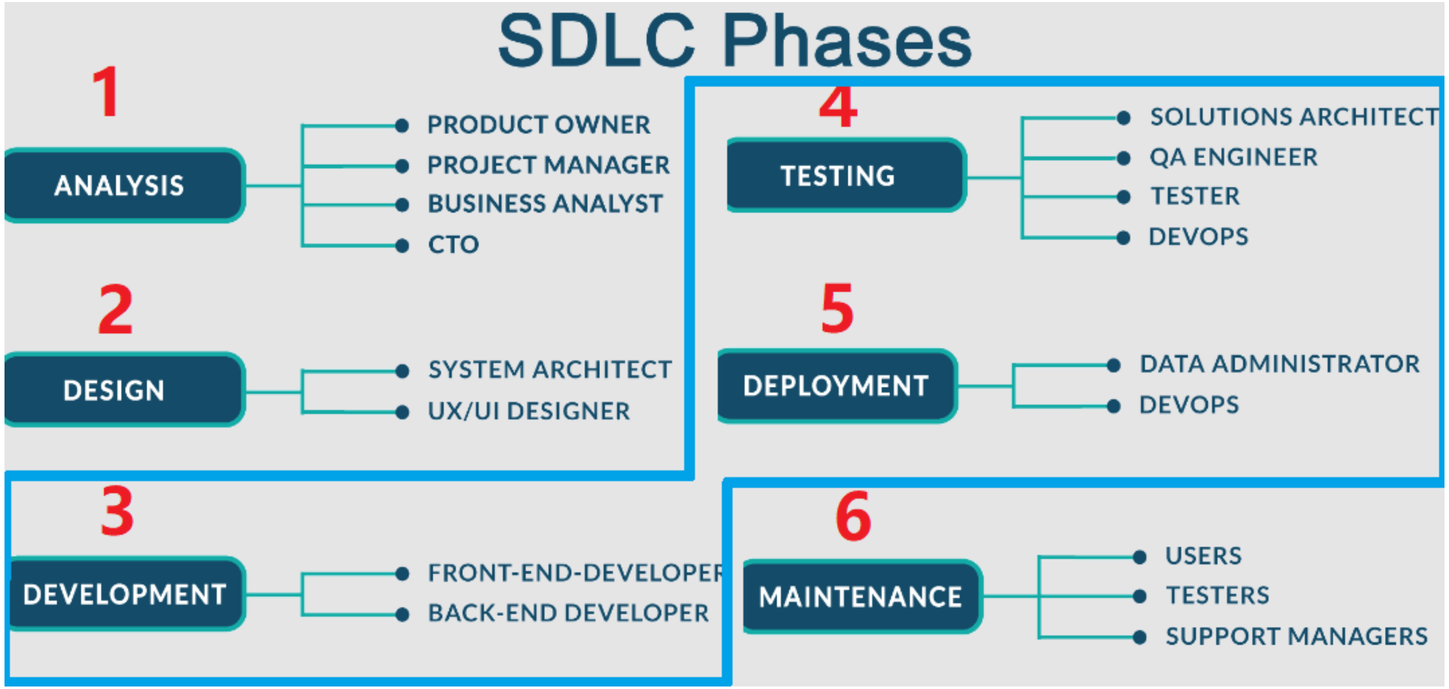
CI/CD

Developing In A Team Environment
Elimination Of Manual Errors

Where is CI/CD located in SDLC?



Let's Recall Agile Principles

- #1 - Customer satisfaction by early and continuous delivery of valuable software
- #2 - Welcome changing requirements, even late in development
- #3 - Working software is delivered frequently
- #7 - Working software is the primary measure of progress
- #8 - Sustainable development, able to maintain a constant pace

What are we supposed to do?

- Write code? Sort of...
- Deliver high quality / working software? **Yes**
- Most likely the code you write is the foundation of your company’s value:
  - Writing bad code means:
    - Your company is worth less
    - Your company can’t add / change quickly to react to business needs
    - You endanger your own position as outsourcing / 3rd party solutions become more attractive
- Given the above should you be:
  - Careful / diligent / thorough
  - Reckless / unplanned / cursory

Solo work is NICE

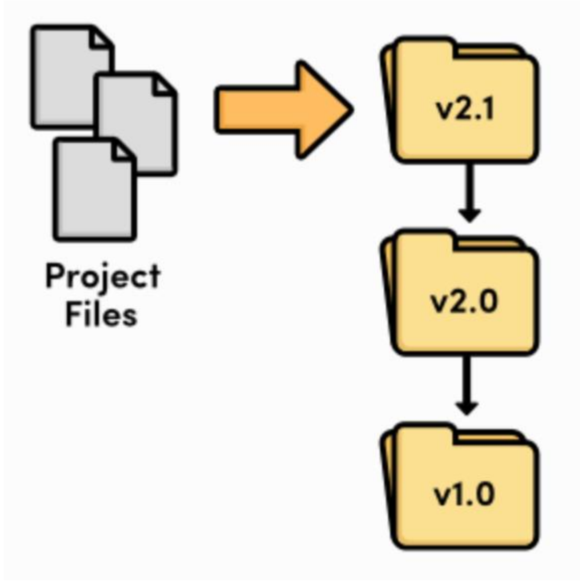
- Wild west!
- If it works, you’re good!
- Easily take any shortcut you want, as long as you know how it works that’s all that matters
- Tailor everything to your preference / convenience:
    - IDE settings
    - Configuration methods / values / namings
- You can adopt any methodology you want, you are the arbiter of the level of quality you deliver to your customer
- FAST

But, when working in a Team Environment...

- Everyone has their own preferences:
  - Naming conventions
  - Indent conventions (spaces / tabs : the answer is tabs)
  - Project layout
  - Operating systems might vary
  - IDE / editors (e.g. line endings)
- Multiple people working on the **same** component / source code
- Multiple contributors to the same code base
- The **mediocre programmer** problem:
  - Bad (not smart), just filling a slot
  - Lazy (maybe smart), takes any and every shortcut no matter the cost to others / organization
  - Unmotivated (maybe smart), chip on their shoulder / burned out
  - The bigger your company the more you have, you will always have one

Also, Poor Practice

- How we build our apps
- No source control tools Put files on local or shared drive



Poor Tools

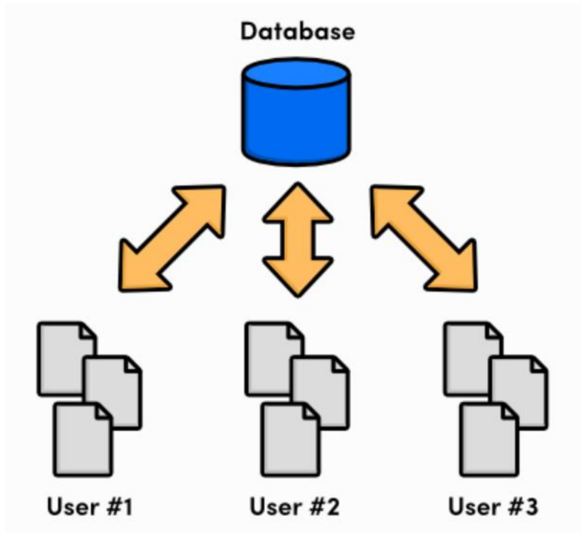
- Pool tools like CVS,SVN

  Centralized architecture, working one file at time

  Limited branching and merging capabilities

  Poor handling of binary files

- Build project manually, Poor build tools like ant
- Deploy the app manually by creating tickets for another team



What are the problems?

- **Increased risk of defects**: there is a higher likelihood of introducing defects into the software. This is because code changes are **not continuously tested and integrated**, and manual testing can be prone to human error.
- **Longer time to market**: Without automated testing and deployment, software releases can take longer to deploy, and manual testing can cause significant delays. This can result in missed business opportunities and can make it difficult to keep up with the competition.
- **Lower quality software**: Manual testing is not as thorough or reliable as automated testing, which can lead to lower quality software. This can result in more bugs and a lower level of customer satisfaction.
- **Higher costs**: Manual testing and deployment processes can be time-consuming and expensive.

Here comes CI/CD

- From Travis CI: “the practice of merging in small code changes frequently - rather than merging in a large change at the end of a development cycle. The goal is to build healthier software by developing and testing in smaller increments.”
- Frequent integration:
  - **IMMEDIATELY** reveals problems when they are small and easier to fix, before going down a long road you can’t easily come back from
  - Reduces merge complexity by merging small changes often vs. massive changes
- Integrating on a developer integration environment lets you work out the problems before it gets to other team members.
  - QA team never blocked by broken builds
  - Branches always compile and (if unit tests are comprehensive) always work. Merges are blocked if code fails to compile or unit tests fail.

Tools you may need to use

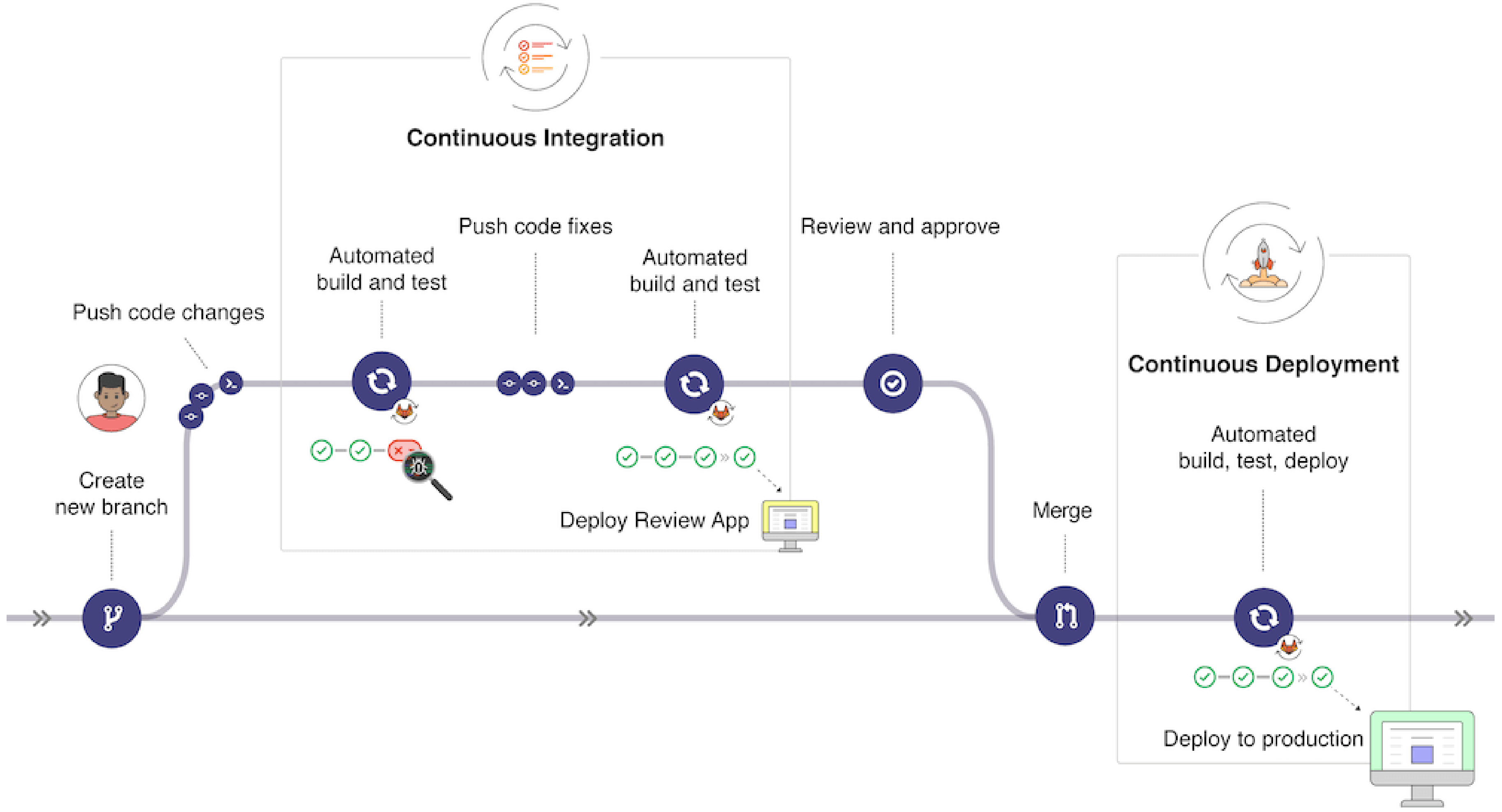
- **Jenkins**: An open-source, self-hosted automation server that can be used for CI/CD. Jenkins supports a wide range of plugins and integrations, making it highly customizable and adaptable to different workflows.
- **GitLab CI/CD**: A built-in CI/CD service offered by GitLab, a web-based repository management platform. GitLab CI/CD is tightly integrated with GitLab's repository management features, providing an all-in-one solution for source code management, CI/CD, and deployment.
- **Travis CI**: A cloud-based CI/CD service that offers seamless integration with GitHub repositories. Travis CI provides a simple YAML-based configuration and supports multiple languages and platforms.

GitLab CI/CD workflow

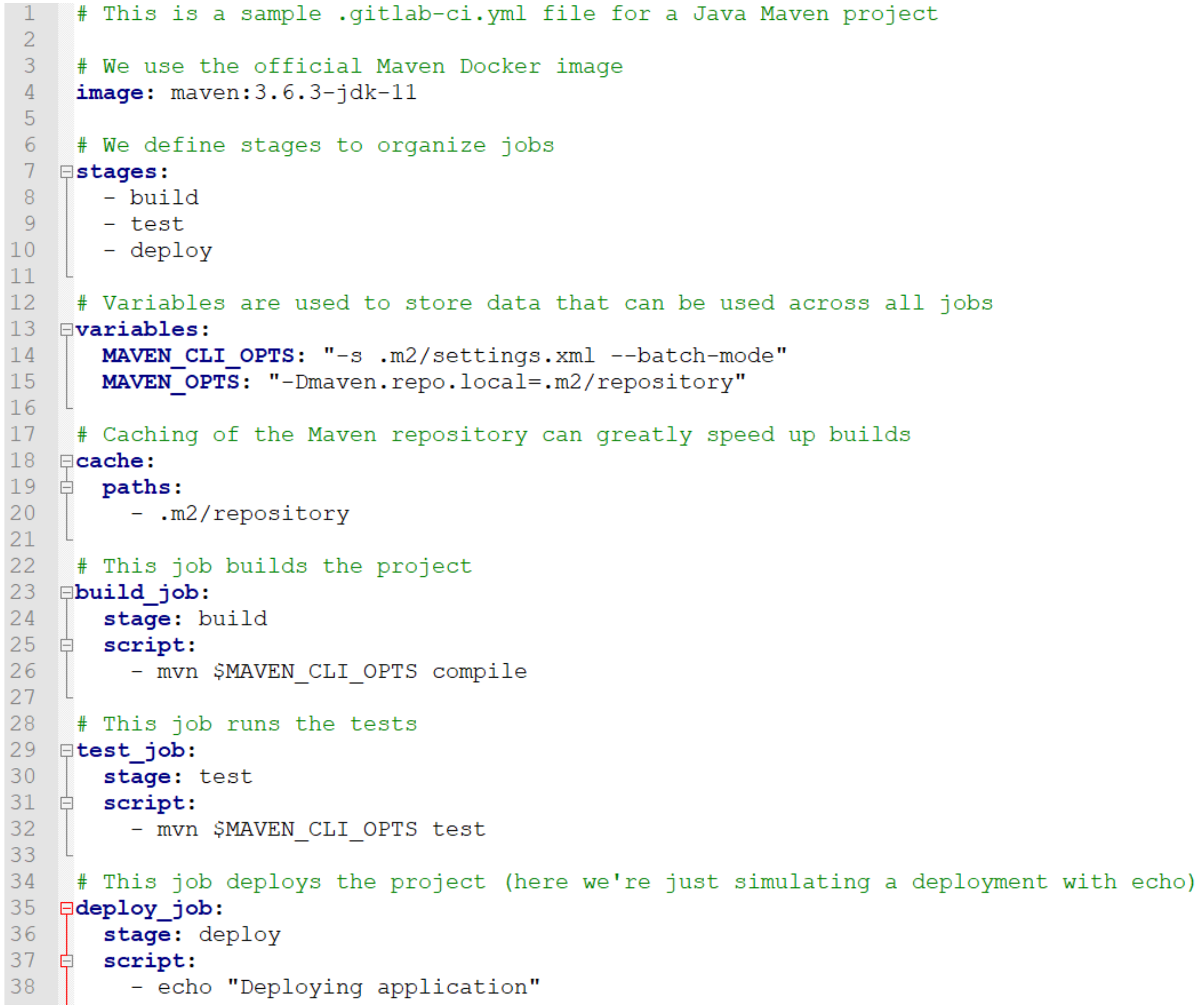
- **Discussion**: Start by discussing proposed changes in an issue.
- **Local Work**: Make and test changes in your local development environment.
- **Push Changes**: Commit changes to a feature branch and push to the GitLab repository.
- **Trigger CI/CD**: Push action triggers the CI/CD pipeline in GitLab defined in your **.gitlab-ci.yml** file.

GitLab CI/CD workflow

- **Run Scripts**: The pipeline automatically builds, tests, and deploys a Review App.
- **Code Review**: Submit your changes for review and approval.
- **Merge Changes**: Once approved, merge the feature branch into the default branch.(Master main)
- **Automatic Deployment**: GitLab CI/CD deploys the changes to the production environment.


Continuous Integration
Push code fixes
Review and approve
Automated build and test
Automated build and test
Push code changes
Continuous Deployment
Automated build, test, deploy
Create new branch
Deploy Review App
Merge
Deploy to production

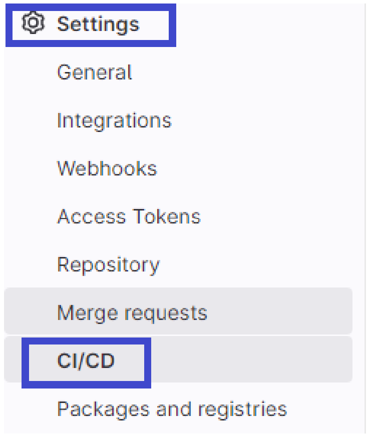
.gitlab-ci.yml

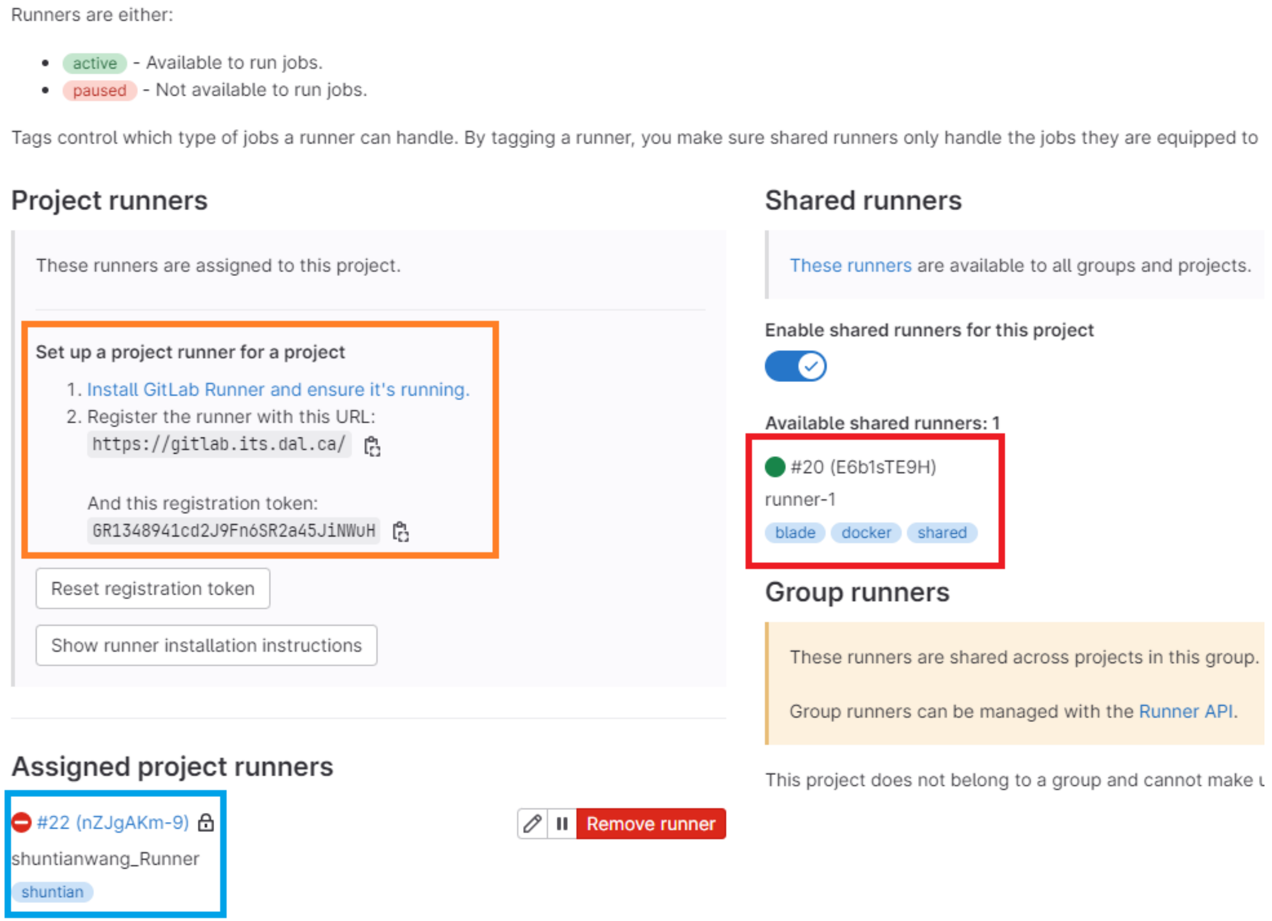


```yaml
# This is a sample .gitlab-ci.yml file for a Java Maven project

# We use the official Maven Docker image
image: maven:3.6.3-jdk-11

# We define stages to organize jobs
stages:
  - build
  - test
  - deploy

# Variables are used to store data that can be used across all jobs
variables:
  MAVEN_CLI_OPTS: "-s .m2/settings.xml --batch-mode"
  MAVEN_OPTS: "-Dmaven.repo.local=.m2/repository"

# Caching of the Maven repository can greatly speed up builds
cache:
  paths:
    - .m2/repository

# This job builds the project
build_job:
  stage: build
  script:
    - mvn $MAVEN_CLI_OPTS compile

# This job runs the tests
test_job:
  stage: test
  script:
    - mvn $MAVEN_CLI_OPTS test

# This job deploys the project (here we're just simulating a deployment with echo)
deploy_job:
  stage: deploy
  script:
    - echo "Deploying application"
```

Runner

- GitLab Runner is an application that works with GitLab CI/CD to run jobs in a pipeline.
- you can run your CI/CD jobs on runners hosted by GitLab
- OR, you can install GitLab Runner and register your own runners on GitLab.com or on your own instance.




Runners are either:

- active - Available to run jobs.
- paused - Not available to run jobs.

Tags control which type of jobs a runner can handle. By tagging a runner, you make sure shared runners only handle the jobs they are equipped to

Project runners

These runners are assigned to this project.

Set up a project runner for a project

1. Install GitLab Runner and ensure it's running.
2. Register the runner with this URL:
   `https://gitlab.its.dal.ca/`

   And this registration token:
   `GR1348941cd2J9Fn6SR2a45JiNWuH`

Reset registration token

Show runner installation instructions

Assigned project runners

#22 (nZJgAKm-9)
shuntianwang_Runner
shuntian

Remove runner

Shared runners

These runners are available to all groups and projects.

Enable shared runners for this project

Available shared runners: 1

#20 (E6b1sTE9H)
runner-1
blade docker shared

Group runners

These runners are shared across projects in this group.

Group runners can be managed with the Runner API.

This project does not belong to a group and cannot make u

Executer

- When you register a runner, you must choose an executor.
- An executor determines the environment each job runs in.

Best Practices of CI/CD

- **Automate as much as possible**: Automation is a key aspect of CI/CD. Automate your build, testing, and deployment processes to reduce errors, improve consistency, and save time.
- **Test early and often**: Running automated tests frequently and as early in the development cycle as possible can help to catch errors before they become more difficult and expensive to fix.
- **Maintain a single source code repository**: Keeping all code in one repository makes it easier to manage and reduces the chances of code conflicts and inconsistencies.
- **Implement Continuous Integration**: Integrate code changes into a shared repository regularly and automatically run tests to catch errors early.
- **Use version control**: Keep track of changes and maintain a clear history of your code with version control tools like Git.

Best Practices of CI/CD(Cont.)

- **Continuous Deployment**: Automate the deployment process to reduce the time between the completion of development and the release of the software to end-users.
- **Monitor and measure performance**: Use metrics and monitoring tools to identify issues and track improvements over time.
- **Use environment parity**: Ensure that the environment in which code is built, tested, and deployed is as similar as possible to the production environment.
- **Collaborate and communicate**: Ensure that all members of the development team are aware of the CI/CD process, their roles, and responsibilities, and collaborate closely to ensure a smooth and efficient development process.

Benefits of CI/CD

- **Improved code quality**: With frequent integration and automated testing, CI/CD helps catch bugs and issues earlier in the development process, resulting in higher quality code.
- **Easier identification and fixing of bugs**: Frequent integration and automated testing make it easier to identify and resolve issues, leading to a more stable and reliable application.
- **Increased team productivity**: By automating routine tasks such as builds, testing, and deployment, CI/CD frees up developers to focus on writing code and adding new features.

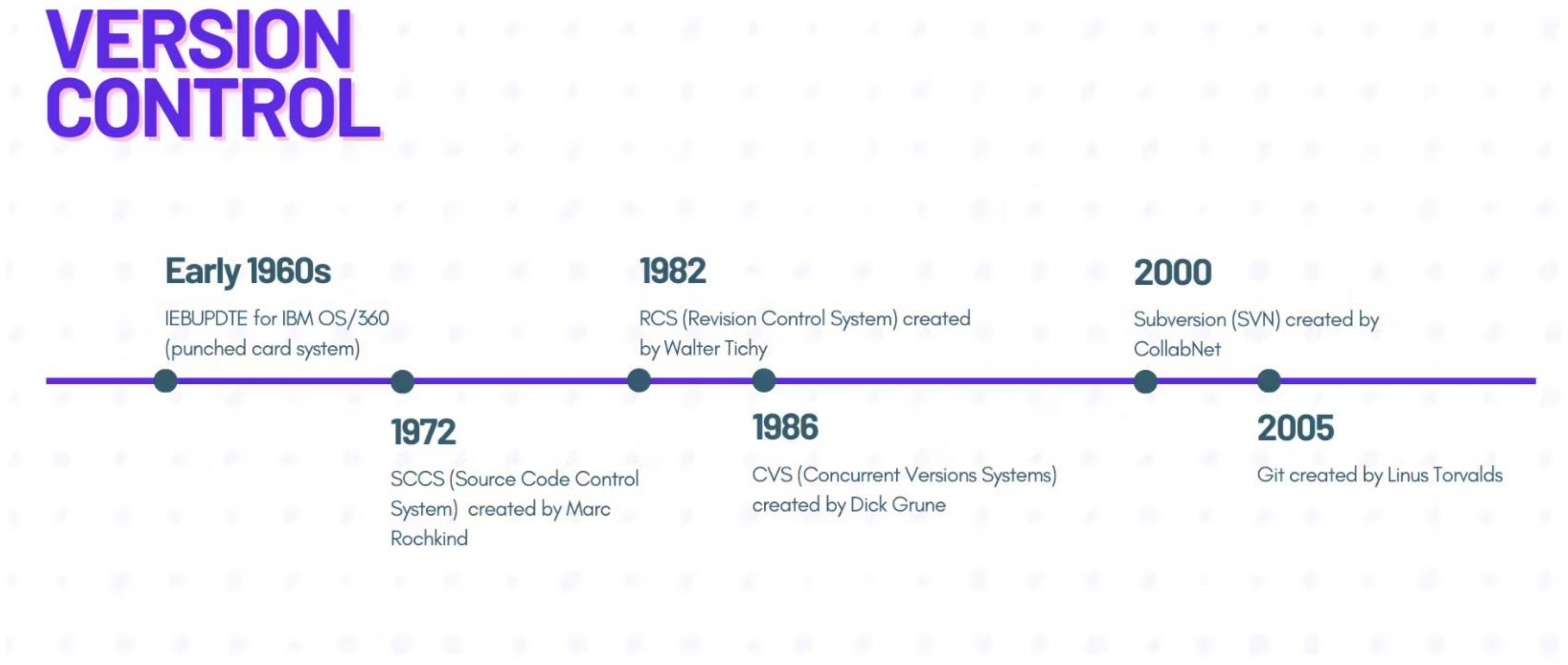
Benefits of CI/CD(Cont.)

- **Faster feedback loops**: CI/CD provides developers with rapid feedback on their changes, allowing them to iterate quickly and make improvements more efficiently.
- **Enhanced collaboration and communication**: By integrating code changes frequently, CI/CD encourages better collaboration and communication among team members, reducing the chances of conflicts and misunderstandings.
- **Hit dates more reliably:** Breaking work into smaller, manageable bites means it's easier to complete each stage on time and track progress

Tools you may need to use(cont.)

- **CircleCI**: A cloud-based CI/CD platform that supports integration with GitHub and Bitbucket. CircleCI offers a flexible configuration with support for parallelism and caching, resulting in faster build times.
- **Bamboo**: A CI/CD server developed by Atlassian, the company behind Jira and Bitbucket. Bamboo integrates well with other Atlassian tools and provides built-in support for deployment projects and release management.
- **TeamCity**: A CI/CD server developed by JetBrains, the company behind IntelliJ IDEA and other popular development tools. TeamCity offers a wide range of features, including build configuration templates, real-time build monitoring, and integration with various version control systems, issue trackers, and other tools.

A little bit of history:


VERSION CONTROL
Early 1960s
IEBUPDTE for IBM OS/360 (punched card system)
1972
SCCS (Source Code Control System) created by Marc Rochkind
1982
RCS (Revision Control System) created by Walter Tichy
1986
CVS (Concurrent Versions Systems) created by Dick Grune
2000
Subversion (SVN) created by CollabNet
2005
Git created by Linus Torvalds

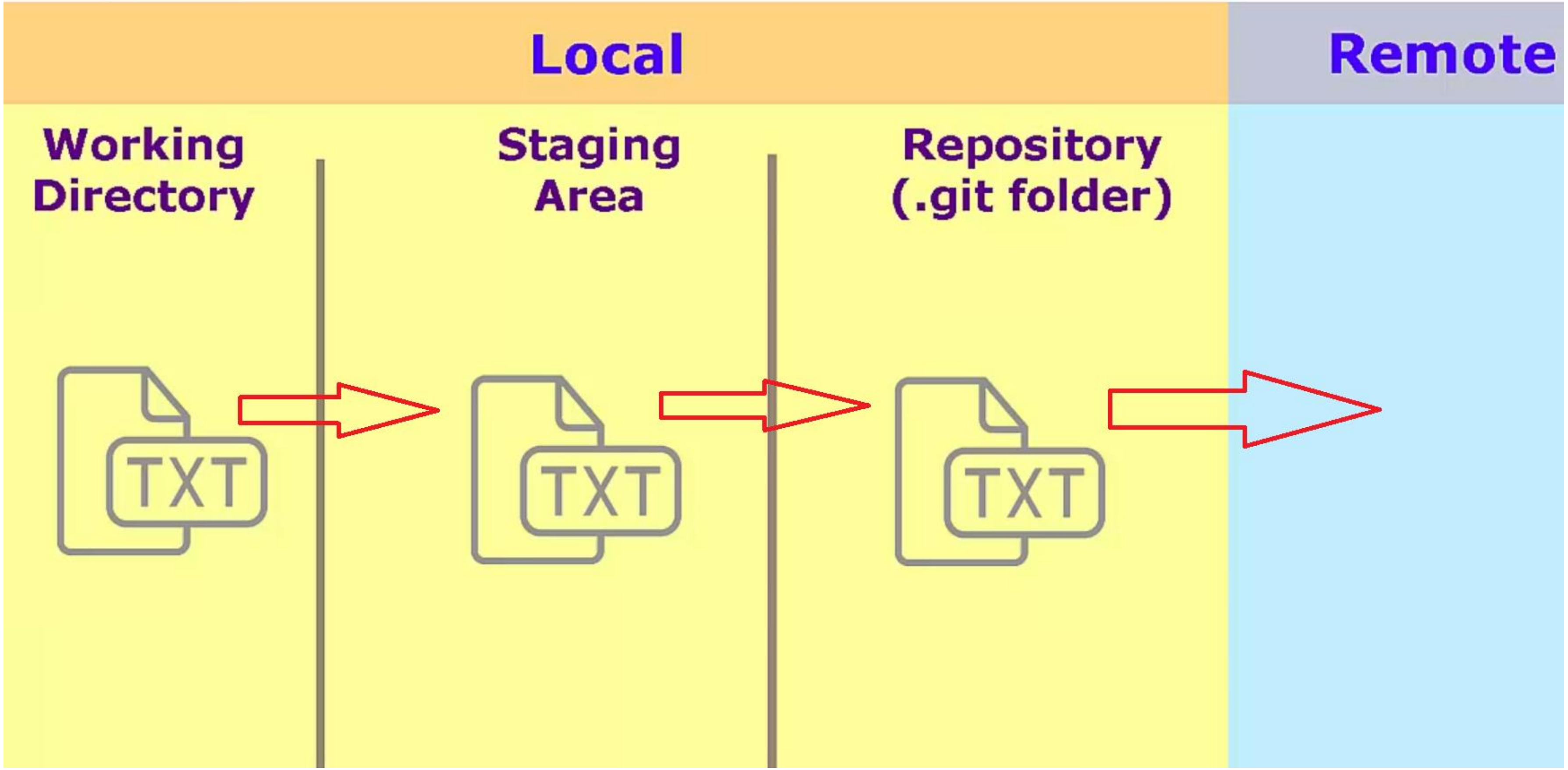
Git

Git is a widely used distributed version control system designed to handle everything from small to large projects with speed and efficiency. It was created by **Linus Torvalds**, the creator of the Linux operating system, in 2005. Git enables developers to track changes in their code, collaborate with others, and revert to previous versions when needed.

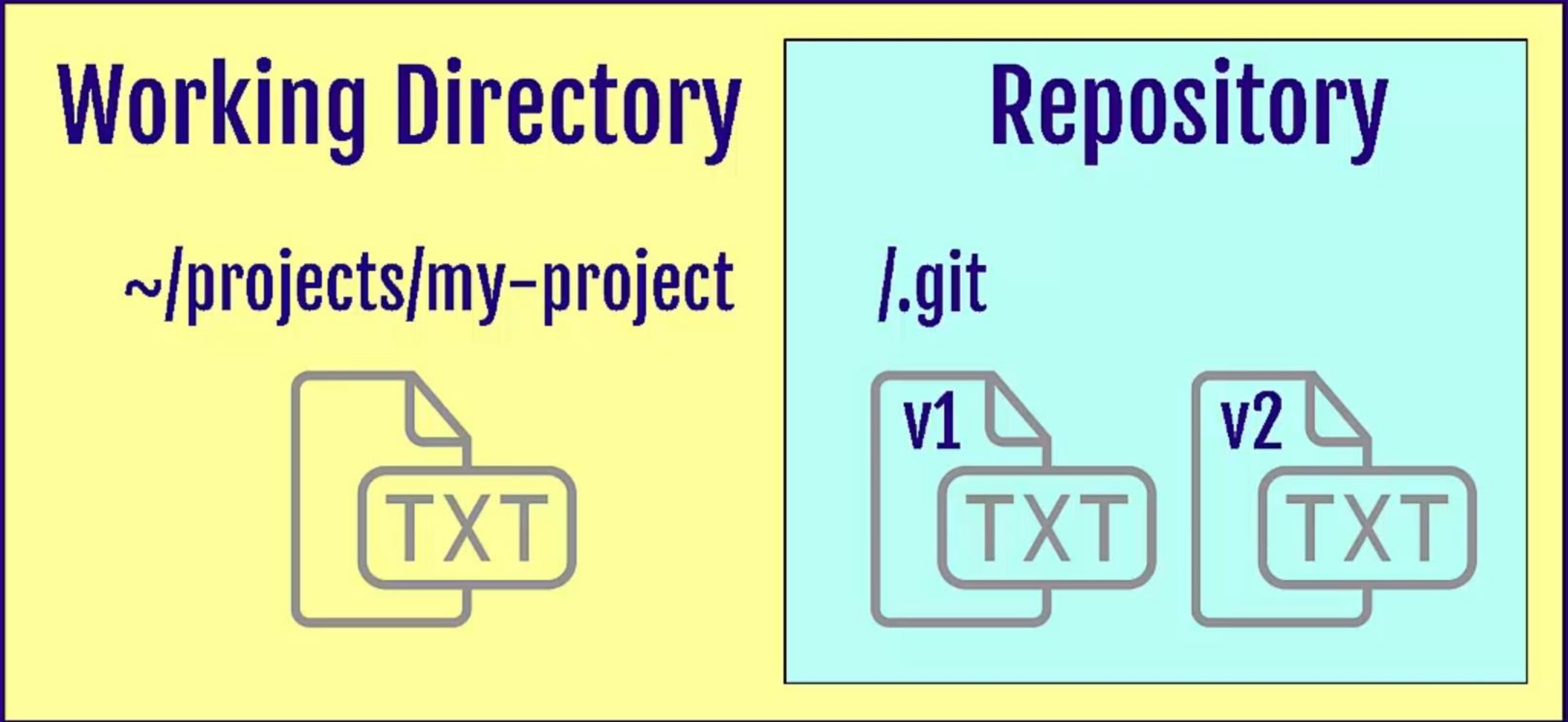
What Git improved

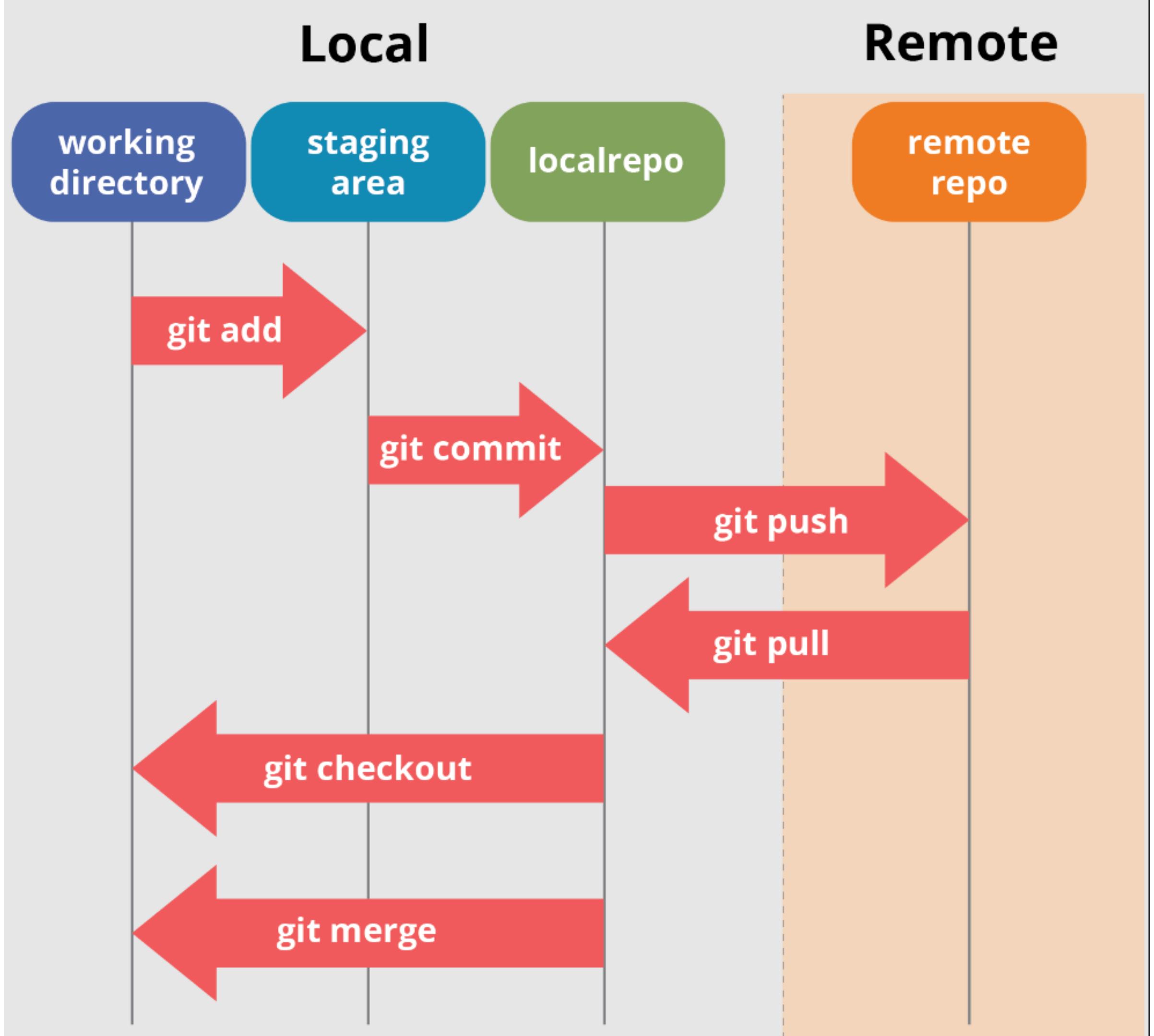
- **Centralized vs. Distributed**: SVN and CVS are centralized version control systems, while Git is a distributed version control system.
- **Performance**: Git generally has better performance compared to SVN and CVS, mainly because most operations in Git are performed locally.
- **Branching and Merging**: Git has superior support for branching and merging compared to SVN and CVS. Branching in Git is fast, lightweight, and an integral part of the development workflow.
- **Storage Model**: Git stores data as snapshots of the entire project at each commit, whereas SVN and CVS store data as a series of deltas (differences) between successive revisions.

Basic Git Workflow Life Cycle



The Git Repository




Local
Remote
working directory
staging area
localrepo
remote repo
git add
git commit
git push
git pull
git checkout
git merge

Recall questions

- Why we need CI/CD?
- What are steps of the Gitlab CI/CD flow?
- What is a runner in Gitlab?
- What is an executer in Gitlab?
- What does .gitLab-ci.yml file do?
- What git command triggers CI/CID pipeline to start?
- What are the significant improvements when transitioning from CVS and SVN to Git?
- What are the four states when using GIT?
- Which ones are local and which one is remote?
- Which directory store the tracking history of the files?

Mostly used Git Commands

- **`git clone <repository_url>`**: Clones an existing repository from a remote source. (Remote repository -> Local Repository)
- **`git init`**: Initializes a new, empty Git repository. (Local working directory)
- **`git add <file>`**: Adds a specific file to the staging area. (Local working directory - > Local staging area)
- **`git commit -m "Commit message"`**: saves the changes in your local repository with a descriptive message. (Local stage Staging area -> local repository)
- **`git push`**: Pushes your commits to the remote repository.(Local Repository ->Remote Repository)

Mostly used Git Commands(Cont.)

- **`git branch <branch_name>`**: Creates a new branch with the given name. **??**
- **`git checkout <branch_name>`**: Switches to the specified branch. (Local working directory -> Local working directory)
- **`git merge <branch_name>`**: Merges the specified branch into the current branch.
- **`git fetch`**: Retrieves updates from the remote repository but does not merge them. (Remote repository -> Local Repository)
- **`git pull`**: Fetches updates from the remote repository and merges them into the current branch. (Remote repository -> Local Repository)

Source Control is not enough!
Git Workflow - GitFlow

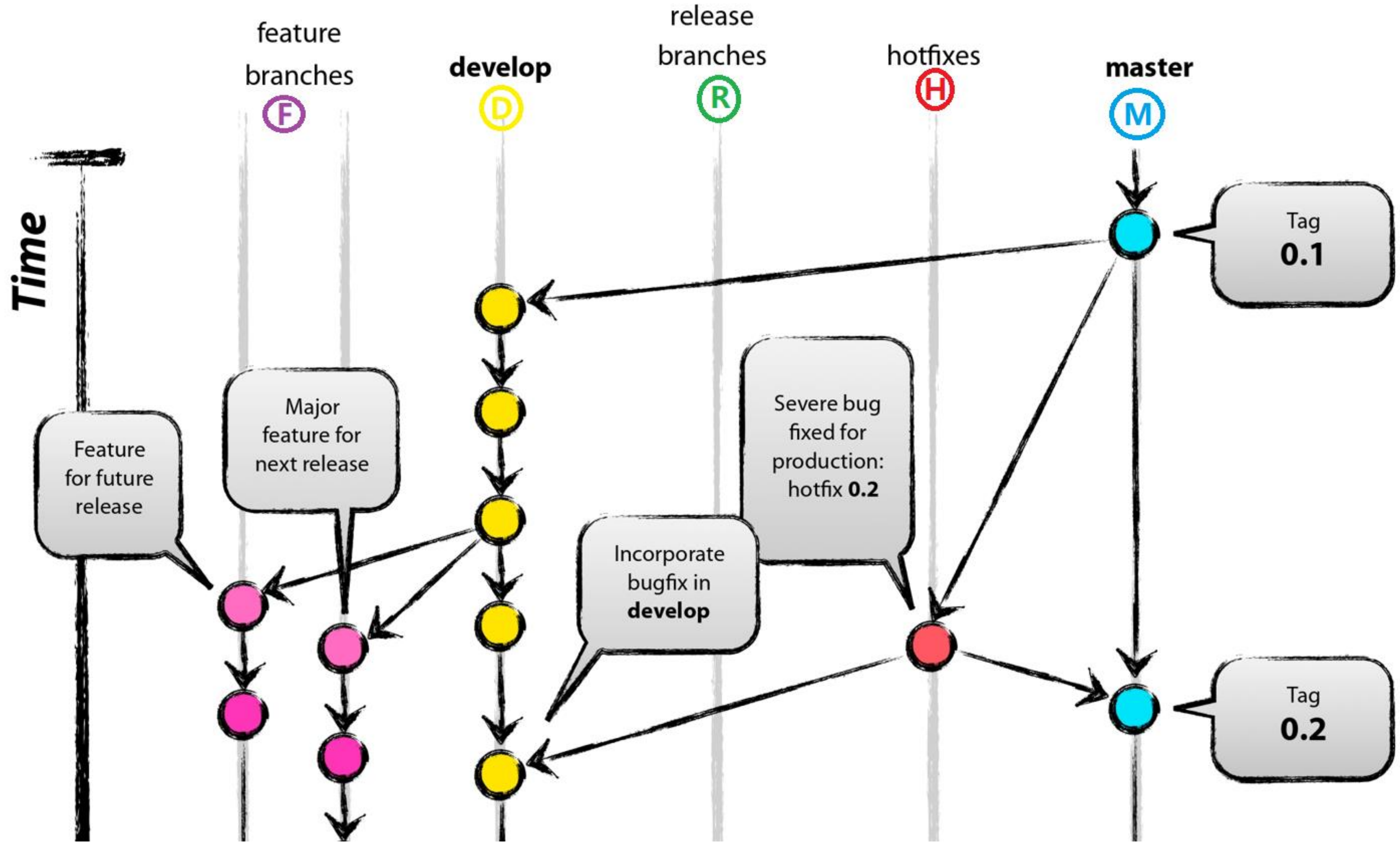
GitFlow is a branching model for Git introduced by Vincent Driessen in a 2010 blog post. It provides a structured and organized approach to managing code in a Git repository, with a focus on parallel development, release management, and support for multiple versions of a project.

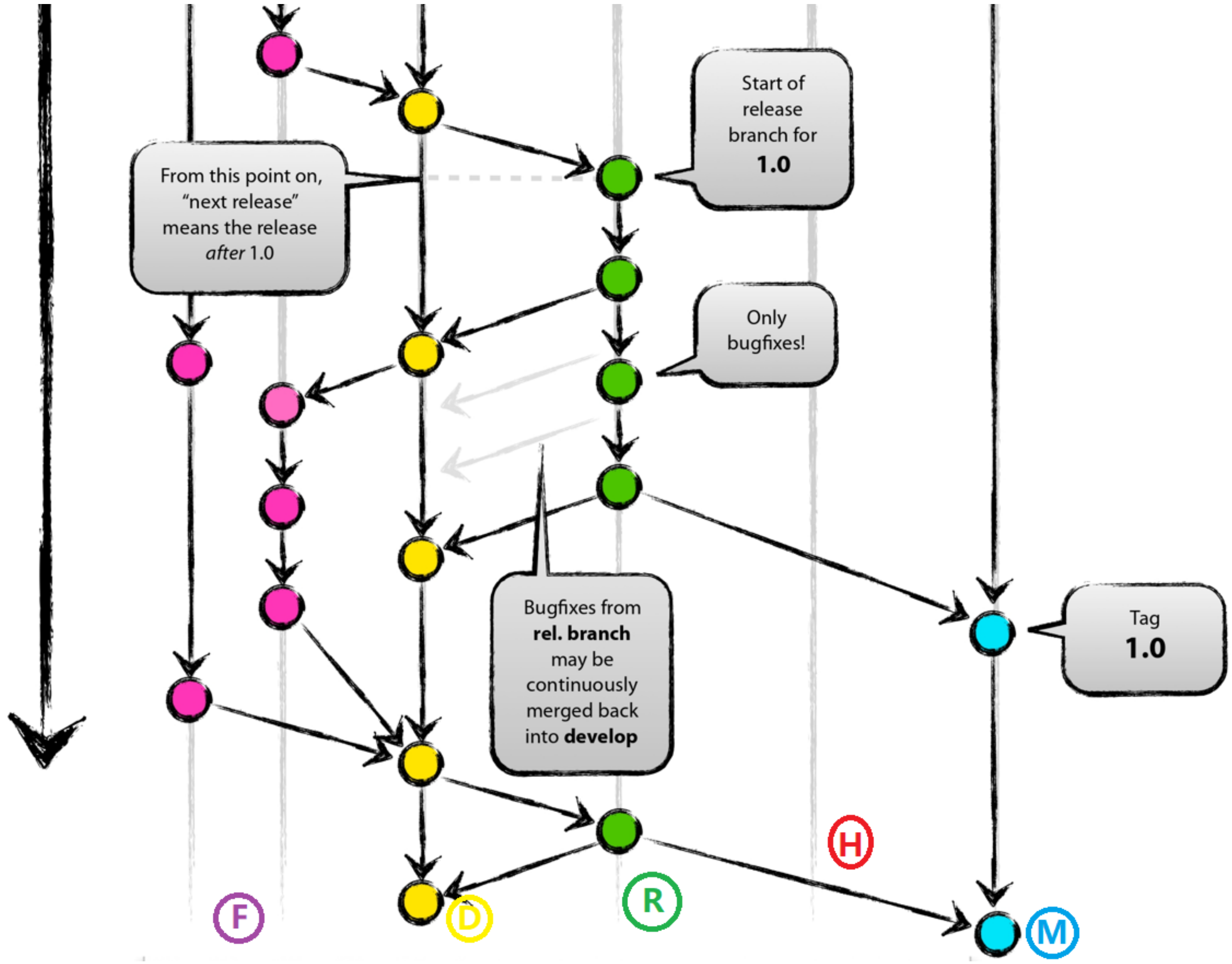
GitFlow

- Main Branches:
  - **master** (or **main**): Represents the production-ready state of the project. Only stable, tested, and approved changes are merged into this branch.
  - **develop**: Serves as an integration branch for features and is where developers merge their completed feature branches. This branch contains the latest development changes.

GitFlow (Cont.)

- Supporting Branches:
  - **feature** branches: Created from the **develop** branch to develop new features or enhancements. Once completed, they are merged back into the **develop** branch through a pull request.
  - **release** branches: Created from the **develop** branch when the project is close to a new production release. These branches allow for final testing, bug fixing, and adjustments before merging the changes into the **master** (or **main**) branch for deployment. Once merged, the changes are also merged back into the **develop** branch.
  - **hotfix** branches: Created from the **master** (or **main**) branch to quickly address critical issues or bugs found in the production version. Once the fixes are complete, the changes are merged back into both the **master** (or **main**) branch and the **develop** branch.


feature branches
F
develop
D
release branches
R
hotfixes
H
master
M
Time
Tag 0.1
Severe bug fixed for production: hotfix 0.2
Major feature for next release
Feature for future release
Incorporate bugfix in develop
Tag 0.2


Start of release branch for 1.0
From this point on, "next release" means the release *after* 1.0
Only bugfixes!
Bugfixes from **rel. branch** may be continuously merged back into **develop**
Tag 1.0
F
D
R
H
M

GitFlow

Advantages:

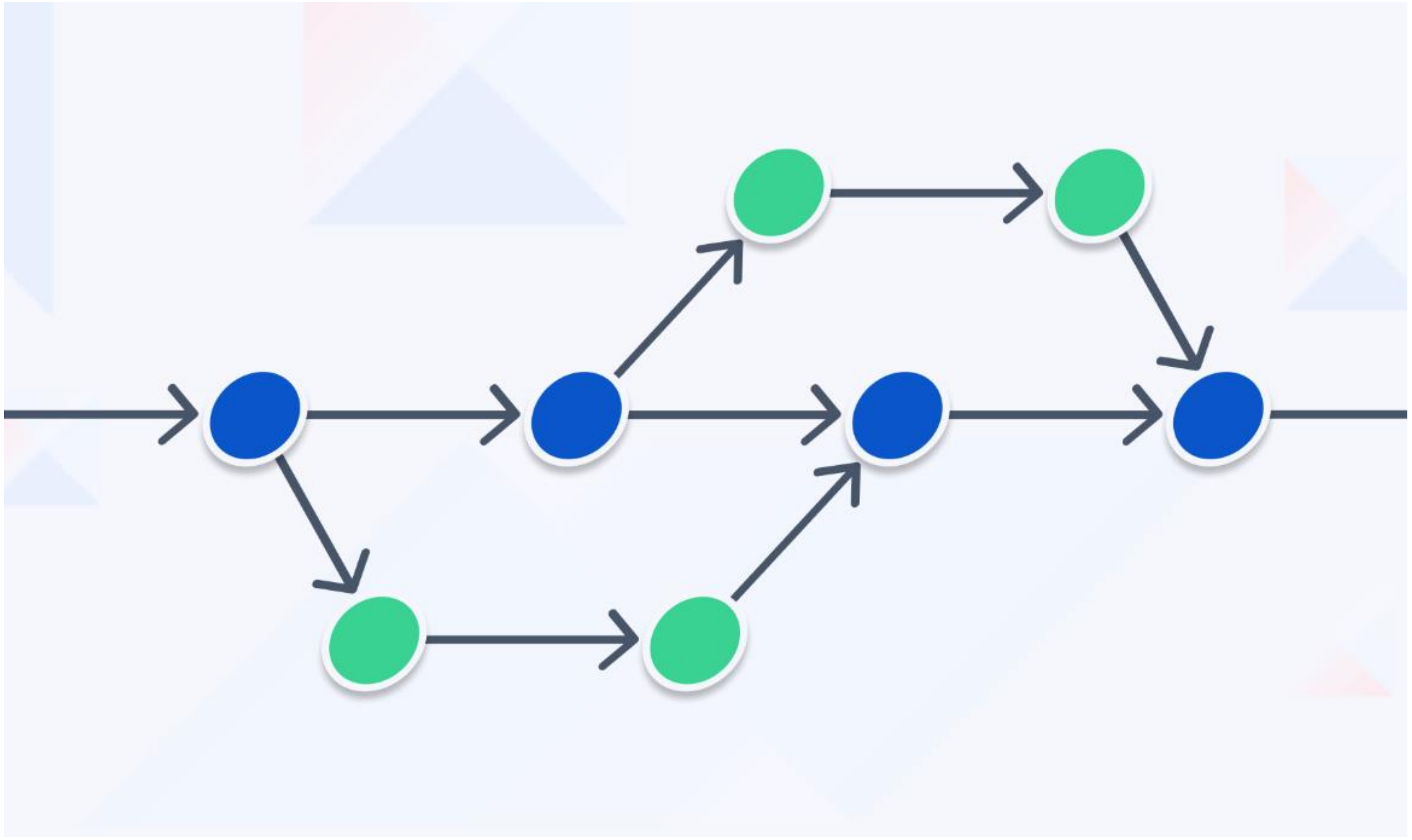
- Clear separation of branches for development, release, hotfix, and feature work.
- Supports parallel development of multiple features or bug fixes.
- Allows for simultaneous maintenance of multiple releases.
- Well-suited for projects with scheduled releases or a need for long-term support.

Disadvantages:

- More complex and harder to learn, especially for newcomers to Git.
- Requires more branch management and overhead, as it involves many branches and merges.
- May be overkill for smaller projects or teams with a simple development workflow.

Trunk Based workflow

Trunk-based development is a software development approach that emphasizes simplicity, frequent integration, and minimizing long-lived branches. **The core idea is that all developers work directly on the main branch (or trunk) of the repository,** integrating their changes frequently to keep the codebase stable, maintainable, and always in a releasable state.



Trunk Based Development

Advantages:

- **Simplified branch management**: Trunk-based development eliminates the need for managing multiple feature or release branches, making it easier for developers to collaborate and reduce the complexity of merging code.
- **Continuous integration**: Developers merge their code changes frequently, leading to faster feedback on potential issues, easier bug identification, and quicker resolution. This encourages a culture of continuous integration and deployment.
- **Faster release cycles**: With trunk-based development, teams can deliver new features and bug fixes more rapidly, as they avoid the overhead associated with merging multiple long-lived branches.

Trunk Based Development (Cont.)

- **Improved code quality**: Frequent code integration and testing help identify and fix issues early, leading to a higher-quality codebase.
- **Better collaboration**: Working on a single branch encourages communication and collaboration between team members, as they have to coordinate and share their changes regularly.

Trunk Based Development (Cont.)

Disadvantages:

- **Merge conflicts**: Frequent merging can lead to conflicts that need to be resolved, which may cause delays and introduce the risk of human error.
- **Limited isolation**: Working on a single branch can make it challenging to isolate and test individual features, especially when dealing with large or complex changes.
- **Challenging to revert changes**: If a feature or change needs to be reverted, it can be more difficult to identify and remove the specific changes in a trunk-based workflow.

Trunk Based Development (Cont.)

- **Requires strong development practices**: Trunk-based development requires disciplined development practices, such as thorough code reviews, testing, and automation. Teams with less mature practices may struggle with the rapid pace of change.
- **Scalability concerns**: In larger teams, trunk-based development may become difficult to manage due to the increased number of developers working on the same branch, leading to more frequent merge conflicts and integration issues.

Implementation of CD (Build and Deploy)

Ant

What is the major problem ? Let's check out a simple ant build.xml file

<APACHE ANT>

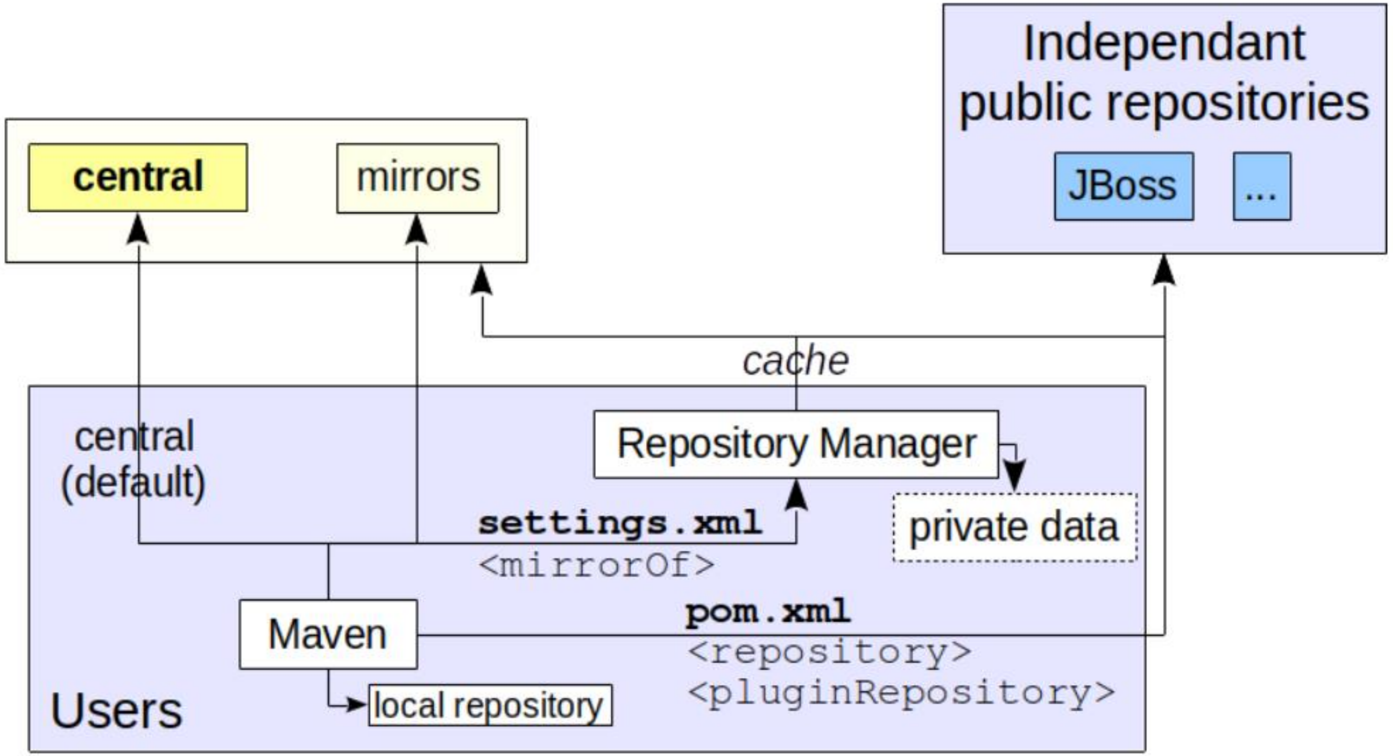
Build.xml

```
<?xml version="1.0"?>
<project name="SimpleJavaProject" default="run" basedir=".">
    <property name="src.dir" value="src"/>
    <property name="build.dir" value="build"/>
    <property name="classes.dir" value="${build.dir}/classes"/>
    <property name="lib.dir" value="lib"/>

    <path id="classpath">
        <fileset dir="${lib.dir}" includes="*.jar"/>
    </path>

    <target name="compile">
        <mkdir dir="${classes.dir}"/>
        <javac srcdir="${src.dir}" destdir="${classes.dir}" classpathref="classpath"/>
    </target>

    <target name="run" depends="compile">
        <java classname="com.example.Main" fork="true">
            <classpath>
                <path refid="classpath"/>
                <pathelement location="${classes.dir}"/>
            </classpath>
        </java>
    </target>
</project>
```

Maven

Repository System and Dependency Management: Maven uses a centralized repository system for storing and sharing artifacts. Also it automatically manages external dependencies, including downloading, versioning, and handling transitive dependencies.

Maven (Repositories)



Maven (Standard Project Structure)

Standard Project Structure: Maven enforces a standard project directory structure, promoting consistency across projects and simplifying the build process.

```
my_project/
  ├─ src/
  │   ├─ main/
  │   │   ├─ java/               # Java source code
  │   │   │   └─ com/
  │   │   │       └─ example/
  │   │   │           └─ MyApp.java
  │   │   ├─ resources/          # Non-code resources (e.g. configuration files, properties, etc.)
  │   │   │   └─ application.properties
  │   │   └─ webapp/             # Web application resources (for web projects)
  │   │       └─ WEB-INF/
  │   │           └─ web.xml
  │   └─ test/
  │       ├─ java/               # Java test source code
  │       │   └─ com/
  │       │       └─ example/
  │       │           └─ MyAppTest.java
  │       └─ resources/          # Test resources (e.g. test configurations, data files, etc.)
  │           └─ test.properties
  │
  ├─ target/                     # Build output directory (generated by Maven)
  ├─ pom.xml                     # Maven project configuration file
  └─ .gitignore                  # Git ignore file (optional, if using Git version control)
```

Maven D.M. (POM.xml heart of Maven)

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Maven D.M. (Transitive Dependencies)

- A transitive dependency is a dependency of a dependency. Maven can automatically resolve and manage transitive dependencies, ensuring that all required libraries are available for your project. It also takes care of potential conflicts, such as different versions of the same library, by following a well-defined dependency mediation strategy.

Maven D.M. (Dependency Scope)

Control the visibility and lifecycle of a dependency within the project to ensure that each dependency is available only where it's needed.

- **compile**: Default scope, used for most dependencies. Required for compilation and included in runtime and test classpaths.
- **provided**: Needed for compilation but expected to be provided by the runtime environment. Included in compile and test classpaths, but not in runtime classpath or final artifact.
- **runtime**: Not needed for compilation but required during runtime. Included in runtime and test classpaths, but not in compile classpath.

Maven D.M. (Dependency Scope Cont.)

Control the visibility and lifecycle of a dependency within the project to ensure that each dependency is available only where it's needed.

- **test**: Required only for testing purposes. Included in the test classpath, but not in compile or runtime classpaths.
- **system**: Similar to provided, but specified as a local system file. Rarely used and not recommended, as it can cause portability and reproducibility issues.
- **import**: Used only with dependency type **pom** in **<dependencyManagement>** section. Allows merging of dependency management information from the imported POM.

Maven D.M. (Dependency Scope Cont.)

```xml
<!-- A runtime dependency -->
<dependency>
    <groupId>com.example</groupId>
    <artifactId>database-driver</artifactId>
    <version>2.0.0</version>
    <scope>runtime</scope>
</dependency>

<!-- A test dependency -->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

Maven D.M. (Dependency Exclusions)

Exclude a specific transitive dependency from your project due to conflicts or other issues

```
<dependencies>
    <dependency>
        <groupId>com.example</groupId>
        <artifactId>library-a</artifactId>
        <version>1.0.0</version>
        <exclusions>
            <exclusion>
                <groupId>com.example</groupId>
                <artifactId>library-b</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

Maven D.M. (Multi-module Projects Management)

Use the **<dependencyManagement>** section of the parent POM to manage the versions and configurations of dependencies for all child modules

Maven D.M. (Multi-module Projects Management Cont.)

Parent / pom.xml

```xml
<project>
    <groupId>com.example</groupId>
    <artifactId>parent-project</artifactId>
    <version>1.0.0</version>
    <packaging>pom</packaging>

    <modules>
        <module>module-a</module>
        <module>module-b</module>
    </modules>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>com.example</groupId>
                <artifactId>shared-library</artifactId>
                <version>2.0.0</version>
            </dependency>
            <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-core</artifactId>
                <version>5.3.10</version>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```

Child / pom.xml

```xml
<project>
    <parent>
        <groupId>com.example</groupId>
        <artifactId>parent-project</artifactId>
        <version>1.0.0</version>
    </parent>

    <artifactId>module-a</artifactId>

    <dependencies>
        <dependency>
            <groupId>com.example</groupId>
            <artifactId>shared-library</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
        </dependency>
    </dependencies>
```

Maven (Build Lifecycle)

Maven defines a set of standard build lifecycle phases that help streamline and standardize the build process. Each phase is responsible for a specific task.

- **validate:** check if all information necessary for the build is available
- **compile:** compile the source code
- **test:** test the compiled source code using a suitable unit testing framework
- **package:** package compiled source code into the distributable format (jar, war, …)
- **verify:** run any checks on results of integration tests to ensure quality criteria are met
- **install:** install the package to a local repository
- **deploy:** copy the package to the remote repository

Maven (Build Lifecycle)

It won't only execute the specified phase, but all the preceding phases as well.

Exmple: mvn deploy
(validate -> compile -> test -> package -> verify -> install -> deploy)

Gradle

Gradle, Maven Comparison (Language and syntax)

- **Maven:** Maven uses an XML-based syntax for its configuration file pom.xml"
- **Gradle:** Gradle uses a Groovy-based DSL (Domain Specific Language) or a Kotlin-based DSL for its configuration files called "build.gradle" or "build.gradle.kts", respectively. These files are more readable and concise compared to Maven's XML files, allowing developers to express complex build logic more easily.

Gradle, Maven Comparison (build.gradle)

```
plugins {
    id 'java'
}

group = 'org.example'
version = '2.0'

repositories {
    mavenCentral()
}

dependencies {
    testImplementation platform('org.junit:junit-bom:5.9.1')
    testImplementation 'org.junit.jupiter:junit-jupiter'
}

test {
    useJUnitPlatform()
}
```

Gradle, Maven Comparison (Performance)

- **Maven:** Maven is known for its slower build times compared to Gradle. It relies on the traditional build lifecycle and is limited by its **linear execution model**.
- **Gradle:** Gradle supports **incremental builds and parallel task execution**. This results in significantly faster build times, especially for large projects.

Gradle, Maven Comparison (Performance)

- **Update Gradle and Java version**
- **Enable parallel execution**

  gradle.properties

  ```
  org.gradle.parallel=true
  ```
- **Re-Enable Gradle Deamon**

  gradle.properties

  ```
  org.gradle.daemon=true
  ```
- **Enable Build Cache**

  gradle.properties

  ```
  org.gradle.caching=true
  ```
- **Increase the heap size**

  gradle.properties

  ```
  org.gradle.jvmargs=-Xmx2048M
  ```

Gradle, Maven Comparison (Flexibility and extensibility)

- **Maven:** Maven follows a strict convention over configuration approach, making it less flexible when it comes to customizing the build process. However, this also means that Maven projects tend to have a more standardized structure, which can be beneficial in large teams or organizations.
- **Gradle:** Gradle offers greater flexibility and extensibility, as it allows developers to define custom build tasks and plugins using Groovy or Kotlin. This makes it a better choice for projects that require complex and tailored build processes.

Gradle, Maven Comparison (Community and ecosystem)

- **Maven:** Maven has been around since 2004, making it a mature and well-established build tool. It has a large user base and a wide range of plugins available, which can help with various tasks and integrations.
- **Gradle:** Gradle was introduced in 2009, and although it is newer than Maven, it has gained significant traction and popularity over the years. Gradle is the default build tool for Android projects, which has also contributed to its growing user base. Gradle's plugin ecosystem is not as extensive as Maven's but continues to grow rapidly.

The Future - Ephemeral/Scalable Solutions

- Serverless & Containers
  - Infinite scaling / configuration / deployment
  - Docker, Heroku, etc...
  - Cloud state machines (e.g. AWS step functions)
- Cloud computing
  - Amazon AWS
  - Azure
  - Google
  - IBM
  - Etc...
- Low code development platforms:
  - Appian
  - Salesforce
  - Etc...

Project Discussion

- Gitflow is required, show us some intermediate releases and the final release
- CI Implementation requirements:
  - You require 3 environments:
    - **devint** - Your local machine, where you integrate changes from other developers and develop features
    - **test** - Where your team integrates each other's features and tests the assembled project
    - **production** - The server that you use to demonstrate to the TAs and instructor
  - Host your test and production environments somewhere:
    - Timberlea (e.g. a Java command line app)
    - Something else? Ask permission. **Do not consume credits from AWS, Azure or Google, you will need these in other courses.**
  - Configure your GitLab project to automatically build, test and deploy your **develop branch to your test** environment
  - Configure your GitLab project to automatically build, test and deploy your **release branch to your production environment**
    - TAs and the instructor will require you to demo to them on production
    - Test your project for bugs on your test environment
  - Local development should connect to your **devint** database
  - The app running on your test environment should connect to your **test** database
  - The app running on your production environment should connect to your **production** database