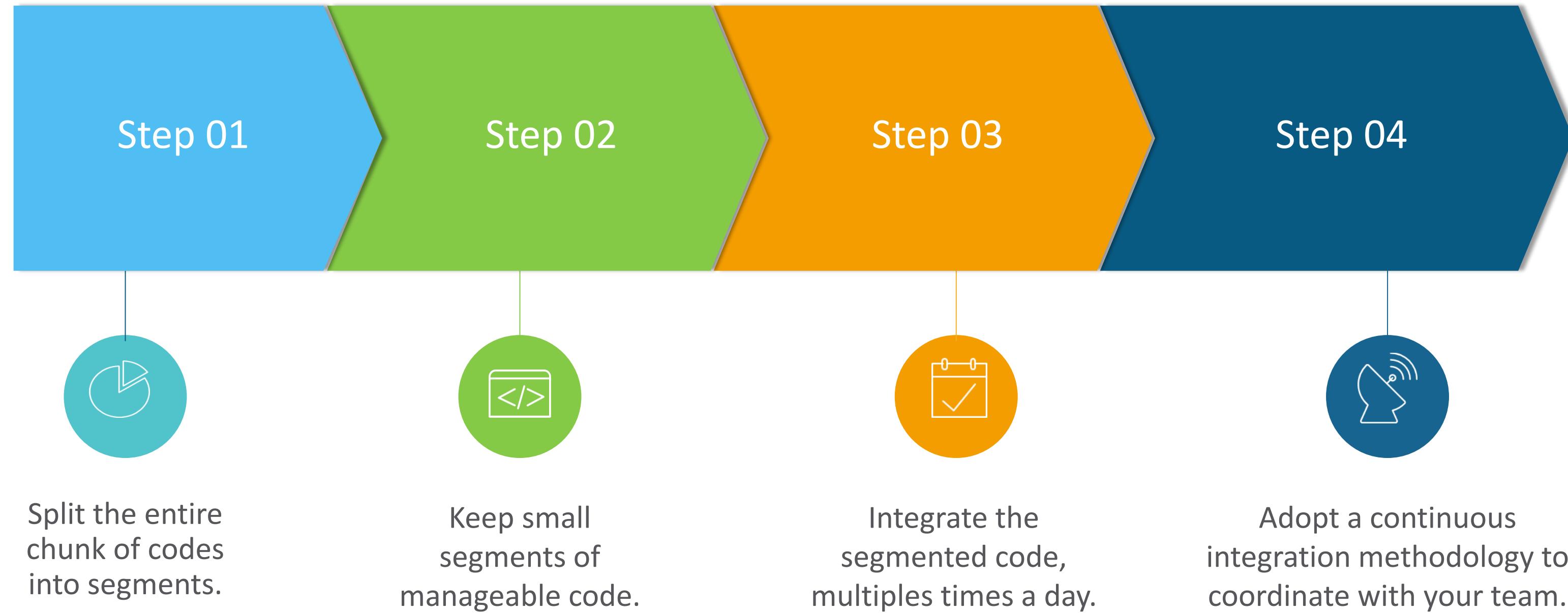
---

- Difficult to Implement all the details that have changed
- Difficult to get the latest version of the code early
- Difficult to manage all the changes with the changing API versions.
- Difficult to manage the entire subsystem of the application that behave differently
- Difficult to rewrite the whole code



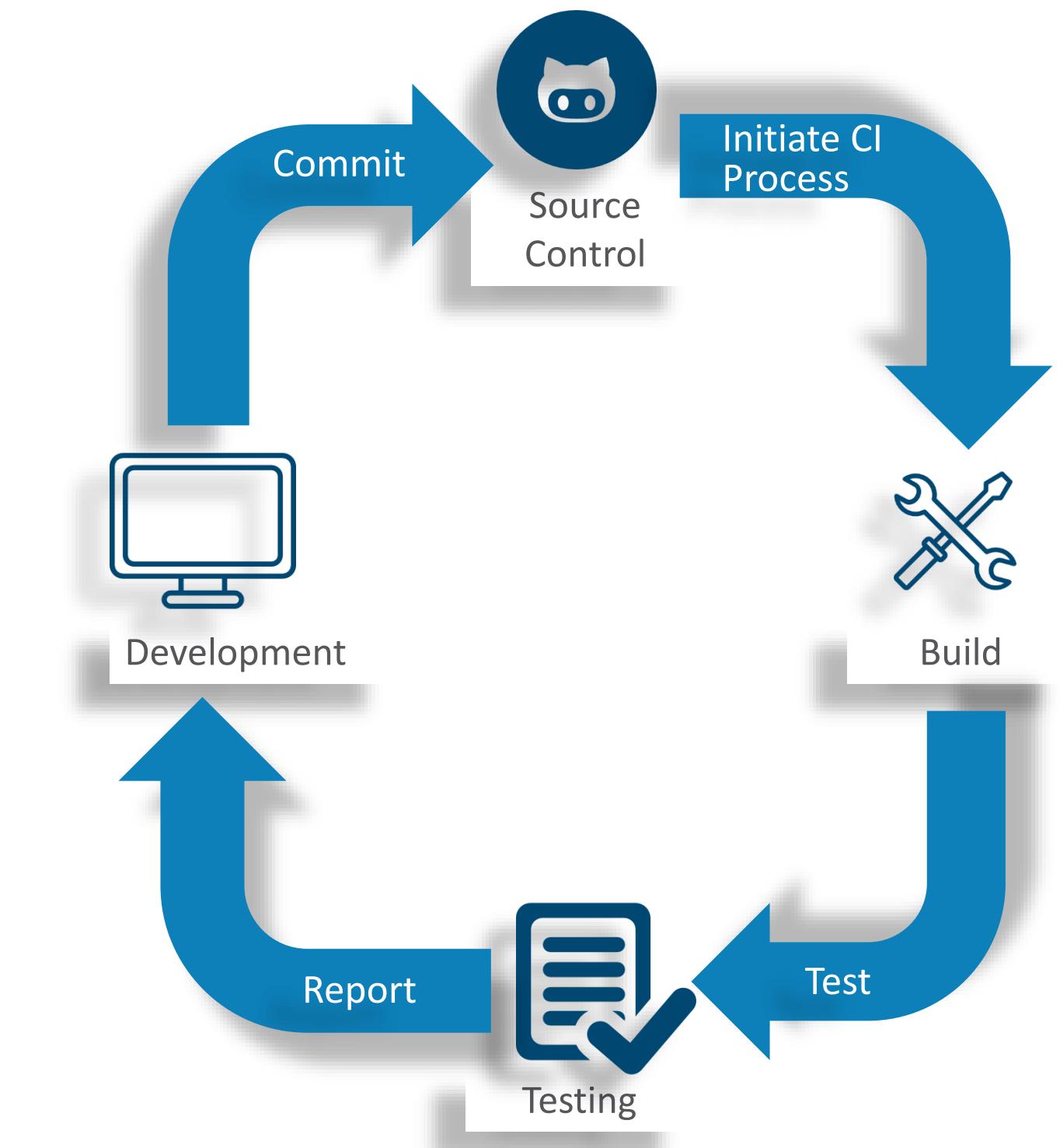
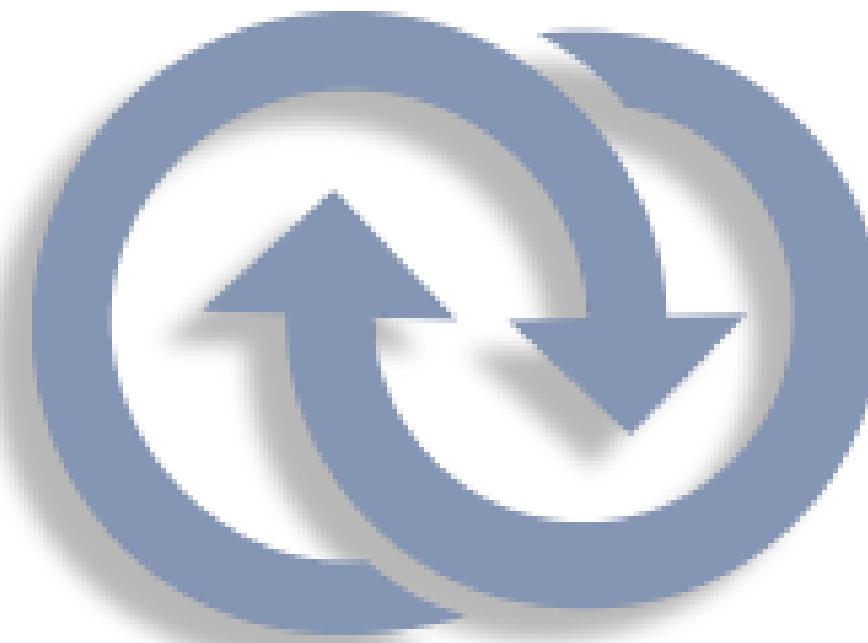
# How To Solve The Problem?

---



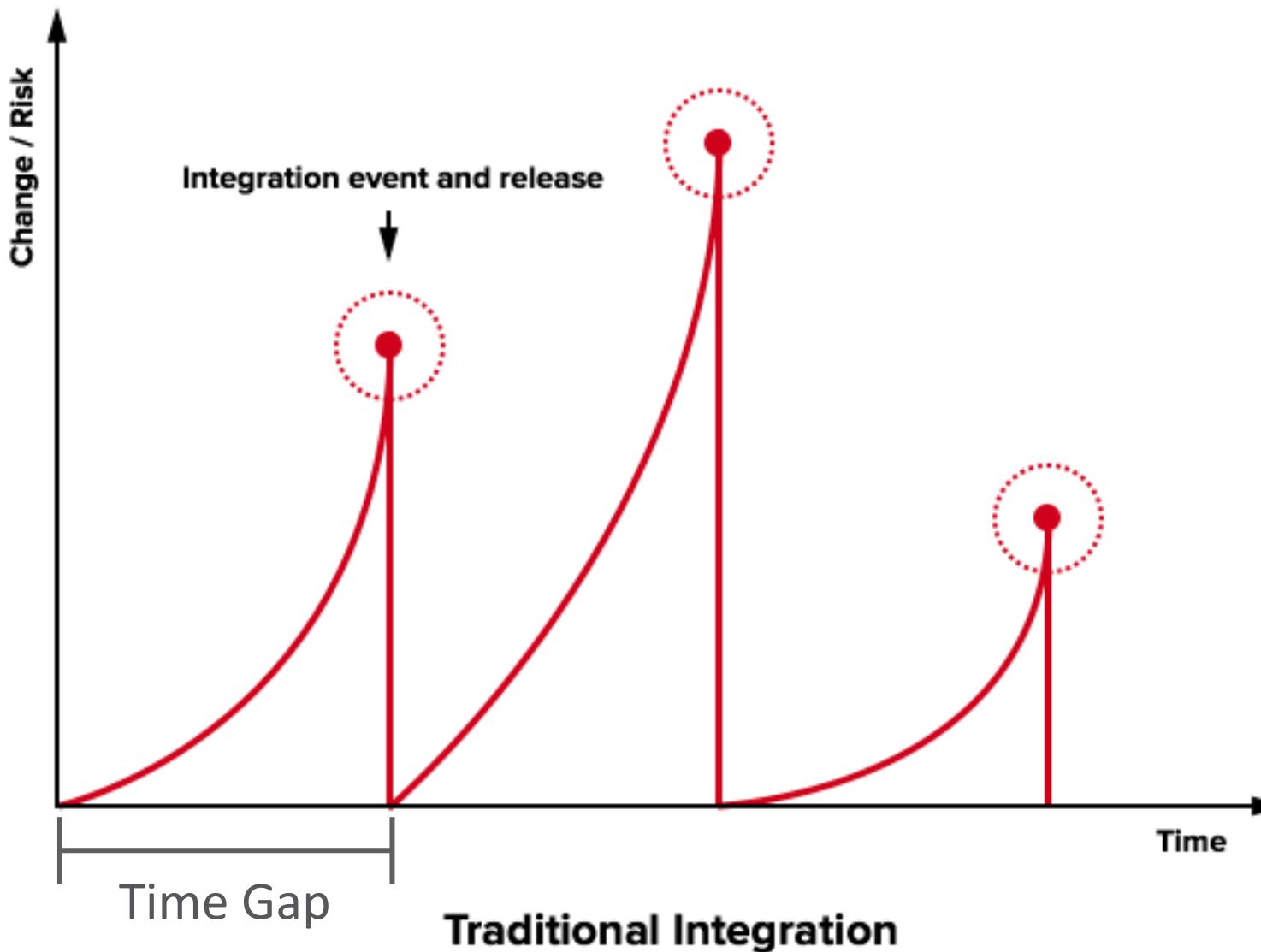
# What Is Continuous Integration?

“It is the process of automating the building and testing of code, each time one of the team member commits changes to version control.”

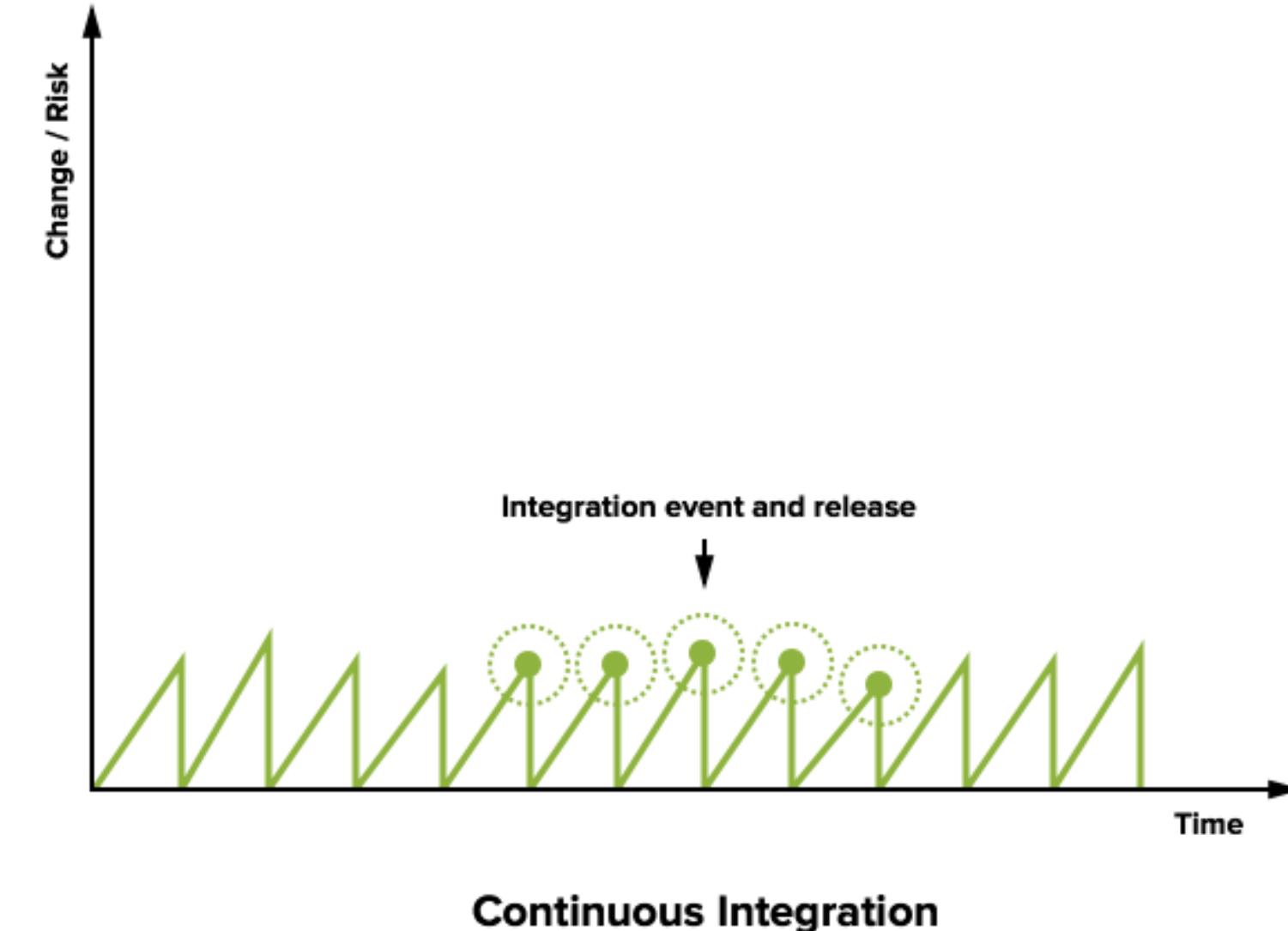


# Traditional Integration Vs Continuous Integration

Traditional Integration



Continuous Integration



- Greater time gap in case of Traditional Integration
- Relatively greater risk or change in conflicts

- Integration time gap is relatively much lesser
- Relatively lesser risk or change in conflicts

# Importance Of Continuous Integration

---



**Improves Quality**  
Improves the quality by running multiple unit tests and analysing various static code

01

02

Increases Productivity

Automating build of code saves a lot of time, thereby increasing productivity



03

**Reduces Risk**

Eliminate the risk of potential human errors by automating test

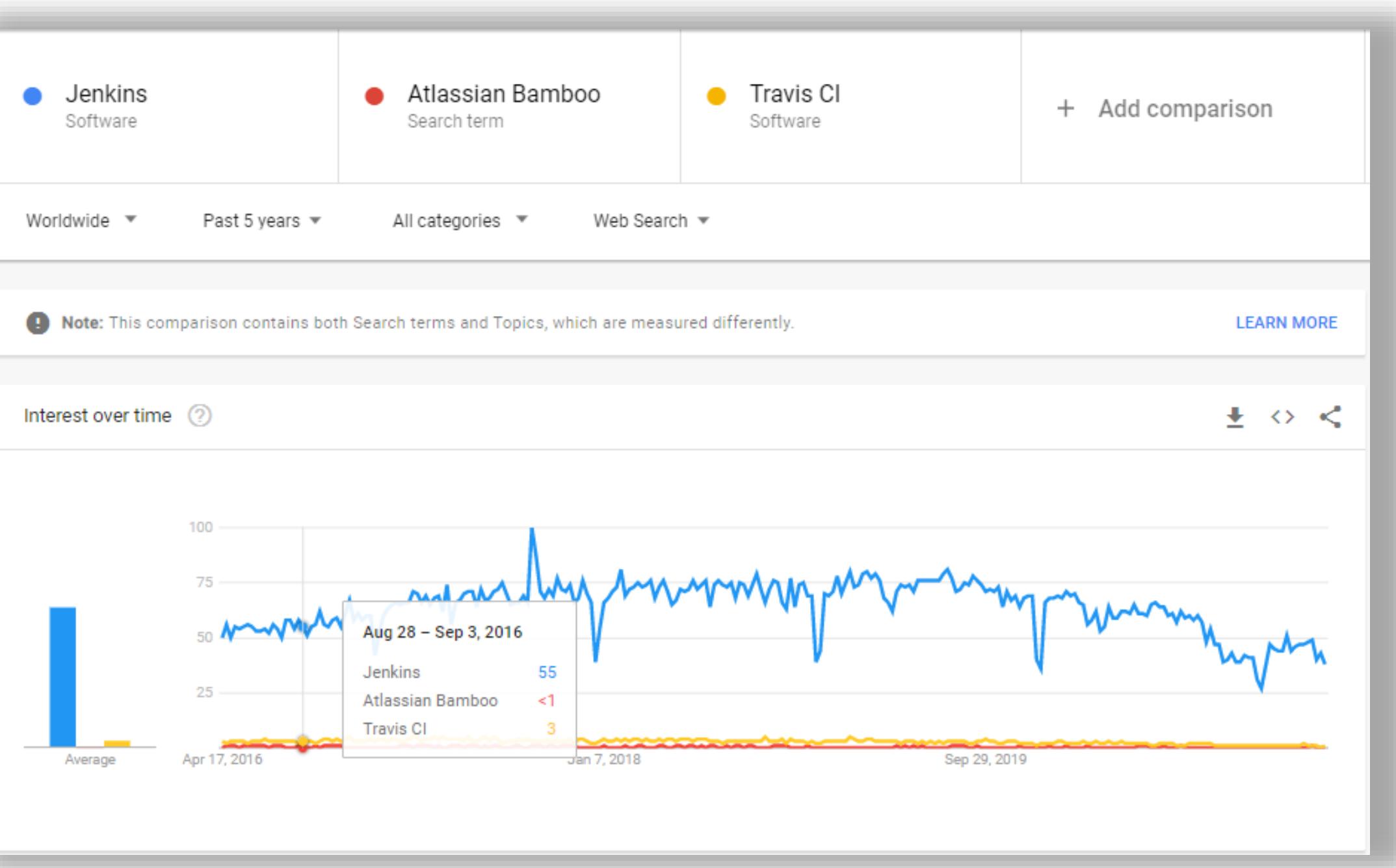


# Popular Continuous Integration Tools

---



# Jenkins Popularity: Google Trend

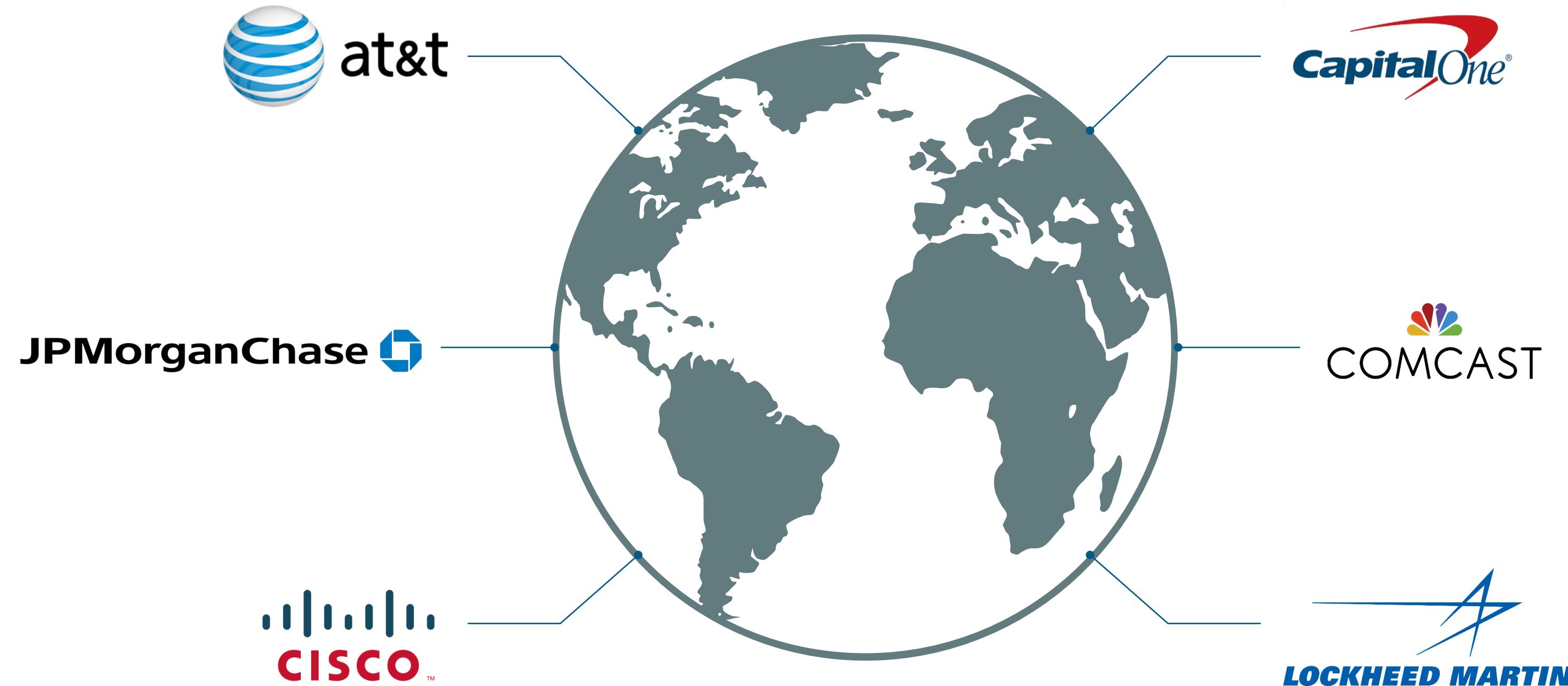


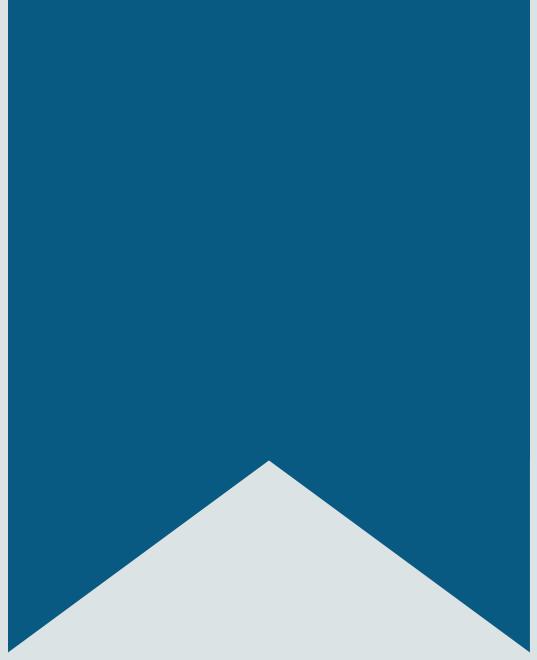
the popularity of Jenkins is consistent and its use has increased in the past years

Google Trends: Jenkins VS Travis VS Atlassian Bamboo

# Companies Using Jenkins

---



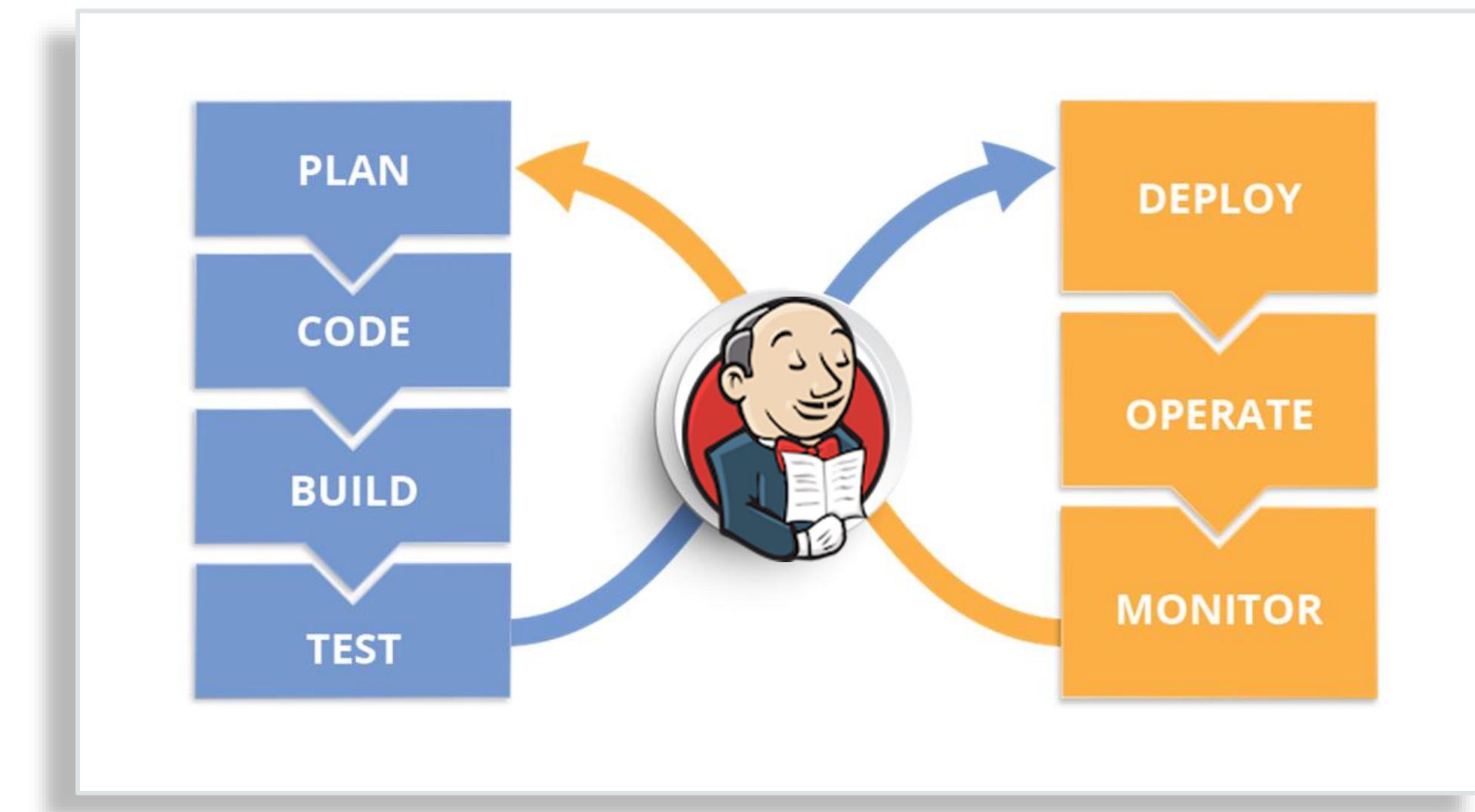


# Introduction To Jenkins

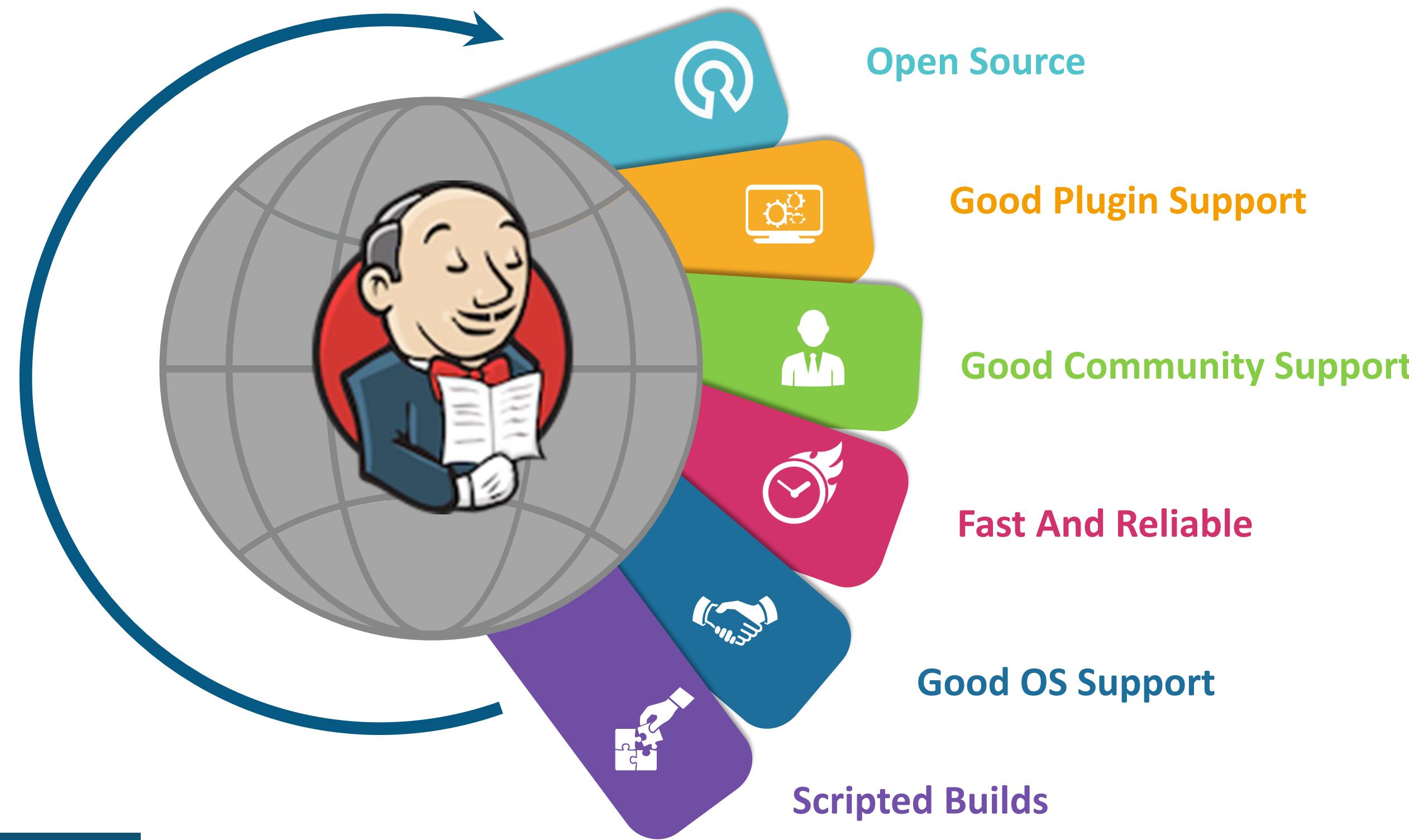
# What Is Jenkins?

---

“A Continuous Integration server which manages and control processes such as plan, code, build, test, deploy, operate and monitor in DevOps Environment.”

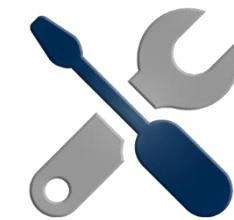


# Why Jenkins Is So Popular?

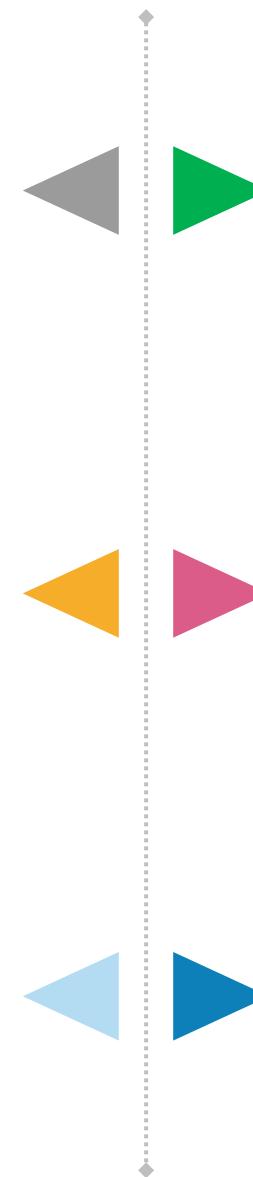


# Features Of Jenkins

---



Easy Installation Process



Provides advance security



Optimized Performance

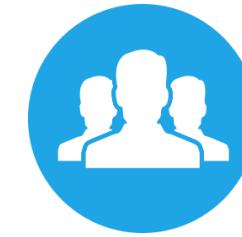
Upgrades are easily available



Lightweight container support



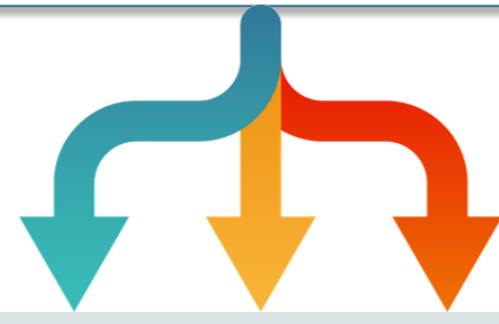
Distributed Team Management



# Summary

## Branching

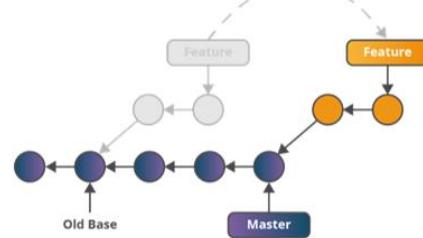
A project in its development could take multiple different paths to achieve its goal. Branching helps us take these different directions, test them out and in the end achieve the required goal.



Copyright © edureka and/or its affiliates. All rights reserved.

## Branch Rebasing

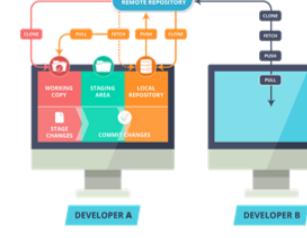
Git rebasing is used, when changes made in one branch needs to be reflected in another branch.



Copyright © edureka and/or its affiliates. All rights reserved.

## Git Workflows

Depending upon the collaborators or the organization using git different workflows could be adopted to maximize productivity and consistency.



Copyright © edureka and/or its affiliates. All rights reserved.

## What is Maven?

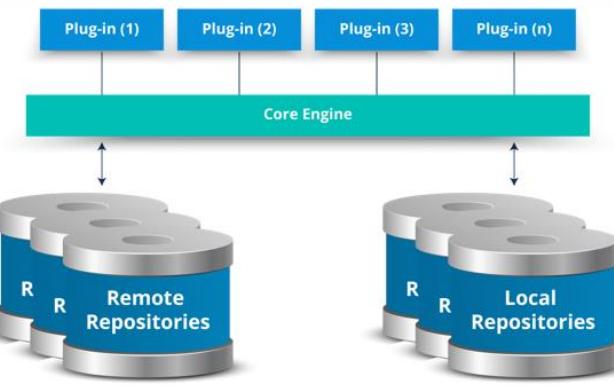
"Maven is a tool that is used to compile, validate codes, and analyse the test-cases in the code."

- Manages the building, reporting and documentation from Source Control Management (SCM)
- Maven projects are configured through Project Object Model (POM)
- pom.xml file contains documentation for all of the objects, properties, methods, and events



Copyright © edureka and/or its affiliates. All rights reserved.

## Maven Architecture



Copyright © edureka and/or its affiliates. All rights reserved.

## What Is Continuous Integration?

"It is the process of automating the building and testing of code, each time one of the team member commits changes to version control."



Copyright © edureka and/or its affiliates. All rights reserved.

# Questions

# FEEDBACK



Survey



Ratings



Ideas



Comments



Suggestions



Likes



# Thank You

---

For more information please visit our website  
[www.edureka.co](http://www.edureka.co)