Large-Scale Software Development

Lecture 3 : Build tool(s) and CI/CD

LINKÖPINGS UNIVERSITET

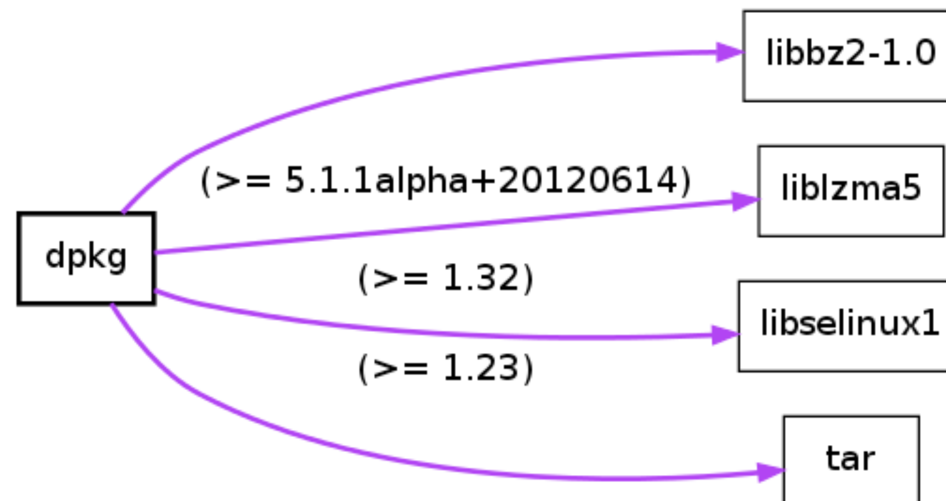
Agenda

- Questions
- Seminar
- Build tools in general
  - Maven in particular
- CI/CD in general
  - Gitlab-ci in particular

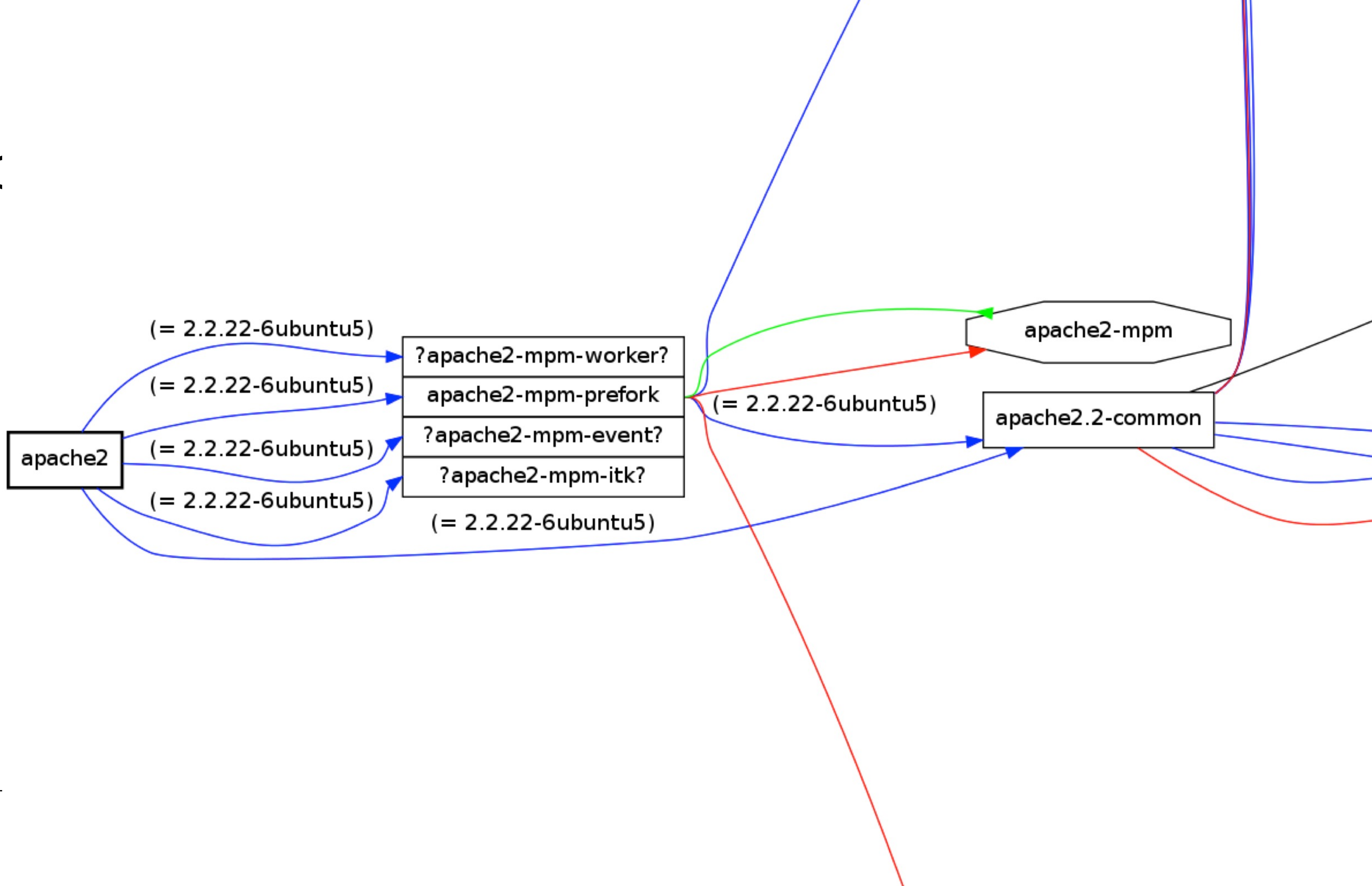
Scenarios - actions

1. Many groups of developers, multiple changes — Package management system
2. Building artefacts based on multiple files with dependencies — build scripts
3. Conducting multiple actions with inter-dependencies on multiple files … — Flexible build system

Package management systems



Dep



Dependency management issues

- Is a request to modify the current software component graph satisfiable?
  - Are additions compatible with other components?
  - Are deletions safe with respect to other dependencies?
- Given a component, determine versions of other components we can safely rely on

Dependency management issues

- Y depends on X >= 1.8. X makes binary incompatible changes from v. 1.9 to v. 2.0…
- Can components be installed from local sources as well as from remote?
- Should OS-specific dependency management or language-specific be used?

Software package management systems

Name	Environment	Format
NuGet	.Net CLR	XML
Gradle	JVM	XML
dpkg/APT	Linux	Ar archive
Rubygems	Ruby	Ruby
MSI	Windows	In-file DB
BSD Ports	OS X/Linux/BSD	Makefile
…		

Maven

Maven is a project management tool which encompasses a project object model, a set of standards, a project lifecycle, a dependency management system, and logic for executing plugin goals at defined phases in a lifecycle. When you use Maven, you describe your project using a well-defined project object model, Maven can then apply cross-cutting logic from a set of shared (or custom) plugins

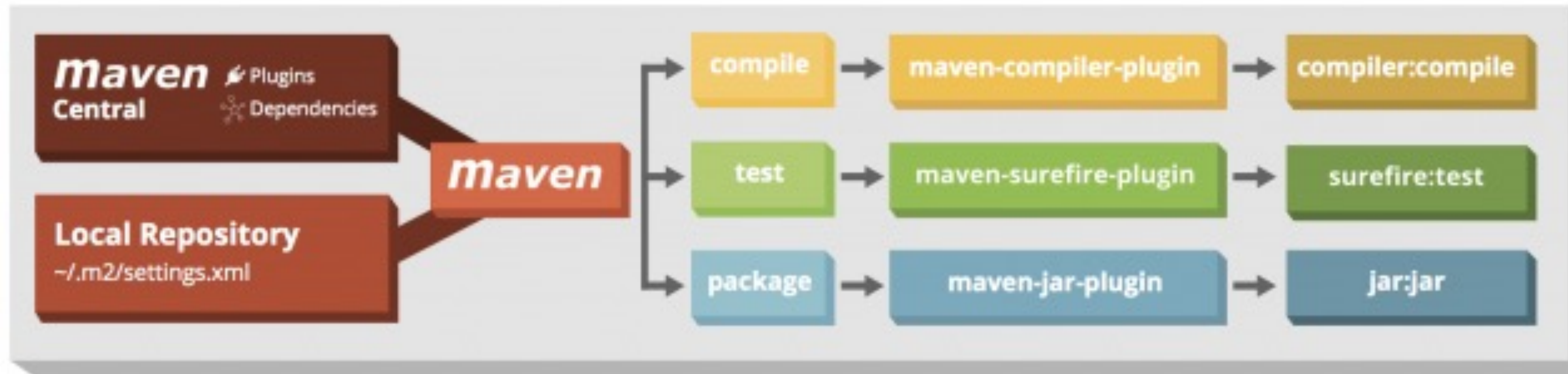
Maven - Convention Over Configuration

```
${basedir}/src/main/java

${basedir}/src/main/resources

${basedir}/src/test
```

Lifecycle , Phases and plugins
mvn clean compiler:compile package

- Three built-in Lifecycles
  - default, clean and site
- Phases in a lifecycle
  - validate, compile, test, package, verify, install, deploy
  - pre-*, post-*, or process-*
    - are not called from the cli (often used in testing)
- Phase are made of Plugin goals
  - compile complier:compile

Maven- Plugins



Maven

```
mvn -h
```

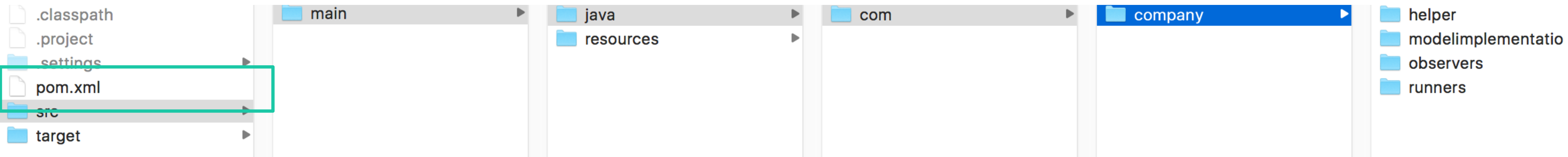
Life cycles

Clean ↑ Default ↑ Site

Default ← validate, compile, test, package, verify, install, deploy

- validate
- compile
- test
- package
- verify
- install
- deploy

Maven – structure




```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -Darchetyp
```

Maven – Configuration

```
<parent>
    <groupId>org.graphwalker.example</groupId>
    <artifactId>graphwalker-example</artifactId>
    <version>3.4.2</version>
</parent>

<artifactId>java-petclinic</artifactId>
```

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.1</version>
            <configuration>
                <source>1.7</source>
                <target>1.7</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.graphwalker</groupId>
            <artifactId>graphwalker-maven-plugin</artifactId>
            <version>${project.version}</version>
            <!-- Bind goals to the default lifecycle -->
            <executions>
                <execution>
                    <id>generate-sources</id>
                    <phase>generate-sources</phase>
                    <goals>
                        <goal>generate-sources</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Demo

Scenarios - actions

1. Single developer, multiple changes — Version control system
2. Many developers, multiple changes — Distributed version control system
3. Many groups of developers, multiple changes — Package management system
4. Building artefacts based on multiple files with dependencies — build scripts
5. Conducting multiple actions with inter-dependencies on multiple files … — Flexible build system
6. Automatically sensing changes and conducting such actions based on changes — Continuous integration tools

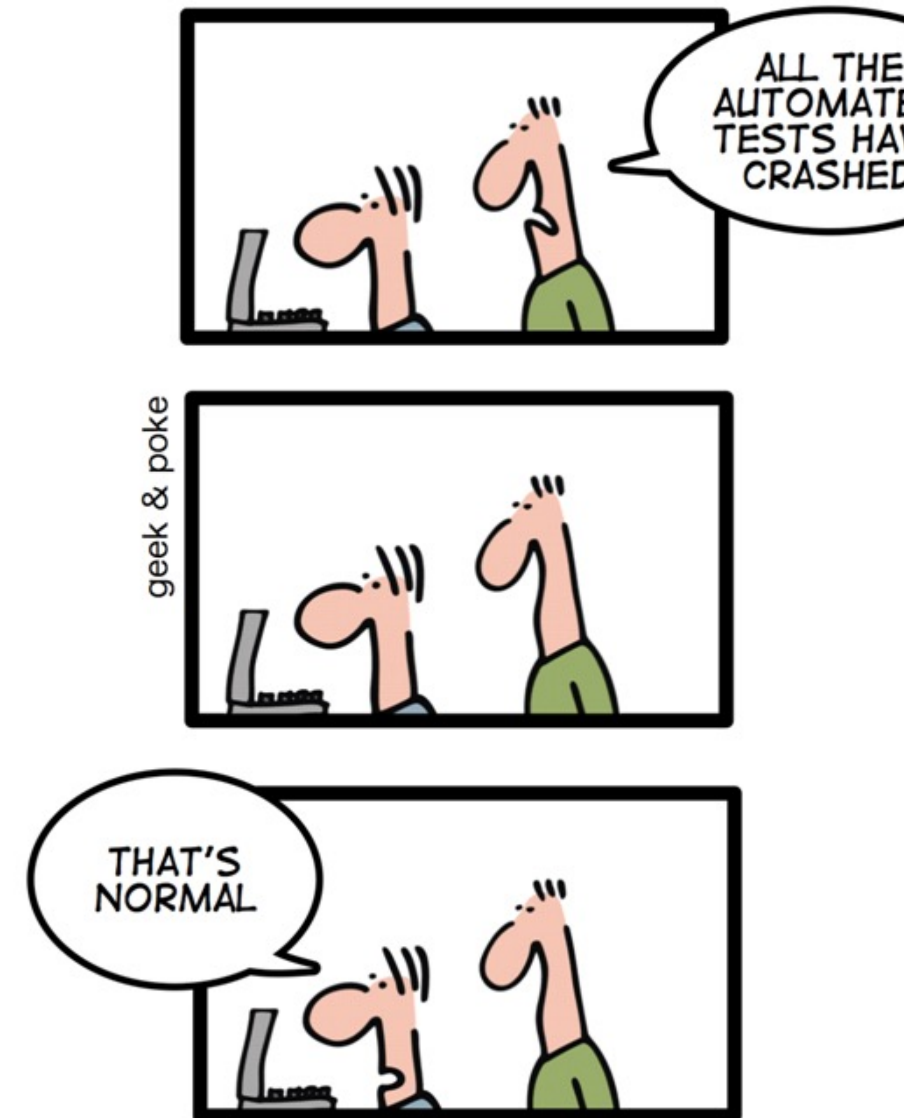
CI - Continuous Integration

“Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. “

Martin Fowler

Geek & Poke List of Best Practices

CI Gives you the comforting feeling to know that everything is normal



https://www.zuehlke.com/blog/app/uploads/2015/11/geek-and-poke.png

Why?

Detect development problems earlier

Reduce risks of cost, schedule and budget

Find and remove bugs earlier

Deliver new features and get user feedback more rapidly

How?

Maintain a single source repository

Automate the build

Make your build self-testing

Keep the build fast

Keep the build on the CI machine

Test in a clone of production environment

Make it easy for everyone to get the latest executable

Make the process transparent for everyone

CI and CD

Summary

Continuous Integration

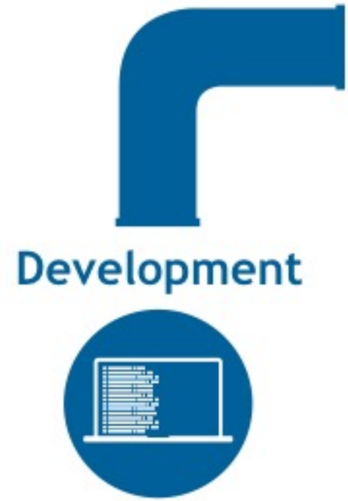
Continuous Delivery

Continuous Deployment

Jenkins

Jenkins

Workflow automation tool



Start

SCM Commit

Repo & Branch Name

Workflow Start

Generate on catch/finally

build-report.html

Stage

SCM Checkout

Collect Dependent Repos

Stage

Build/Launch (Docker)

Parallel

Build/Launch Module 1

. . .

Build/Launch Module N

Stage

Test

Parallel

Integration Test Module 1

. . .

Integration Test Dependency N

Smoke Test

End-to-End Test

Test Results

Stage

Deploy

Workflow End

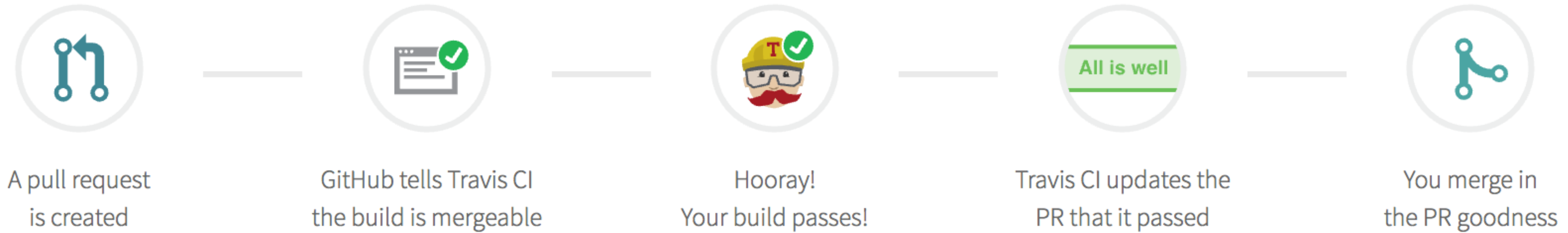
Jenkins

Workflow automation tool - pipelines

```
node { // <1>
   stage('Build') { // <2>
      sh 'make' // <3>
   }

   stage('Test') {
      sh 'make check'
      junit 'reports/**/*.xml' // <4>
   }

   stage('Deploy') {
      sh 'make publish'
   }
}
```

Groovy (JVM-based language)

Travis CI



GITLAB CI

Code and build scripts in the same repo

Easy to start

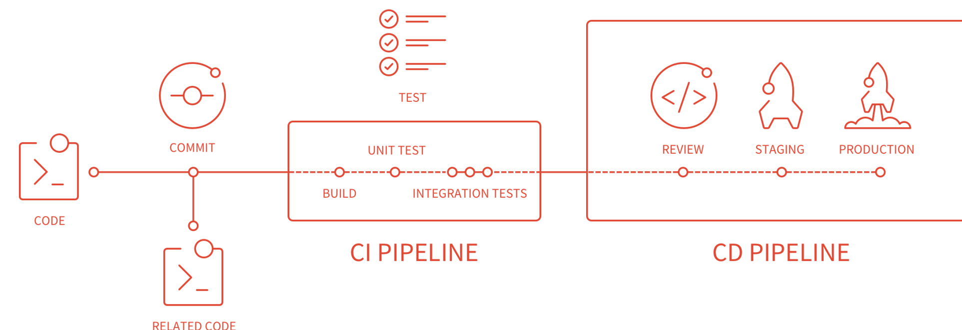
Scalable

Isolated test environment

Gitlab CI: Pipelines and Stages

A pipeline is a group of jobs that get executed in stages(batches). All of the jobs in a stage are executed in parallel, and if they all succeed, the pipeline moves on to the next stage. If one of the jobs fails, the next stage is not executed.

Pipelines are defined in .gitlab-ci.yml by specifying jobs in stages:



Demo