UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

Module 02: CI/CD & DevOps

Advanced Programming

Daya Adianto
Editor: Teaching Team (Lecturer and Teaching Assistant)
Advanced Programming
Semester Genap 2024/2025
Fasilkom UI

Author: *Daya Adianto*
Email: dayaadianto@cs.ui.ac.id

Table of Contents

Learning Objectives

1. Students should be able to understand CI/CD principles.
2. Students should be able to implement automation in the software development process as part of CI/CD.
3. Students should be able to use a build system such as Gradle or Maven.
4. Students should be able to implement automation in maintaining software quality.

References

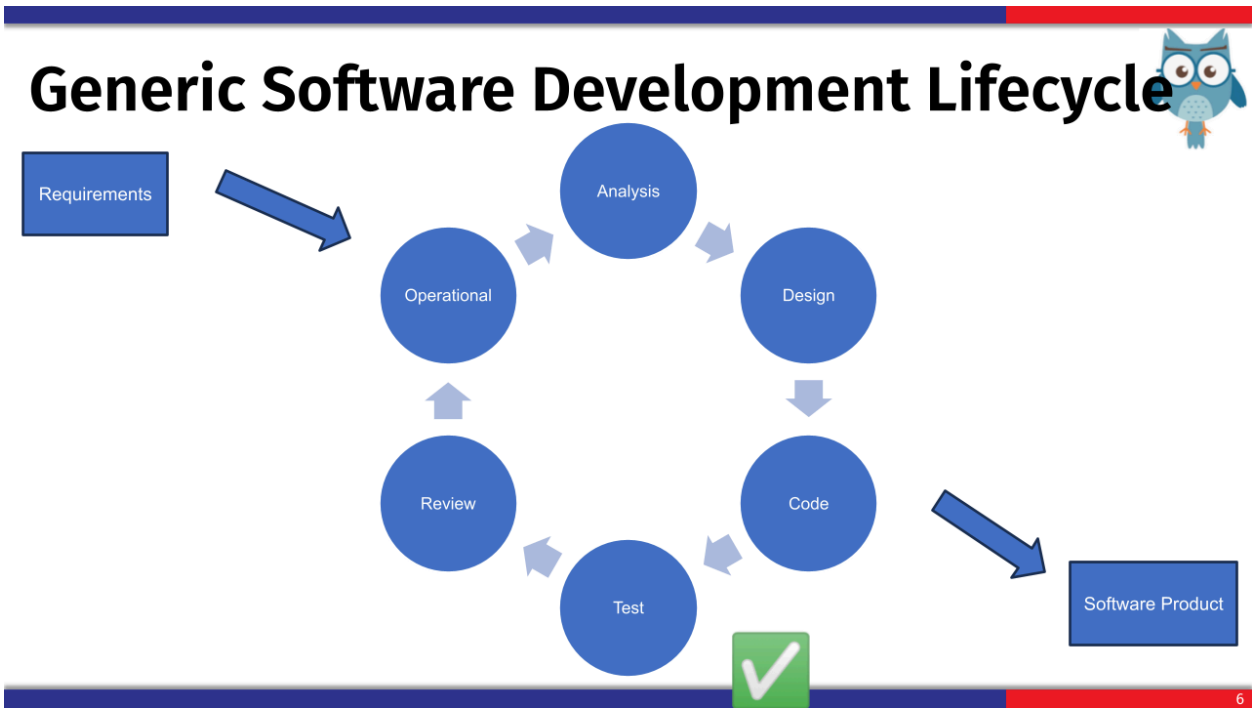
1. Dyck, Andrej, et al. 'Towards Definitions for Release Engineering and DevOps'. 2015 IEEE/ACM 3rd International Workshop on Release Engineering, IEEE, 2015, pp. 3–3.
2. Swaraj, Nikit. AWS Automation Cookbook. Packt Publishing. 2017.
3. Leite, Leonardo, et al. 'A Survey of DevOps Concepts and Challenges'. ACM Computing Surveys (CSUR), vol. 52, no. 6, 2019, pp. 1–35.
4. https://about.gitlab.com/blog/2022/02/11/4-must-know-devops-principles/
5. https://web.devopstopologies.com/

Changelog

V1.0 - Even 2023/2024 - Initial work
V1.1 - Even 2024/2025 - Implemented lecture notes that are previously marked "TBD" and improved formatting consistency.

1. Software Development Lifecycle

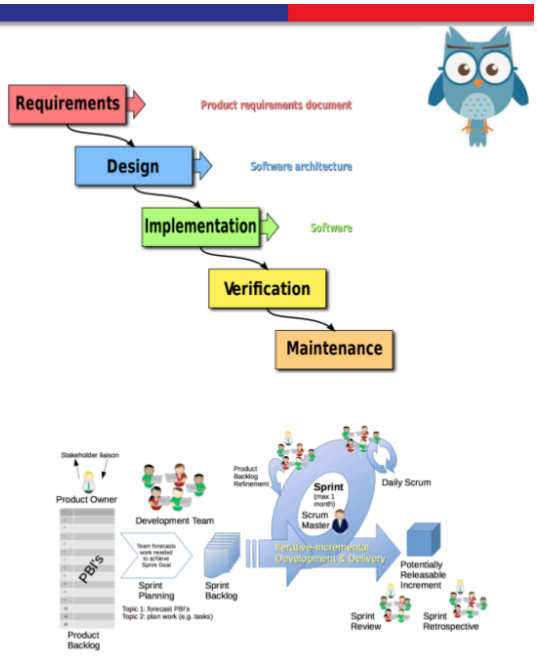
To build a software, we definitely need the specifications of the software. We often describe the specifications as part of the software requirements. Then, based on the existing requirements, we "transform" the requirements through a "process" that eventually results in a software product. The process could involve many activities, where one of them is, well, drinking a beverage.



Joking aside, we usually undergo a process that accepts the software requirements. The process has a series of activities that are done sequentially. Eventually, we produce a set of artifacts that include the software product after one or more cycles in the process.

Development Process

- Development process comprises of sequence of phases & activities to produce software artefacts (inc. source code, assets, configuration) from given requirements
- No matter your development process is (e.g., Waterfall, RUP, Scrum, agile), the **essence** of most development processes is **how to transform given requirements to a software product**

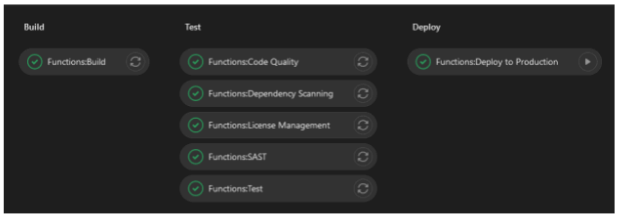


7

The development process may have various stages and activities. It depends on the software engineering textbook that you use, or dictated by organizational policy. However, we can boil down the essence of many software development processes into the process of how to transform the given requirements into a software product.

Process Automation

- Traditional software construction & delivery often still include repetitive tasks
  - Example: running test suite, preparing deployment environment, copying software packages to the deployment environment
  - Anything else?
- Deterministic, repetitive tasks can be **automated** using scripting and software tools



Source: https://gitlab.com/konfui/qir-backend/-/pipelines/71379674

8

Some activities in a traditional software development process may involve repetitive tasks. For example, running test suites every time there are changes to the main branch of a codebase. Deployment may also involve manual actions where the system administrator has to prepare the operating system on the server and install several packages before running the given application from the developer.

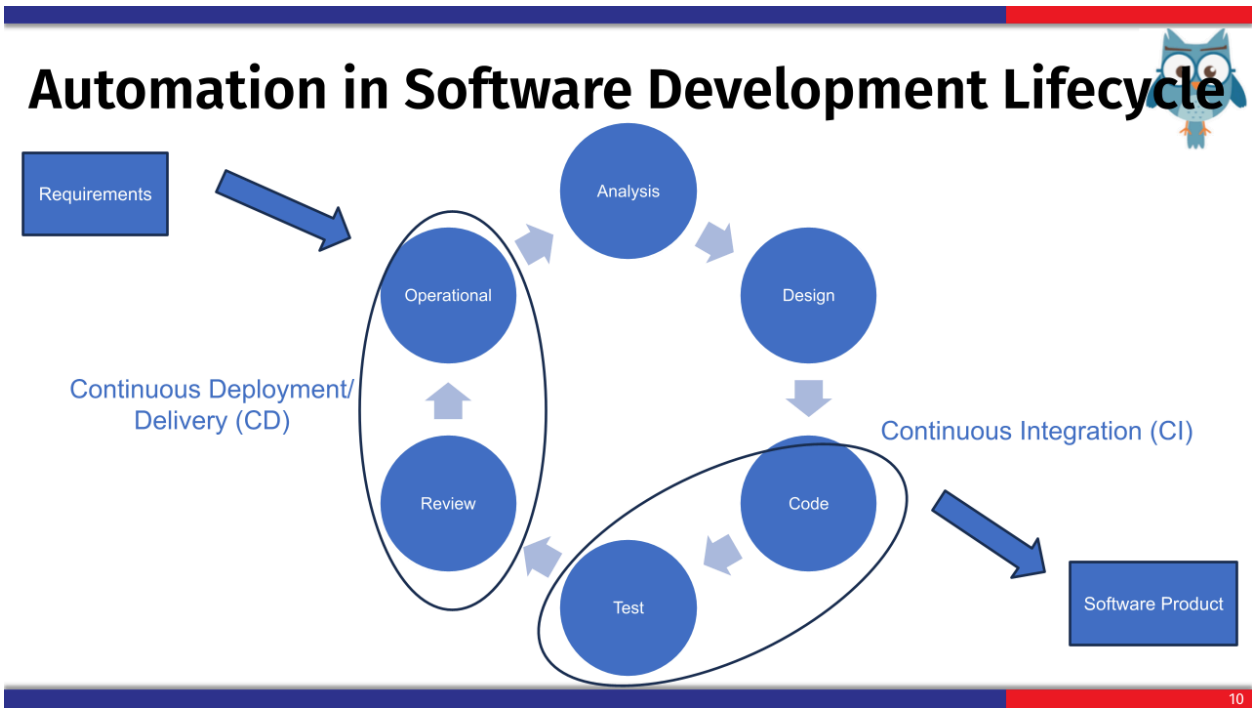
There is nothing wrong with having repetitive tasks done manually. However, it will be better if we can automate those tasks so we can allocate our effort more on doing the job we love, that is problem solving and coding.

Human Factor

- We cannot expect everyone in the team remember to run all the automation scripts
  - E.g. do you always run your test suite in the IDE or terminal each time you made changes?
- To ensure **repeatability**, we let the **machine** to run the scripts

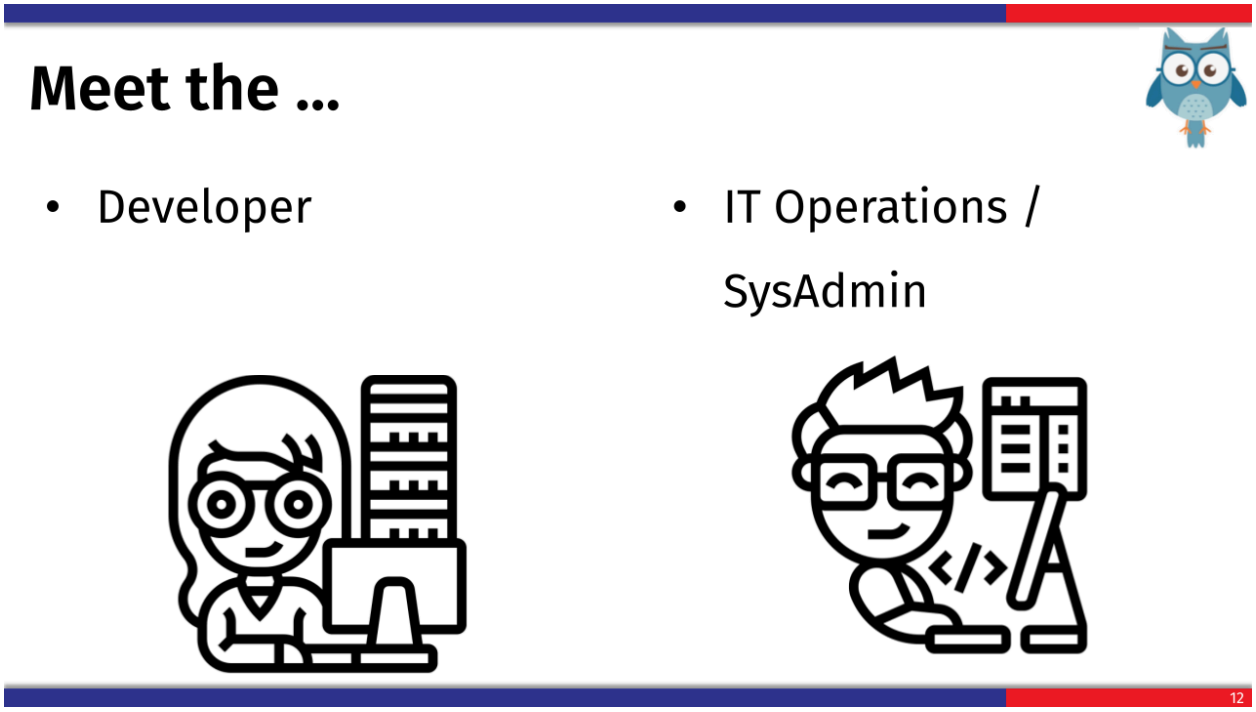
9

The issue with repetitive tasks is we cannot ensure everyone remembers to perform them. Sure, we can have some sort of checklists written in a post-it note and expect every developer to look at them each time they want to integrate their work. However, expecting everyone to follow the checklists will eventually add overhead in the process and the development effort. It is better to spend the work effort actually solving the actual problem (i.e., coding) instead of going through checklists. Therefore, we can try to delegate some of the repetitive tasks to machines.



If we look back at the generic software development lifecycle/process, there are four possible stages that can include automation as part of the activities. The first two stages are Code and Test as part of Continuous Integration. Then, the last two stages that can have automation are Review and Operational as part of Continuous Delivery/Deployment.

1.1 Traditional, Manual CI/CD



Before we look at how we can implement CI/CD, we need to be aware of the historical context. Let's start with the traditional division of roles in a software development organization. There are two main roles: Developer and IT Operations (or sometimes known as System Administrator). If you prefer having a name, you can name the Developer as "Alice", and the IT Ops as "Bob".

What Developer Does ... (Traditionally)

- Gather requirements
- Understand requirements
- Build software
- Test software (locally or on dev. env.)
- Deploy software (locally or on dev. env.)

Test Summary

16	0	1	1.938s
tests	failures	ignored	duration

100% successful

Ignored tests Packages Classes

ApiActivityControllerWebTest. http_getActivities_withBeginAndEnd_ok()

13

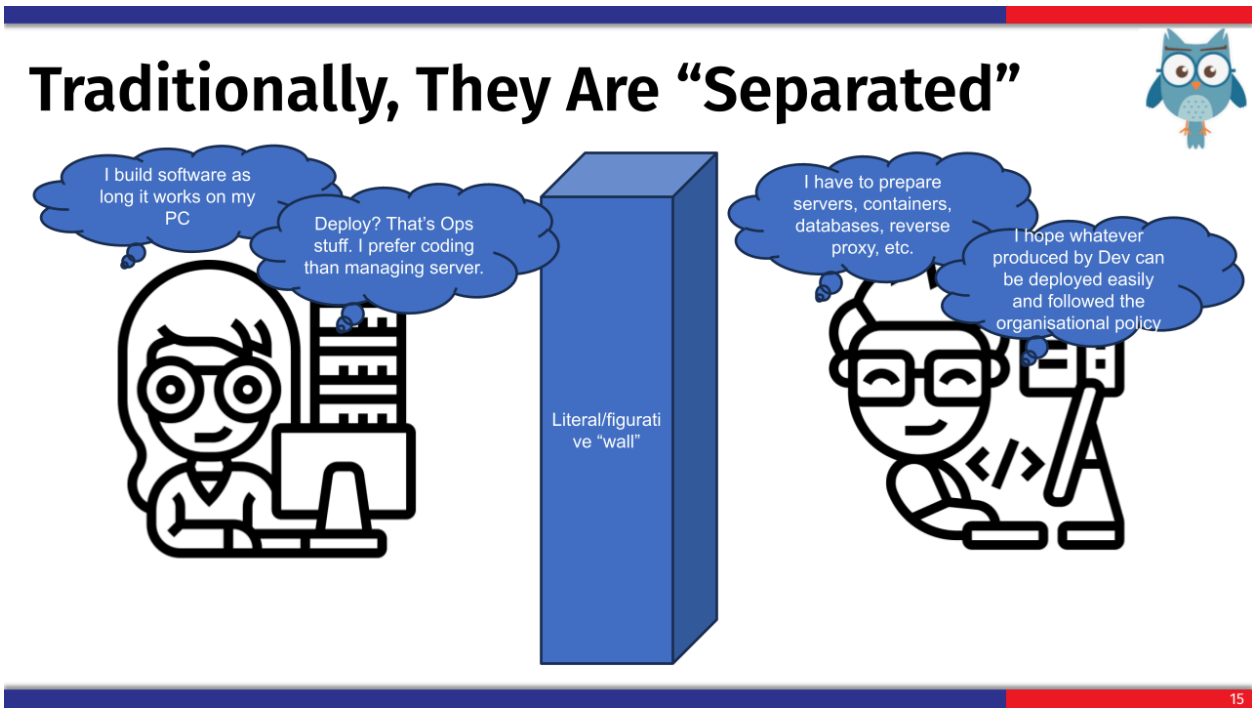
The main job of a Developer encompasses the generic software development process. They transform the given requirements into a software product. They are also able to deploy the software. But, they only deploy locally or to a development environment which has been prepared by the IT Operations specifically for previewing their work (i.e., not actual production or accessed by users).

What IT Operations Does ... (Traditionally)

- Prepare a “deployment environment”
  - E.g., “dev”, “staging”, “prod”
- Deploy software to a deployment environment
- Monitor a deployment environment

14

The IT Operations are often only involved in the process at the end. Once the Developer has finished implementing the software, they give the source code or the executable build of the software to the IT Operations for actual deployment.



Both Developers and IT Operations play an important role. However, traditionally, both of them are separated. The separation can be in the form of physical separation where they do not work in the same room, or organizational separation where they have their own unit/division. This leads to some communication overhead between two roles. In addition, both roles also have their own main concerns. Developers focuses on building the product, whereas IT Ops focuses on running the product.

Discussion: As a Developer ...

- Let's pick a tech stack that you may be familiar with: Python & Django Framework + PostgreSQL database
- How do you prepare your local development environment?
- How do you **build** the app?
  - Using build system such as Gradle? Or using shell script?
- How do you **test** the app?
  - Locally? On a development environment?

16

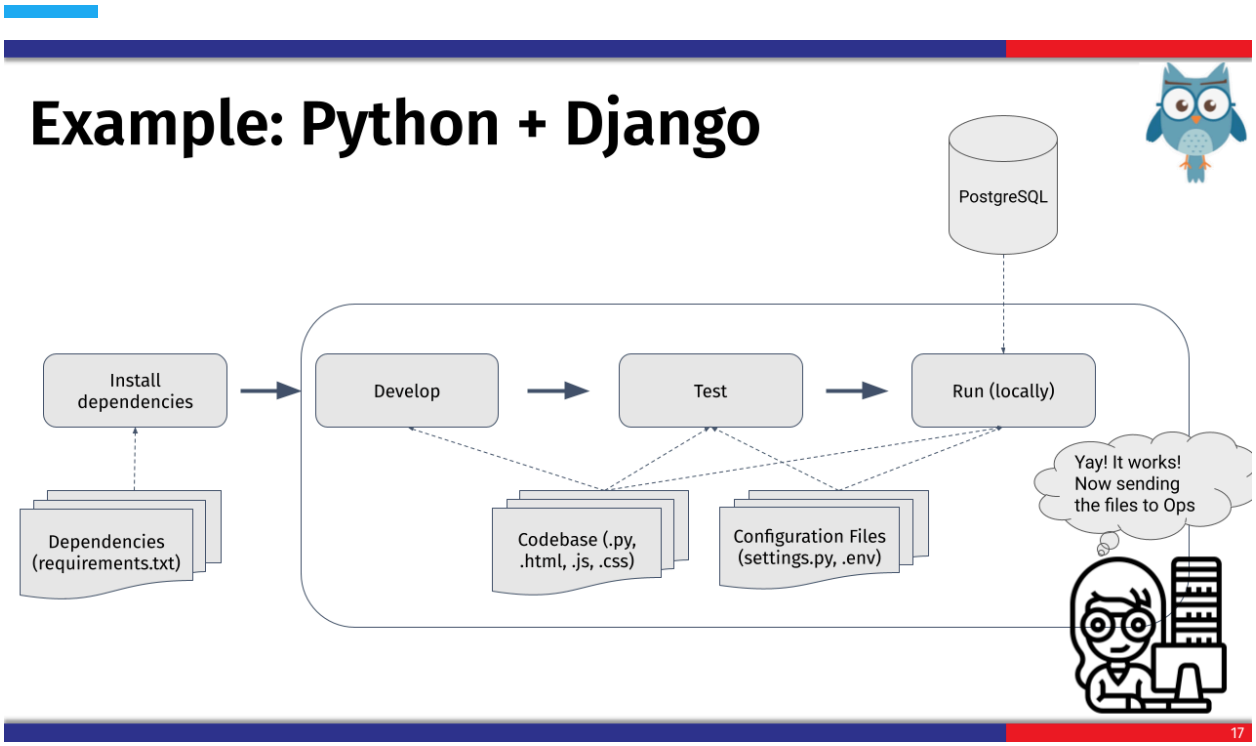
Since you are familiar with the Developer role, let's discuss the software development activities as a Developer. Suppose you look back at a Web app development project from the Platform-Based Development course. You might remember that you used Python as a programming language and Django Framework as the Web development framework. In addition, your app may have used a PostgreSQL database to store the data.

Try to answer following questions:

1. What did you do to prepare the local development environment in your computer?
2. Did you require any command to **compile** or **build** your project to an executable version?
3. How did you **test** your project? What were the commands to run the test?

Based on your prior experience in developing Web applications using Python and Django, you may have noticed that you did not **compile** your project in order to run it. Some programming languages such as Python are **interpreted**. It means that they can immediately run the source code without requiring it to be compiled to another format such as native code or intermediate code.

In contrast, if you have finished the first exercise from last week, Java did require you to compile and build the source code to an executable format (i.e., the JAR file). You are required to run a command using the build system (e.g., Gradle or Maven) to build the project.



If you forgot how to run a Django project, here is a quick refresher:

1. Prepare the local development environment by setting up a virtual environment and install the required dependencies into the virtual environment.
2. During the development, including testing and running the project, you make your changes to the codebase and the configuration files.
3. To ensure your project actually stores the data to a PostgreSQL instance, you modify the configuration (i.e., settings.py) at the database configuration section.
4. Finally, you run the project and test it if it works.

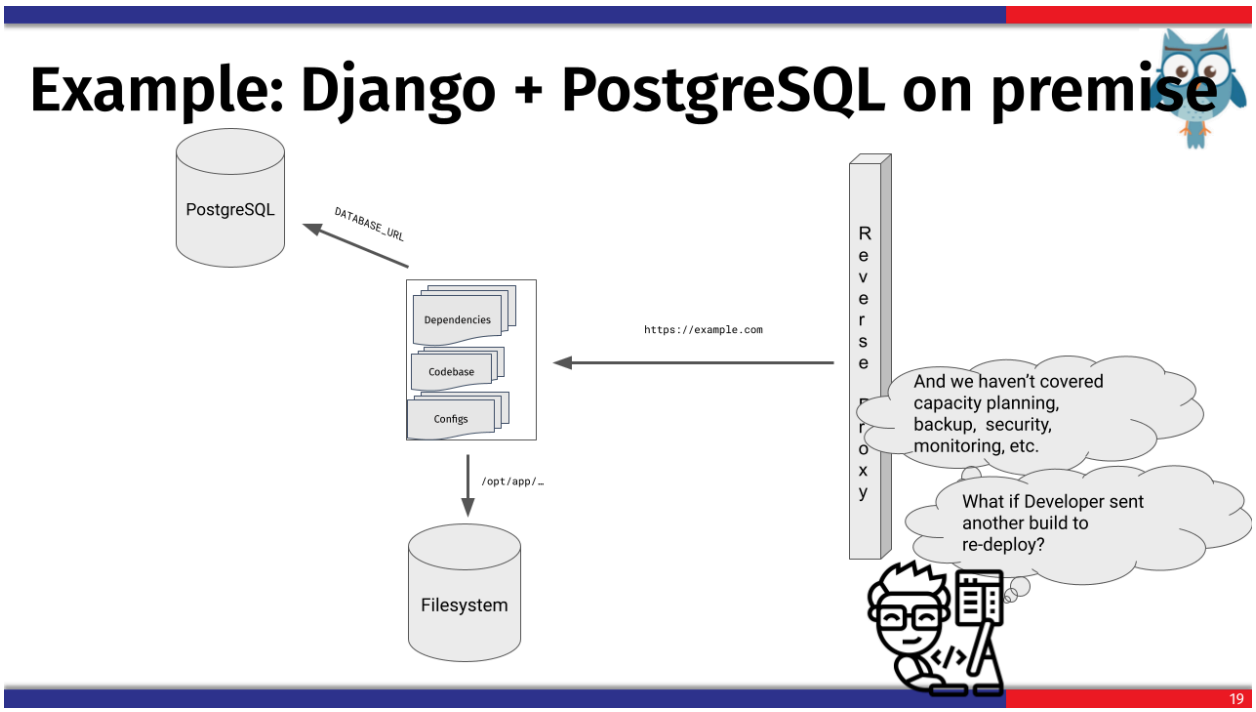
Traditionally, the Developer only deployed their app locally as part of the testing process. They try to verify that all the required features work locally. Most often, there is no collaboration with the IT Operations. They will only interact when the Developer hands over the built codebase to the IT Operations for actual deployment on a staging or production environment.

As an IT Operations ...

- What information do you need to deploy the app?
- Do you have enough resources (capacity) to deploy the app?
- How the app will be deployed according to the organisation's policies?

18

Let's look at IT Operations. They are responsible for managing computational resources available in the organization. They need to plan how many resources can be allocated for deploying a project from the Developer. Moreover, they also need to ensure that the project developed by the Developer meets the organizational policies so it can utilize the allocated computational resources.



Here is a quiz to jog your memory about running a Django app on a deployment environment. What is the command to run a Django app on an actual deployment setting? Is it:

A. python manage.py runserver

   or,

B. gunicorn DJANGO_PROJECT_NAME:application

If you answered A, you are partially correct. It is true we can run a Django app using runserver command. However, it is only suitable for development purposes. The runserver command makes it convenient for developers to run and test the app locally but at the expense of performance. The B option, i.e., gunicorn command, is more appropriate for running the app on a production environment since it runs the Django app using a dedicated application server.

There are also other factors to consider when deploying a Django app on a deployment environment. The most obvious one, especially if you are using Windows, is the operating system (OS). Since the majority of servers are using Linux-based OS, we have to ensure paths to any resources in the filesystem are correct. In addition, there is also a subtle issue about line ending format where Windows uses CR LF and Linux uses LF.

To summarize, deploying a Django application is not as simple as it seemed. Pretty often developers are only aware that running a Django application is simply by running python manage.py runserver. They often forget that there are other factors to consider when deploying the Web app in an actual production environment.

Summary

- Different concerns between Developer & IT Operations
  - **Building** vs. **running** a product
- Dev and Ops are highly specialised role, so it is understandable if there is a "wall" separating them
  - If we want to increase our velocity in delivering software, that "wall" needs to be tear down
- Deploy may happen multiple times!
  - It is pretty unlikely to get a software built and deployed correctly for the first time
- There must be a better way, right?
  - Automation → **CI/CD**
  - Culture → **DevOps**

20

As have been outlined, Developer (Dev) and IT Operations (Ops) have different concerns. The first specializes in **building** a software product, while the latter focuses on **running** a software product. In a traditional software development process, both roles may also be separated from each other. However, in the times where we want to enhance the agility of the development process, we may want to make Dev and Ops collaborate closer.

To improve the agility of the software development process, we can start by improving on two aspects: process **automation** and development **culture**. The process automation is realized by implementing Continuous Integration & Continuous Delivery (**CI/CD**), while the development culture is improved by implementing **DevOps** practices.

2. Continuous {Build, Integration, Deployment, Delivery}

Continuous Integration (CI)

*"A software development practice where continuous change & updates in codebase are integrated and verified by an **automated build script using various tools.**" (Swaraj, 2017)*

- The common theme in Continuous {X} is **automation** and **tools support**
- Ideally supported with tests
  - We want a working, runnable, and **tested** product
  - Without tests **Continuous Build**
- CI examples
  - Run the test suites for every new changes to the codebase
  - Create an executable build of the software in daily basis
  - Scan for any potential security issues in the included 3rd party libraries used in the codebase

22

Continuous Integration (CI) is "a software development practice where *continuous changes & updates* in codebase are integrated and verified by an **automated build script using various tools**." (Swaraj, 2017) It is a practice where we want to automate the process of integrating changes by utilizing tools.

As part of verifying changes continuously, we also include testing as part of the process. We want to produce a working, runnable, and **tested** software product. If we did not include testing as part of the process, we are still able to produce software products but with **unverifiable quality.**

The Main Recipe of CI

Source: https://commons.wikimedia.org/wiki/File:Cosplayer_of_Pikotaro_at_CWT44_20161211.jpg

23

If you remember a viral video on YouTube from 7 years ago, there is a video depicting an eccentric person who stabs a pen to an apple. We can have an analogy where tests and automation are kind of the “pen” and “apple” in a CI. In order to implement CI, we have to have “tests” and run it along with “automation” in the software development process.

Continuous Deployment (CD)

"It is also a software development practice. Its role is to automatically deploy the code to the application folder on the specified server" (Swaraj, 2017)

- Takes one step further from CI where the automation also include the delivery/deployment process to an environment
- Some also refer CD to Continuous Delivery
  - Delivery **manual** deploy/release
  - Deployment **automatic** deploy
- CD example:
  - Automatically deploy the latest build to an environment (e.g., cloud, on-premise server)

24

We can implement CI further to include the delivery or deployment process. We refer the further improvement to CI as Continuous Deployment (CD). In CD, we are not only automating the build and testing of a software, but also the delivery process.

The delivery process here can be still performed manually or fully automated. If we only produce a runnable product at the end of a CD process, then the "D" in the "CD" stands for "Delivery". However, if we implement the delivery process until automating the delivery (or, "deployment"), then the "D" in the "CD" stands for "Deployment".

Traditional CI/CD

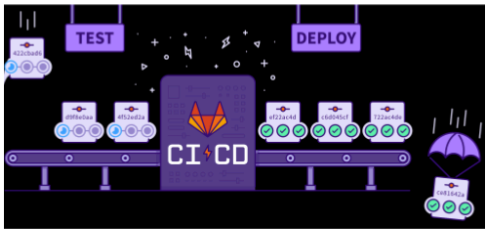
- Traditional CI/CD involves a lot of **scripting**
- Both Dev and Ops may develop their own scripts/tools as part of the CI/CD process
- Custom built CI/CD is often an one-off endeavor → not reusable to different projects

25

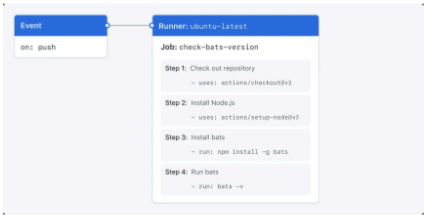
In a traditional, “old school” CI/CD implementation, we use many scripts (e.g., Batch scripts in Windows, Bash shell scripts in Linux) to perform repeated tasks. The Dev may write their own script to automate the build process of their project. Similarly, the Ops may also write their own script to automate the deployment process. Pretty often these scripts are built in an ad-hoc manner and may not be reusable to different projects.

Modern CI/CD

- Use existing tools & platforms available in the community as a framework for building CI/CD
  - Still involves scripting, but the execution will be automated



Source: https://about.gitlab.com/images/ci/ci-cd-test-deploy-illustration_2x.png



Source:
https://docs.github.com/assets/cb-62091/mw-1440/images/help/actions/overview-actions-event.webp

Nowadays, CI/CD can be implemented by reusing tools and platforms available in the community. Tools such as Maven and Gradle in the Java ecosystem allow us to build and run Java-based projects without compiling each Java file one by one. Similarly, platforms such as GitHub Actions & GitLab CI/CD allow us to automate CI/CD on a project that we host on a Git repository.

Modern CI/CD still involves scripting, but there are existing libraries or scripts that the developers can reuse. In addition, some CI/CD are implemented on a platform where the execution of the scripts are automated on the platform.

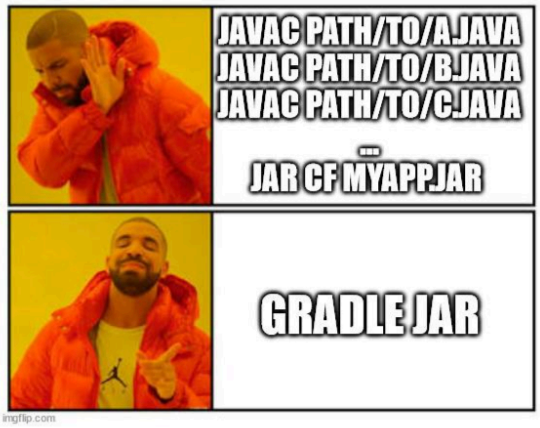
2.1 Gradle Build Tool

Why?

- The main reason:

  automation

- Other reasons:
  - Dependency management
  - Plugin ecosystem



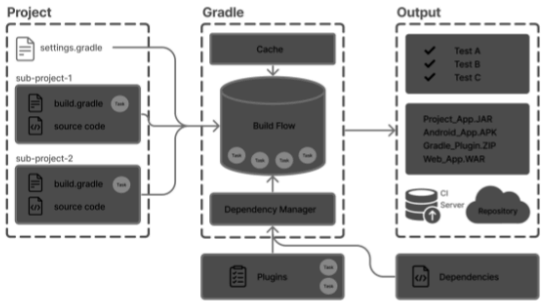
28

In the Java ecosystem, we often use a build tool such as Gradle. The main reason for using such a tool is mainly to automate tasks performed when developing Java-based projects. For example, it would be very tedious if we have to compile each Java file if we are working on a medium- or large-sized Java project. Additionally, there might be other tasks involved when building the project such as running the test suite and building the executable version of the project. Instead of writing our own custom scripts, we can reuse a task provided by the Java plugin in Gradle to automate, for example, the compilation process to produce a JAR file.

There are also other reasons, such as managing dependencies used in developing a Java project and reusing community-developed plugins to assist the CI/CD process of the project.

Core Concepts

- Project → a piece of software that can be built
  - Single project builds include a single project called the root project
  - Multi-project builds include one root project and any number of subprojects
- Build scripts → Steps to take to build the project
- Dependency management → automated technique for declaring and resolving external resources required by a project
- Tasks → basic unit of work executed as part of build script or plugin
- Plugins → extend Gradle's capability and may contribute additional tasks to a project



Source: https://docs.gradle.org/current/userguide/img/gradle-basic-1.png

30

There are five core concepts according to Gradle official documentation that we need to know. Let's start with the `Project`. The `Project` represents the codebase of a software that we develop. The `Project` can represent a single codebase called the root project, or multiple codebases represented as subprojects

Build Scripts represent the steps to run in order to build the Project. The build script includes the required dependencies, tasks, and plugin.

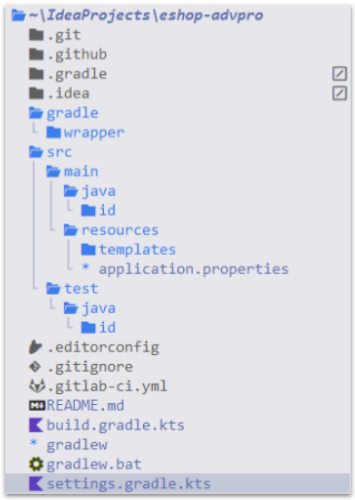
Dependency Management manages the external resources used by the Build Script to build the Project. The external resources are not only limited to 3rd party libraries downloaded from the Internet, but also Gradle plugins and output of a subproject if you are developing a multi-project.

Tasks represents the unit of work executed as part of a build script or plugin. It could be a series of actions that are encapsulated into pieces of code that are executed by Gradle.

Lastly, we have **Plugins**. Plugins are software components for extending Gradle's capability and may contribute additional tasks into a Project.

Project Structure (Single Project)

- Main indicators
  - gradle directory in the root directory
  - gradlew/gradlew.bat scripts in the root directory
  - build.gradle & settings.gradle files in the root directory
  - settings.gradle only contains single project declaration (i.e., root project)



30

To identify a Gradle-managed project, we can look at the structure of the workspace directory. The main indicators that we can find to see if a project is using Gradle are as follows:

1. There is a directory named `gradle` in the root directory.
2. There are `gradlew` or `gradlew.bat` shell scripts in the root directory.
3. There are `build.gradle` and `settings.gradle` in the root directory.
4. The `settings.gradle` only contains a declaration of a single project (or also known as *root project*)

Project Structure (Multi Project)

- Similar to single project
- There are subproject directories, along with the corresponding build script
- settings.gradle contains a root project and subprojects

```
project
├── gradle
│   ├── libs.versions.toml
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
├── settings.gradle(.kts)
├── subproject-a
│   ├── build.gradle(.kts)
│   └── src
└── subproject-b
    ├── build.gradle(.kts)
    └── src
```

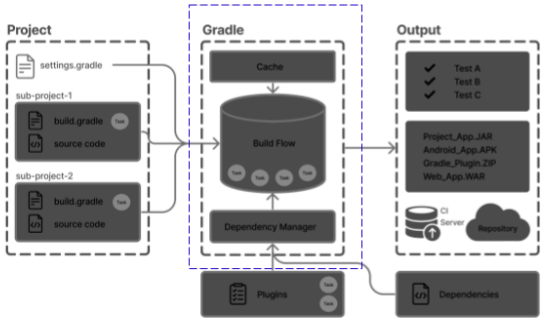
Source: https://docs.gradle.org/current/userguide/gradle_basics.html#gradle

31

The structure of a Gradle multi-project is similar to a single-project. Both have the usual `gradle`-related root files and directories. However, a Gradle multi-project may have one or more subdirectories that contain a Gradle single project. The `settings.gradle` in the root project also contains the name of the root project and the included subprojects.

gradlew (Gradle Wrapper)

- Although you can install Gradle locally, it is more recommended to use the included Gradle Wrapper in a project
- Why?
  - Minimise development environment parity → ensures everyone (people & machine/CI server) uses the same Gradle version



Source: https://docs.gradle.org/current/userguide/img/gradle-basic-1.png

32

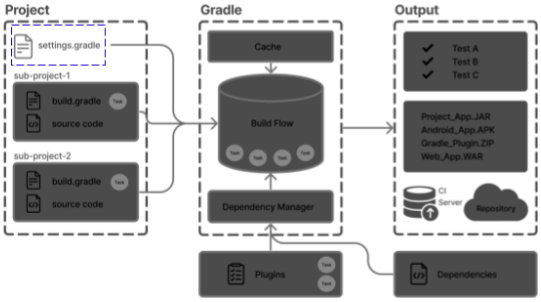
While it is possible to install Gradle, it is more recommended to use the provided Gradle Wrapper in a project. By doing so, we can minimize the environment parity among the team members and environments.

One may ask, “what is environment parity”? It means the degree of differences found in development environments used by all parties (both human and machine) in a software development process. We strive to have the same development environments on local development (i.e., developer’s computer) and production environment to ensure the project can run on both environments with minimal modification. For example, if the project uses PostgreSQL, then both the local development environment and production environment need to install PostgreSQL as well.

To learn more about the environment parity, please refer to the “Dev/prod parity” factor in the Twelve-Factor App methodology here: https://12factor.net/dev-prod-parity

Settings File

- settings.gradle defines a Gradle project and any subprojects
- Optional in a single project
  - You can omit settings.gradle if you only have a root project



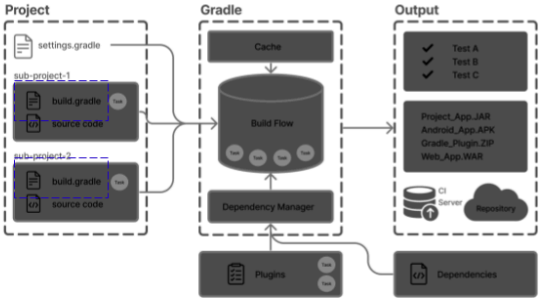
Source: https://docs.gradle.org/current/userguide/img/gradle-basic-1.png

33

The `settings.gradle` file defines the root/top-most Gradle project and any subprojects. At minimum, it contains the name of the root Gradle project. You can omit the file if you are working on a single project. But it is still recommended to have one in the project, regardless if you are working on single or multi projects.

Build File

- Describes build configuration, tasks, and plugins



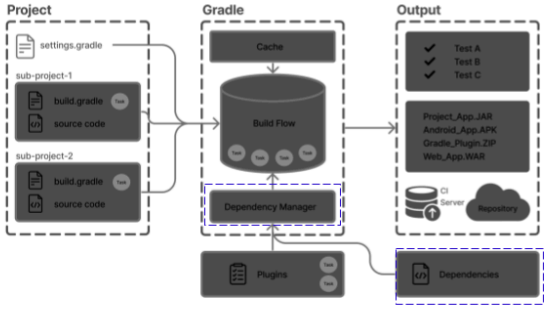
Source: https://docs.gradle.org/current/userguide/img/gradle-basic-1.png

34

The build file (i.e., `build.gradle`) describes various elements involved in running the build script in Gradle. It includes the build configuration (e.g., project versioning, dependencies, rules for certain tasks), the tasks, and plugins in the project.

Dependency Management

- Dependencies refer to external resources such as:
  - JAR (included manually or from different subprojects)
  - Plugins
  - Libraries
  - Source code



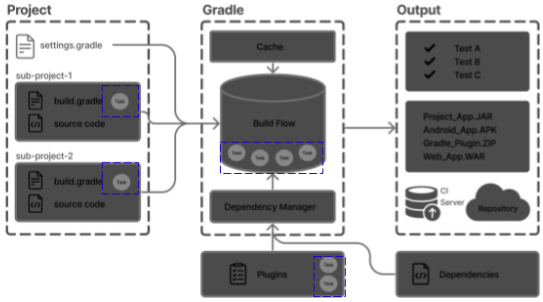
Source: https://docs.gradle.org/current/userguide/img/gradle-basic-1.png

35

As mentioned before, dependency management in Gradle manages the dependencies or the external resources. The external resources include JAR files, plugins, libraries, and source code of a subproject.

Task

- Independent unit of work that a build performs
  - Example: compiling classes, creating a JAR, run test suite
- A task may have dependency from another task
  - Will determine the task execution order by Gradle



Source: https://docs.gradle.org/current/userguide/img/gradle-basic-1.png

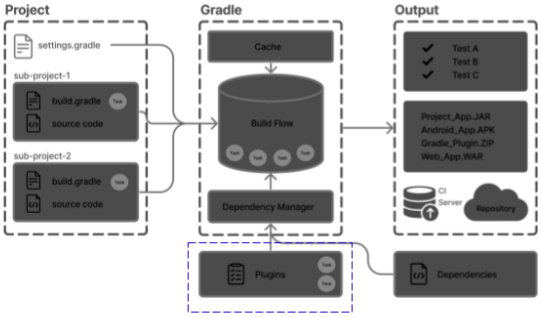
36

Task is arguably the most important component in Gradle. It represents the “unit of work” that a build performs. The “unit of work” in this context refers to the action done by developers had there been no automation in building their work. For example, the action to compile Java classes can be encapsulated into a task that can be reused in a Gradle project.

It is also possible to determine the task execution order in Gradle based on dependency between tasks or order defined by the developer. You can see the list of possible task ordering (e.g., `dependsOn`, `finalizedBy`, `doFirst`) in the Task API here: https://docs.gradle.org/current/javadoc/org/gradle/api/Task.html

Plugin

- Extends build capabilities and customises Gradle
- A plugin includes new tasks
- Example: java plugin (tasks related to compiling & building Java project)



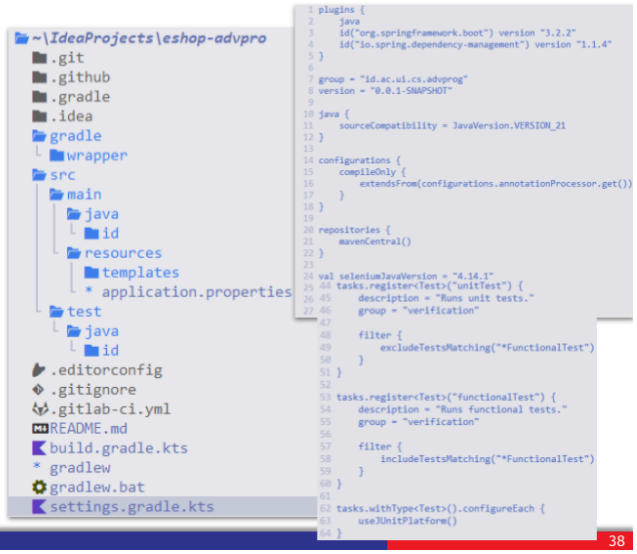
Source: https://docs.gradle.org/current/userguide/img/gradle-basic-1.png

37

Gradle plugins offer additional capabilities to a Gradle project, including new tasks that can be executed in the build script. For example, the `java` plugin in Gradle provides tasks related to compiling and building Java projects.

Example: Module 1 Exercise

- What is the root project name?
  - Hint: Look in settings.gradle file
- Is it a single project or multi project build?
- How many dependencies declared in the build script?
  - Hint: Look at the dependencies block in build.gradle file
- What are the new tasks declared in the build script?
- What plugins used in the build script?



Let's look at an example from the previous module. The Module 1 exercise uses a Gradle build script written in Kotlin (i.e., `build.gradle.kts`). If we look at the `settings.gradle.kts` file, we can see the name of the project. The project itself is a single-project since there are no subfolders representing a Gradle project.

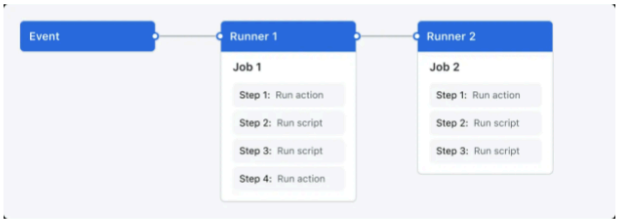
Now let's look at the content of `build.gradle.kts`. There are three plugins used in the project: `java` and two plugins related to Spring Framework. The `java` plugin is configured to parse Java source code using Java version 21. Meanwhile, the plugins related to Spring Framework provide several new tasks into the Gradle project and manage the dependencies required by the framework.

We can also see that there are two new tasks with type `Test` that are declared in the build script. The new tasks specify the test suite to be executed when running unit and functional tests.

2.2 CI/CD on GitHub

GitHub Actions

- The CI/CD platform provided by GitHub
- Actions vs Workflows?
  - Action: the task to be executed as part of a workflow
  - Workflow: series of actions
- Configuration via YAML-formatted files
- Each jobs in a workflow will be executed on a **runner**



Source: https://docs.github.com/assets/cb-25535/mw-1440/images/help/actions/overview-actions-simple.webp

40

Let's look at GitHub Actions as an example of a CI/CD platform. It is pretty likely that you are already familiar with GitHub Actions from your previous Platform-Based Development course.

As its name implies, GitHub Actions is a CI/CD platform that is offered by GitHub. It allows GitHub users to implement CI/CD in the same platform as where they put their codebases. By doing so, the development process can be more integrated and the users do not have to switch to multiple tools used in the development.

Sometimes you may also see workflows mentioned with GitHub Actions. GitHub Actions is actually the product name offered by GitHub. It provides **action** and **workflow** as part of the CI/CD implementation. The **action** refers to the tasks to be executed as part of a **workflow**. The **workflow** itself refers to a series of **actions**.

Basic Concepts

- Workflows → a configurable automated process that will run one or more jobs
- Events → a specific activity in a repository that triggers a workflow run
- Jobs → a set of steps in a workflow that is executed on the same runner
  - Could be a **shell script/command** or an **action**
- Actions → **a custom application** for the GitHub Actions platform that performs **a complex but frequently repeated task**
- Runners → a server that runs a workflow

41

According to GitHub Actions official documentation, there are five basic concepts we need to know in using GitHub Actions. Let's start with Workflow. Workflow represents a process that will run one or more jobs automatically. Event is an activity in a repository that triggers a workflow execution.

Then, we have a Job that comprises a set of steps in a workflow that is executed on the same runner. A job could be executing a shell script/command, or running an action available on GitHub. The Action is actually a custom application on the GitHub Actions platform that encapsulates frequently repeated tasks. And lastly, there is Runner which is a server that runs a workflow.

Example: Automate running Django test suite

- The workflows will run whenever there are push & pull_request events to any branches
- The workflow comprises of single job with four steps
  - Check out/clone the Git repo
  - Set up Python runtime environment (e.g., python, pip)
  - Download the dependencies
  - Run the test suite

.github/workflows/test.yml

```yaml
name: Run the test suite

on:
  push:
  pull_request:

jobs:
  test:
    runs-on: ubuntu-22.04
    steps:
      - name: Clone the Git repository
        uses: actions/checkout@v4
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          cache: pip
          cache-dependency-path: |
            requirements.txt
      - name: Download dependencies
        run: pip install -r requirements.txt
      - name: Execute tests
        run: python manage.py test
```

42

Let's look at a workflow that can be used to run the test suite of a Django project. If we break it down to each basic concept, we have:

- Workflows: there is a workflow named "Run the test suite", which is declared in a file named `test.yml` stored at `.github/workflows` directory of the project.
- Events: the workflow will run when there are events related to `push` and `pull_request` on any branches. It means that the workflow will run each time a developer performs a `git push` or submitting a pull request on the repository.
- Jobs, Actions, Runner: there is a job named `test`. The job is specified to run on a Ubuntu (Linux)-based server. The steps in the job include the usage of several actions such as:
  - actions/checkout ➔ Check out (or, `git clone`) the codebase into the runner's filesystem
  - actions/setup-python ➔ Prepare Python toolchain (e.g., `python`, `pip` commands) in the runner

  The last two steps in the job are running shell commands to install the dependencies and run the test suite.

Example: Automate Django deployment to a PaaS

- Question: What is the condition that will trigger the deployment?
- Deployment steps will depend on the deployment environment
  - E.g., the example uses dokku/github-action because the deployment environment used in PBD last semester was Dokku self-hosted PaaS

.github/workflows/deploy.yml

```yaml
name: Deploy

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-22.04
    steps:
      - name: Clone the Git repository
        uses: actions/checkout@v4
        with:
          fetch-depth: 0
      - uses: dokku/github-action@v1.4.0
        with:
          git_remote_url: "ssh://MY_USERNAME@pbp.cs.ui.ac.id:22/MY_DJANGO_APP"
          branch: main
          command: deploy
          ssh_private_key: ${{ secrets.SSH_PRIVATE_KEY }}
```

43

Let's look at another workflow example. You may also be familiar with this example because it might be similar to the workflow you used during the PBP/PBD course to deploy your Django project.

In this example, the workflow will only be triggered on a certain event: any push to the `main` branch. Then, the jobs have two steps: check out the Git repository and use an action to deploy the project to a PaaS named Dokku.

Remember that the workflows mentioned in the last few pages are only an example. You may follow them verbatim. But most likely, you need to customize them to your needs. For example, if you are deploying to another PaaS other than Dokku, then the deployment step needs to be different. Please consult the documentation of your deployment environment when trying to deploy your project.

Summary

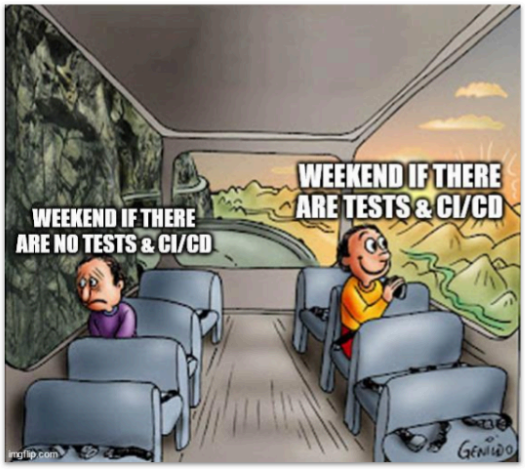
- CI/CD is not only for automating test and deploy, but also for maintaining **various software qualities**
  - E.g., static analysis related to code quality and security
- Issues found during test and deploy can be detected and reported earlier
- Automated test and deploy may lead to faster roll-out of software feature
  - increase customer satisfaction and allow quicker feedback

CI/CD is pretty flexible. We can use CI/CD not only for automating test and deploy, but also automating other activities in software development such as maintaining software qualities (or performing quality assurance). You will see more on this aspect of automating quality assurance in the tutorial and exercise.

2.3 The Big Picture: Software Quality Assurance

Motivation

- Why bother?
  - Producing good quality software product
  - Part of the software requirements (i.e., non-functional requirements)
  - Supporting software maintenance in the future



46

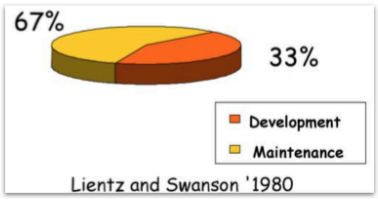
Why bother doing software quality assurance? Of course, as an engineer, we must strive for the best quality for our products. How can we measure the “quality”? We measure the software quality based on “metrics”. These metrics are embedded onto the software requirements, in this case, the non-functional requirements.

Non-functional requirements are things that are not related to what functions/uses our application will be for, but are essential to the application. For example, an application must be able to handle 100.000 unique users accessing the app at one single time. Or for example, the latency of our API should be below 10 ms (milliseconds).

Apart from non-functional requirements, we also want to make sure that others understand our code. We don’t want to be bothered by someone confused on our code that we made 10 years ago, right? This is what we call by “maintainability” ➜ another's ability to maintain the software without the presence of the original author.

Maintenance vs Development

- Software maintenance cost is generally ***twice*** as expensive as development cost.
  - Partly due to the length of time the software is in use.
  - Software Product is not the final goal.
  - Maintenance becomes a significant issue.



47

Software development projects do not necessarily stop once we have delivered the product to the customer. We often have to support, or maintain, the software once it is in the production environment.

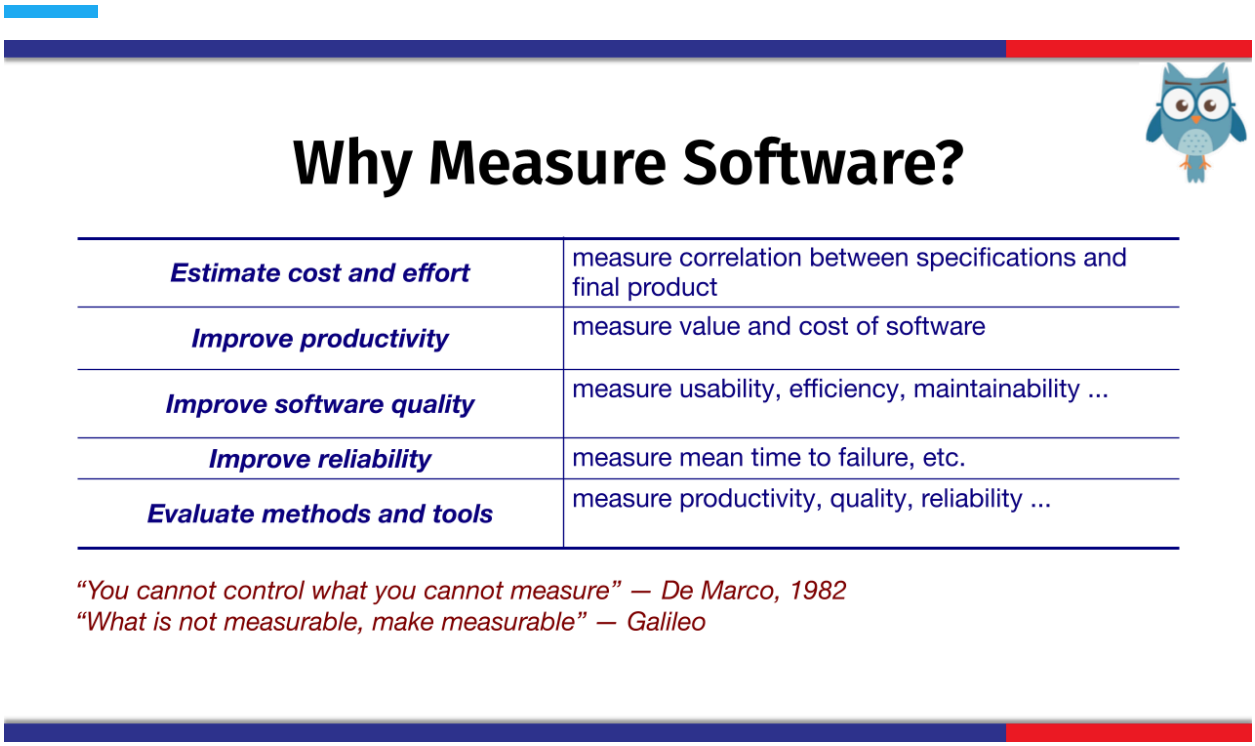
In some surveys, the total maintenance cost may grow larger than the development cost. One of the reasons is due to the time period needed to support the software. The software itself might also be a product that is continuously developed.

Where Do We Start?

- Measure software development lifecycle and its deliverables into quantitative values (“metrics”)
- We will only focus on software quality (inc. maintainability) in this course
  - Take the Software Quality Assurance course if you want to learn more

48

There are some factors to be included as part of estimating cost. But in this topic, we will focus on quality factors such as maintainability. If you want to learn more, you can take the Software Quality Assurance course that is offered as an elective course.



"You cannot control what you cannot measure"

How do we control software quality if we don't have a goal or definition about what that "quality" should be? Defining goals and then defining the measurement towards the goal are the cornerstone of software quality management. In this slide, we define 5 goals:

- **Cost and effort:** Measure correlation between specifications and final product, it can be done using proper documentation on the specifications. For each specification, we need to define "acceptance criteria" checklists. System and user acceptance testing will check whether those "acceptance criteria" are fulfilled or not.
- **Productivity:** Given the specifications and acceptance criterias, product managers can define the business value and predict the cost (man-hours and operational costs).
- **Software quality**: There are several quality aspects that we can refer to, such as McCall's quality factors. For each quality factor, we need to define the indicators relevant to them, and the target metric. We can use these when doing system testing and application performance monitoring (APM).
- **Reliability**: Same as software quality, we can define numerical metrics such as "mean time to failure". This can be used for APMs and alerting.
- **Methodology**: Choosing correct software development methodology and development tools for your development team are also crucial to maintain productivity and software quality.

This module will focus on maintaining **software quality** goals, using CI/CD flows.

Measuring Quality

Internal Quality

- Measured from a codebase
- Examples:
  - Code coverage
  - Amount of code quality issues

External Quality

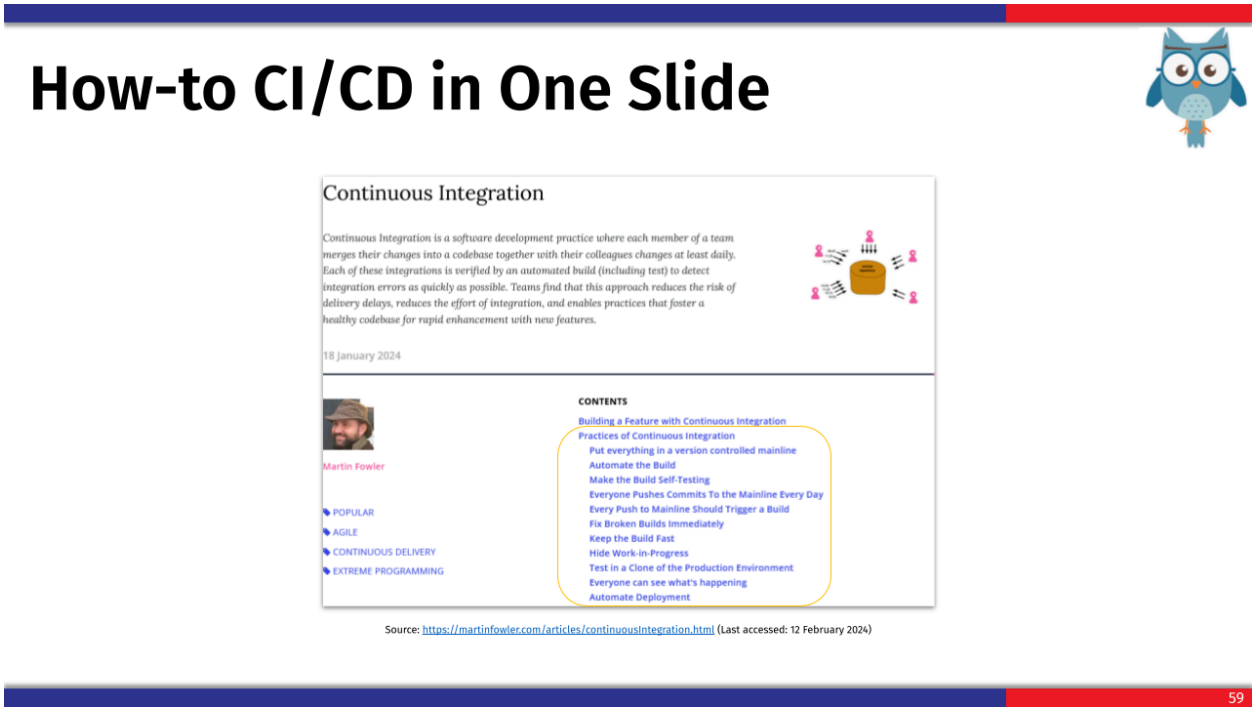
- Measured from a running product on an environment
- Examples:
  - Amount of pass/fail functional tests
  - Performance measurements (e.g., response time, time to start render)

50

Now we went deeper into how we should measure software quality. There are 2 parts of this: internal quality and external quality.

Internal quality is the quality aspect we can measure from our own codebase. For example, unit testing coverage, code convention mistakes, method length, etc. **External quality** is the quality aspect we can measure from a deployment environment. For example, running functional tests, running load/stress tests to measure response time, etc.

2.4 CI/CD Best Practices



You can read how to do Continuous Integration on [Martin Fowler's page](). But we will summarize them a bit here:

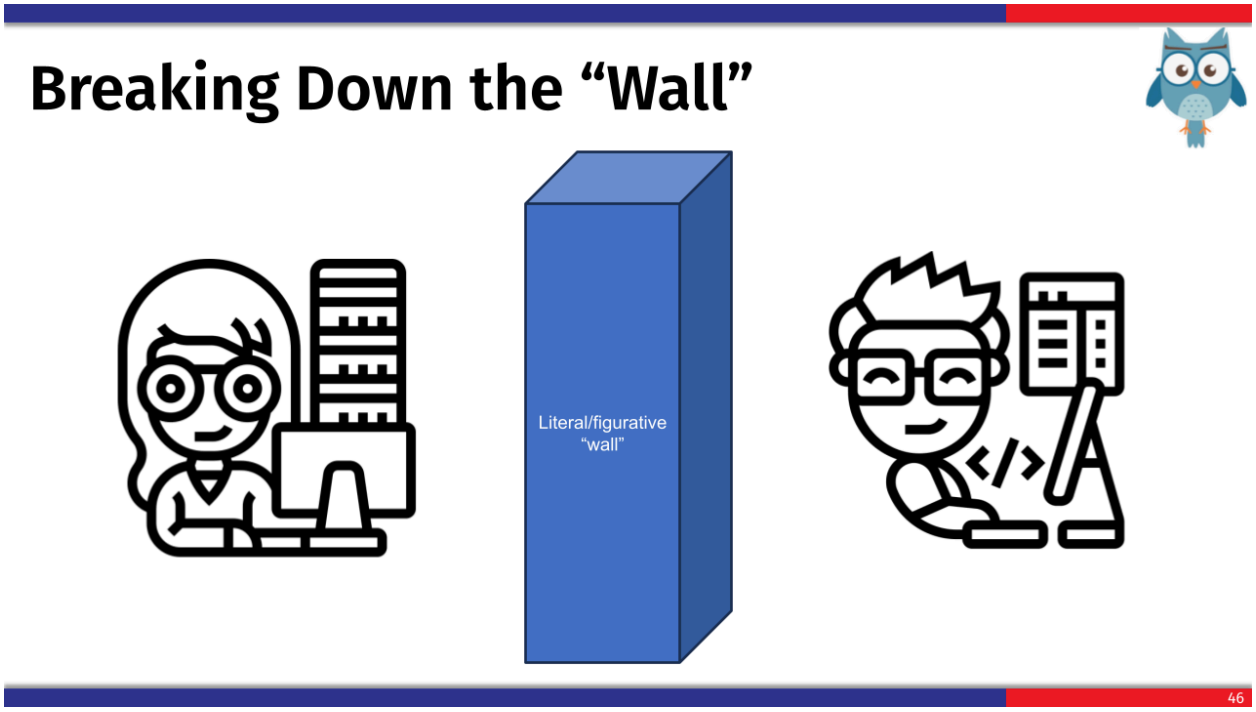
- **Version Control**: In general, we should store in source control everything we need to build anything, but nothing that we actually build. If you use Glt, you can use `.gitignore` to exclude local build/compile results from the repository.
- **Automate the Build**: Any instructions for the build need to be stored in the repository, in practice this means that we must use text representations. That way we can easily inspect them to see how they work, and crucially, see `diff`s when they change.
- **Make the Build Self-Testing**: Use automated unit testing libraries, and make sure the tests thoroughly cover your code and software requirements. (Hands on at Module 4 - TDD and Refactoring)
- **Everyone Pushes Commits To the Mainline Every Day**, **Everyone Pushes Commits To the Mainline Every Day**, and **Every Push to Mainline Should Trigger a Build**. Everyone should commit their daily work into the repository, so that any changes will be recorded, can be reviewed by other team members, and will be built right away. If there is a change that broke the build, we can **Fix Broken Builds Immediately**.
- **Keep the Build Fast**: Slow builds make impatient developers bored and give additional burden to your operational costs (in this case, CI-minute credits).
- **Hide Work-in-Progress**: Branching in Git will help separate work-in-progress code with production code, if managed properly. Although sometimes, we need to test work-in-progress code alongside the production code, for example to test performance or

usability for sampled user groups. You can use Feature Flags, Canary Releases, or other deployment methodologies to assist on that. Further explanation available in Module 11 - Deployment and Monitoring.

- **Test in a Clone of the Production Environment**: Make the staging environment as similar as possible to the production environment. This allows developers to test their builds with production-level infrastructure, without altering the production environment.
- **Everyone Can See What's Happening**: In a nutshell, Continuous Integration is all about communication. We want to ensure that everyone can easily see the state of the system and the changes that have been made to it. Enable status badges in your repository, and set up alerting webhook to your team's main communication platform (e.g. Slack/Discord).
- **Automate Deployment**: The automation should be done across multiple environments (development, staging, and production).

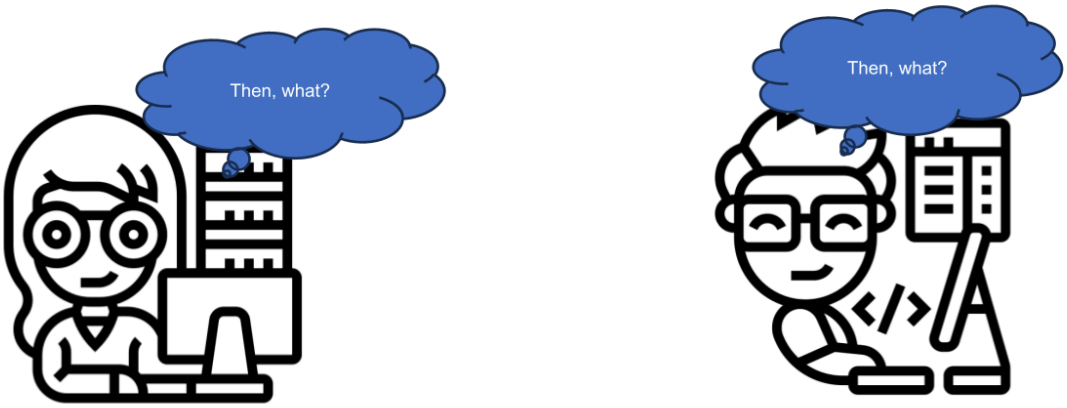
The rule of thumb pointed out by Martin Fowler: Anyone should be able to bring in a clean machine, check the sources out of the repository, issue a single command, and have a running system on their own environment.

3. DevOps



If we remember the traditional context of developing software, the Developer and IT Ops are often separated by a wall, either figuratively (i.e. different department) or literally (different office room). This separation may lead to communication overhead between roles and less empathy toward each responsibility.

Breaking Down the “Wall”



47

If we tear down the “wall” between Developers and IT Ops in order to implement DevOps, they may not necessarily be ready to collaborate closer with each other. It is not enough by simply removing the wall. Technically, we can assume both roles are competent in their technical capability. But culturally, they may need adjustments.

It is not that easy to implement DevOps, as implied by the cat figure in the slide. Sure, DevOps comprises the role Developers and IT Operations joined together. Hence, some organizations think that it is enough by simply making them work together. We will see that DevOps is more than making Developers and IT Operations working in the same room.

It is also a lesson for the software development industry. We are in a field where technical jargon comes up often, which might be proposed by some organization to further their economic interest. Some technical jargon may not necessarily be a novel concept (e.g., service-oriented architecture (SOA) from mid 2000s and microservices from mid 2010s), or get misinterpreted (e.g., DevOps). We need to be able to discern the actual meaning of a concept and decide whether it will be beneficial for us.

DevOps

*"DevOps is an organizational approach that stresses **empathy and cross-functional collaboration** within and **between teams** – especially development and IT operations – in software development organizations, in order to **operate resilient systems and accelerate delivery of changes.**" (Dyck et al., 2015)*

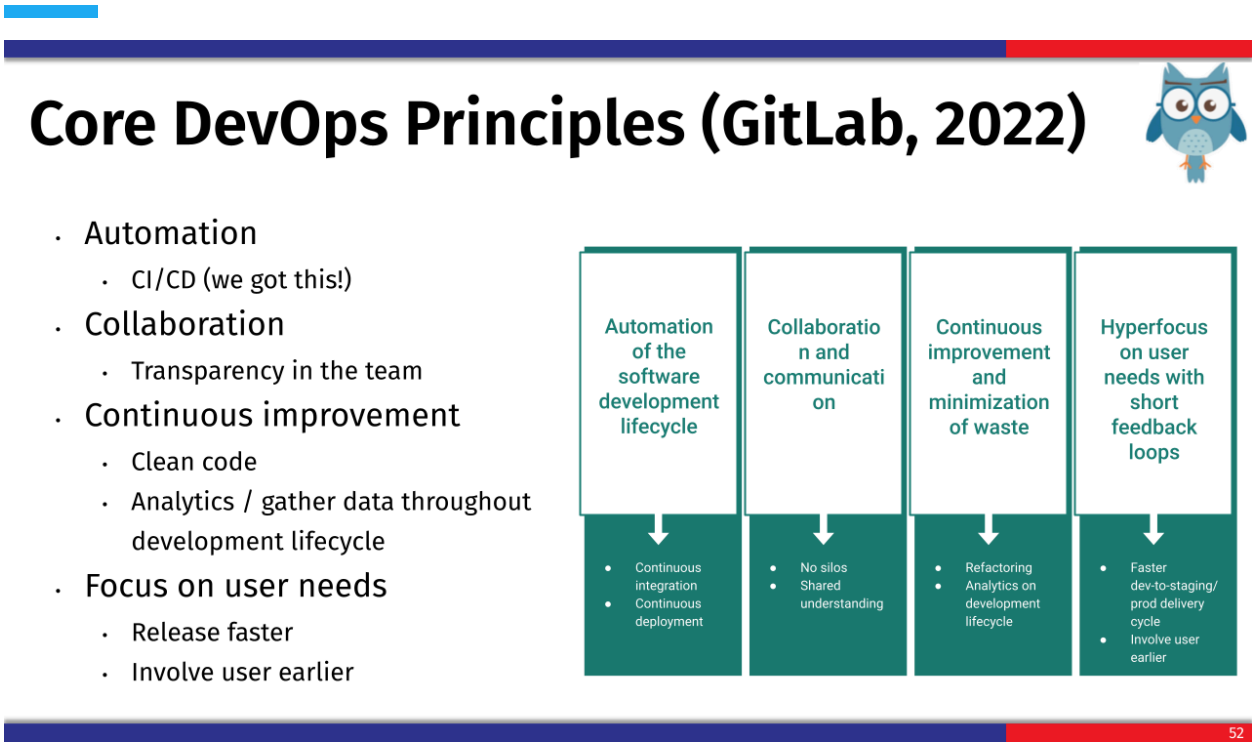
- Reduce barriers between teams
  - It is not as simple as putting Dev and Ops "in the same room"
- Require teams to have cross-functional capabilities
- Include more automation in the process to accelerate delivery/deployment
  - And doing things earlier, aka "Shift-Left"

50

Let's look at the definition provided by Dyck et al. in 2015. They defined DevOps as "an organizational approach that stresses **empathy** and **cross-functional collaboration** within and **between teams** in software development organizations, in order to **operate resilient systems and accelerate delivery of changes**."

First, DevOps emphasizes empathy and cross-functional collaboration. It means the teams need to be aware of the responsibilities of each role and know the required effort to fulfill them. If we look at the traditional roles of Developer and IT Operations, then Developers need to know the workload done by the IT Operations to complete their job, and vice versa. One way to do that is, for example, by letting Developers and IT Operations get exposed to each other's work.

The collaboration in DevOps not only happens in the team, i.e. the development team, but also with other teams in the organization. In addition, the participants are required to have "cross-functional capability" where they need to be competent in other responsibilities other than their main ones. Hence, if we look back at the traditional division of Developers and IT Operations, it is expected that Developers know how to do tasks performed by IT Operations, and vice versa. At minimum, they need to be able to do the other tasks but not necessarily become highly competent.



There are several ways of implementing DevOps in an organization. One such example is the Core DevOps Principles proposed by GitLab. They defines the four principles as follows:

1. Automation of the software development lifecycle

   DevOps implementation needs to be supported by automation. This is the main topic that we cover throughout the module. We want to make repetitive tasks automated so we can focus more on problem solving.

2. Collaboration and communication

   DevOps is not only about implementing automation tools, but also improving communication and collaboration among people in the process. We can start by breaking down the “silos”/”walls” and encourage the teams to be more empathetic.

3. Continuous improvement and minimization of waste

   As the teams implement DevOps, we also strive to improve the work quality and the process. The teams can apply good engineering practices such as clean code to ensure the quality of the codebase. Moreover, the teams can also measure the performance indicators (e.g., average time from requirement to commit, average. CI/CD workflow duration, etc.) related to the development lifecycle and use the data to identify possible improvement on the process.

4. Hyperfocus on user needs with short feedback loops

As DevOps implementation is often paired with a development process that applies agile methodology, it is important to have user involvement earlier in the software development process. The automation implemented as part of DevOps lets us deliver increments of the product faster to the customer. Consequently, we can get feedback earlier and faster from the customer.

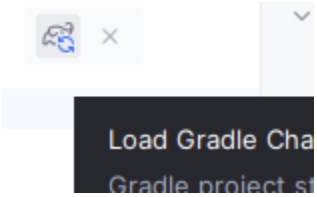
4. Tutorial & Exercise: Implementing CI/CD using Gradle & GitHub Actions

In this tutorial, you will learn how to customize Gradle build script to measure code coverage. Then, you will implement workflows on GitHub Actions to automate test suite, code quality analysis, and deployment.

1. Open your Module 1 Exercise project using IntelliJ or your favorite editor.
- Create a new branch named `ci-cd` based on the latest version of your `main`/`master` branch.
2. Open the `build.gradle.kts` file. To measure code coverage in a Java-based project, you will use a plugin named JaCoCo. Modify the Gradle build script by adding the `jacoco` plugin in the `plugins` block. The final content of the `plugins` block should be similar to the following screenshot:

```
1  plugins { this: PluginDependenciesSpecScope
2      java
3      jacoco
4      id("org.springframework.boot") version "3.2.2"
5      id("io.spring.dependency-management") version "1.1.4"
6  }
```

3. Reload the Gradle build script. If you are using IntelliJ, you can click the icon that appears each time you modify the Gradle build script.



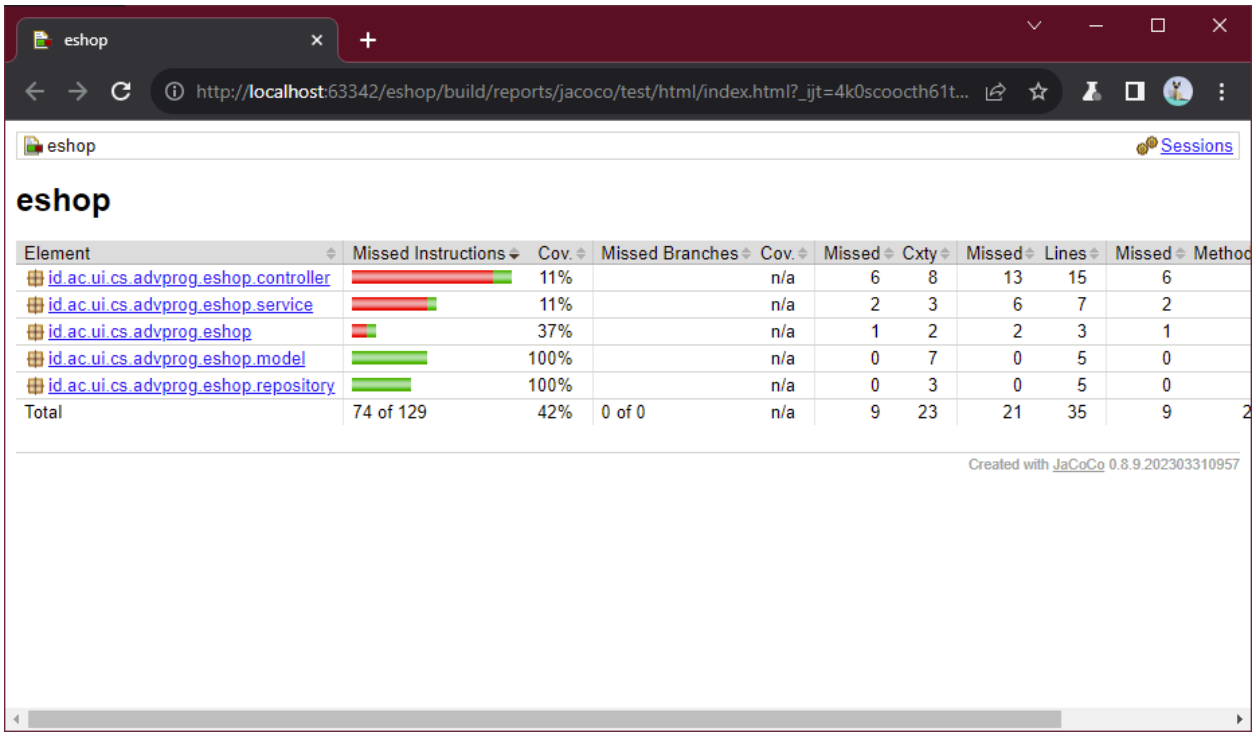
4. See the list of tasks of the project by inspecting the Gradle panel in IntelliJ or running the `./gradlew tasks` command in the terminal. Verify that there are two new tasks called `jacocoTestCoverageVerification` and `jacocoTestReport`.
5. Edit the `build.gradle.kts` to do the following:
    a. Exclude the functional tests from being run by the `test` task. The `test` task is a built-in task provided by the `java` Gradle plugin.
    b. Ensure the code coverage report generation (i.e., `jacocoTestReport`) to be always generated after running the `test` task.
    c. Tell the `jacocoTestReport` task to run after the `test` task.

The resulting `build.gradle.kts` should be similar to the following screenshot:

```
67    tasks.test { this: Test!
68        filter { this: TestFilter
69            excludeTestsMatching("*FunctionalTest")
70        }
71
72        finalizedBy(tasks.jacocoTestReport)
73    }
74
75    tasks.jacocoTestReport { this: JacocoReport!
76        dependsOn(tasks.test)
77    }
```

- Save and commit your latest version of `build.gradle.kts`

6. Reload the changes in Gradle build script and run the `test` task. It will only run the unit test suite and generate the code coverage report.
7. Open the code coverage report. It is stored at `build/reports/jacoco/test/html/index.html` by default. The output may be look similar to the following screenshot:



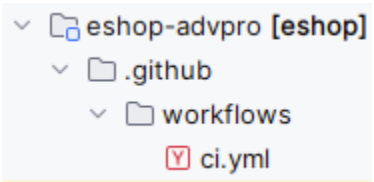
8. As you can see from the screenshot before, the total coverage of the project is currently at 42%. It means that 42% (74 instructions out of 129 instructions) of the instructions in the codebase are not yet executed by the test suite. Your total coverage may be different.
9. Try to open one of the Java source code files listed in the report page. You will see that some of the instructions are highlighted with red and green colors.

ProductServiceImpl.java

```java
package id.ac.ui.cs.advprog.eshop.service;

import id.ac.ui.cs.advprog.eshop.model.Product;
import id.ac.ui.cs.advprog.eshop.repository.ProductRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

@Service
public class ProductServiceImpl implements ProductService {

    @Autowired
    private ProductRepository productRepository;

    @Override
    public Product create(Product product) {
        productRepository.create(product);
        return product;
    }

    @Override
    public List<Product> findAll() {
        Iterator<Product> productIterator = productRepository.findAll();
        List<Product> allProduct = new ArrayList<>();
        productIterator.forEachRemaining(allProduct::add);
        return allProduct;
    }
}
```

The red highlighted ones indicate that the instructions were not executed by the test suite. In contrast, the green highlighted ones indicate that the instructions were executed by the test suite.

10. Now let's try to implement a CI/CD on GitHub using GitHub Actions. Create a new workflow named `ci.yml` at `.github/workflows` directory.



11. Open the `ci.yml` and implement the workflow similar to the following example:

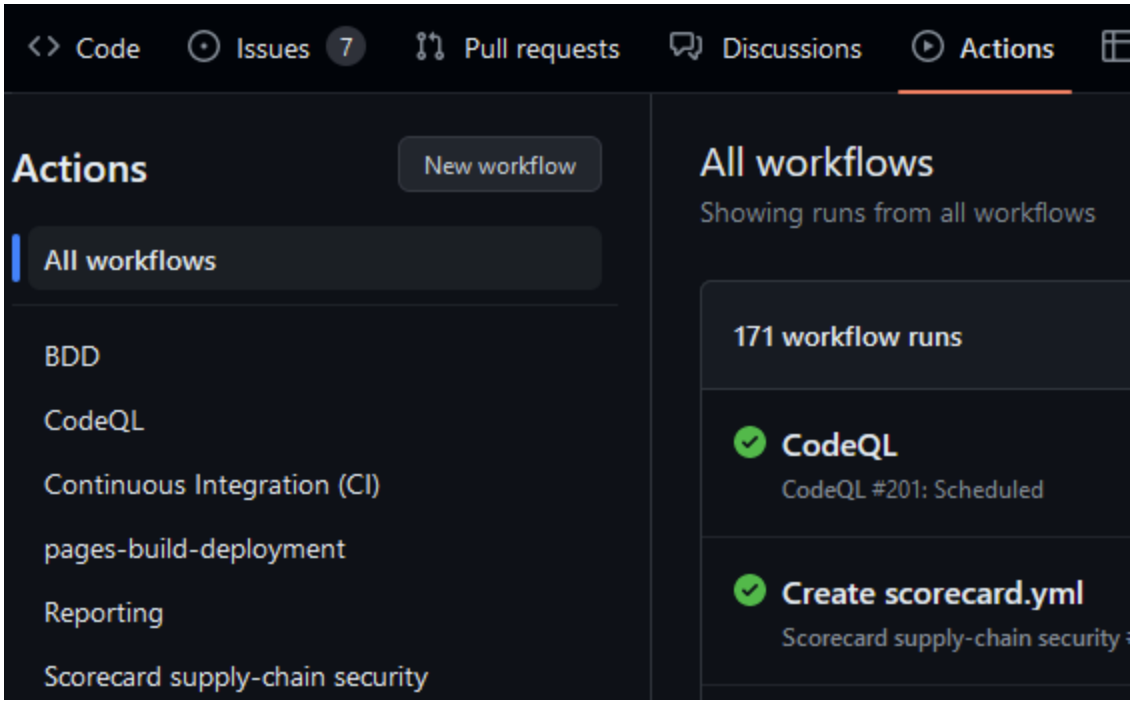
```yaml
name: Continuous Integration (CI)

## Run CI jobs on all branches by default
on:
  push:
  pull_request:

jobs:
  test:
    name: Run tests
    runs-on: ubuntu-22.04
    steps:
      - name: Check out the Git repository
        uses: actions/checkout@v4
      - name: Set up Java toolchain
        uses: actions/setup-java@v4
        with:
          distribution: "temurin"
          java-version: "21"
          cache: "gradle"
      - name: Run unit tests
        run: ./gradlew test
```

Explanation:

- The name of the workflow is set to “Continuous Integration (CI)”
- The workflow will be run for every `push` and `pull_request` events on the repository.
- The workflow contains one job, `test`, that runs on a Linux-based server using Ubuntu distribution.
- The `test` job will run three steps: `checkout` action, `setup-java` action, and `./gradlew test` shell command on the runner.

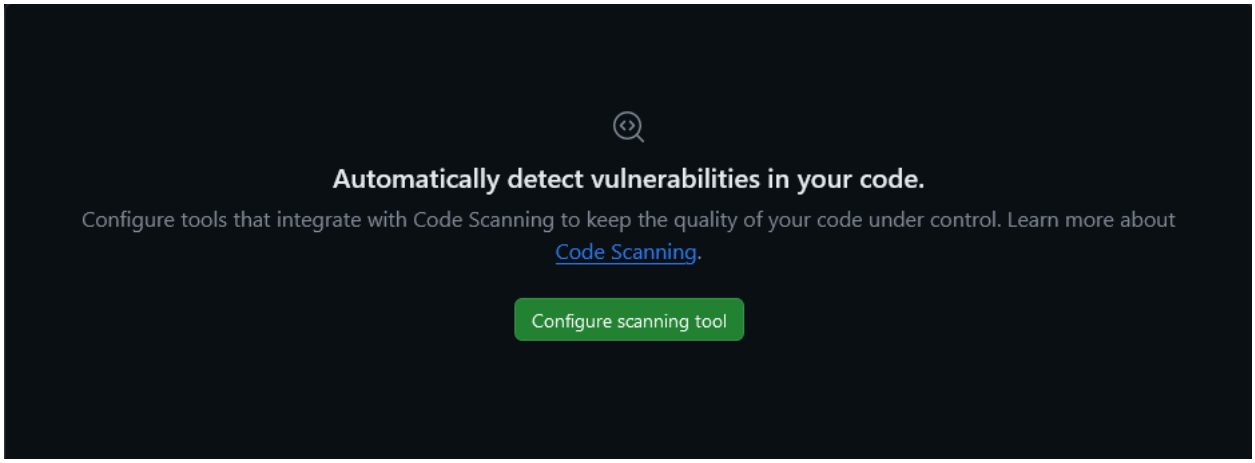
- Save and commit the `ci.yml`

12. Push the branch to your online Git repository repository on GitHub to trigger the CI workflow. Wait for a moment (at least 1 - 2 minutes). You can monitor the CI workflow run status by looking at the Actions page. The following screenshot depicts an example of Actions page of an open-source Java-based project on GitHub:

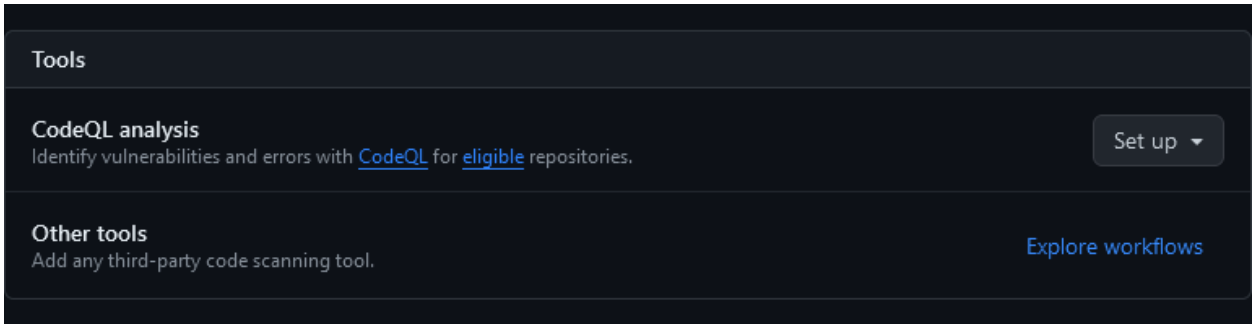


13. In addition to running the test suite, we can also run code scanning tools in the CI/CD. One example that we will try is the OSSF Scorecard that was mentioned in the previous module.

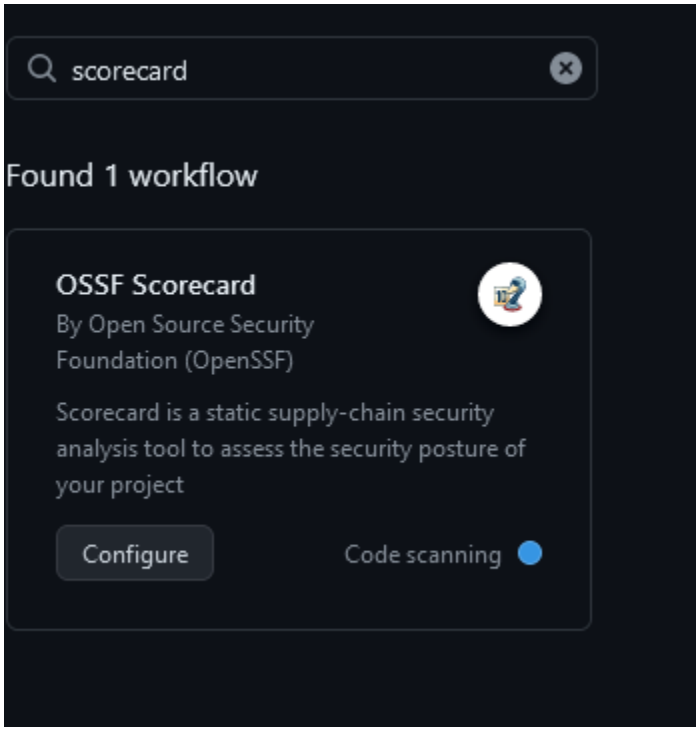
14. Go to the Security Overview page by clicking on the Security tab of your GitHub repository. Then, if you have not configured any code scanning before on GitHub, click the "Code scanning" menu at the left-side panel of the page and choose "Configure scanning tool".



15. You will be redirected to the "Code security & analysis" page. In the "Code scanning" section, pick "Explore workflows".



16. In the “Choose a workflow” page, search for “scorecard” and pick the OSSF Scorecard from the search results by clicking the “Configure” button.

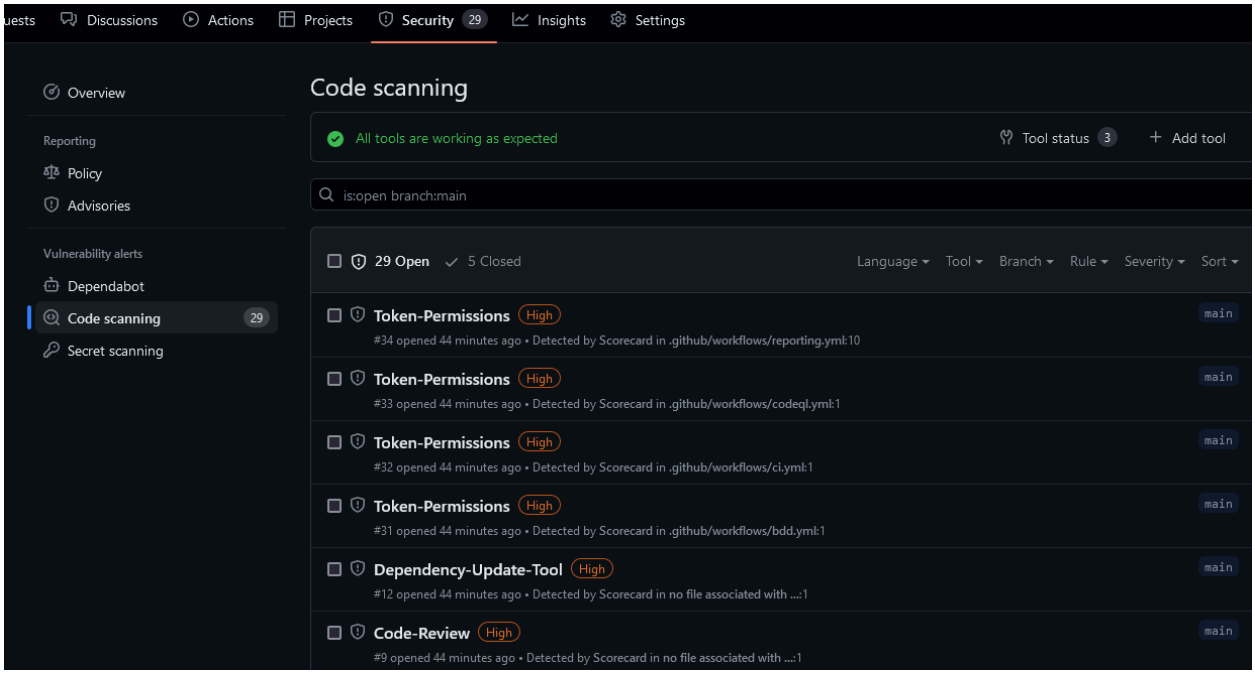


17. GitHub will suggest a change containing a new workflow to run OSSF Scorecard. Make sure the proposed change is targeted to the `main`/`master` branch of your codebase. You can commit the proposed change via the Web interface on GitHub. Alternatively, you can also write the proposed change locally. Then, save the proposed change as a new commit and push it to your online Git repository.

- Commit the workflow for running the OSSF Scorecard and push it to `cicd` branch

18. Wait for a moment. You can monitor the workflow run status at the Actions page.

19. If the OSSF Scorecard detected any issues, the result will be available for perusal at the Security Overview page. One example of the OSSF Scorecard output can be seen in the following screenshot:



a. If you host your online Git repository on GitLab or GitLab CSUI, you can run the OSSF Scorecard program (i.e., `scorecard` command) on GitLab CI/CD. Make sure the output is displayed as plain string or JSON format to the standard output stream. Refer to the following example provided by OSSF in their GitLab's repository: https://gitlab.com/ossf-test/scorecard-pipeline-example

20. Merge the `cicd` branch to your `main`/`master` branch on your online Git repository.

4.1 Exercise

- If you haven't done so, please merge the `cicd` branch to your `main`/`master` branch.
- Create a new branch named `module-2-exercise` based on the latest commit on the `main`/`master` branch.
- If the code coverage of your Module 1 Exercise is not 100% yet, try to improve the code coverage of your project. You can do so by adding a new unit test into the project. Then, re-run the test suite and JaCoCo test reporting.
- If you added a new test case, don't forget to commit it into the `module-2-exercise` branch.
- Take note of the code coverage improvement after adding the new test case.
- Add another code scanning/analysis tool into your CI/CD process. Some alternatives that you can use:
    a. SonarCloud (https://sonarcloud.io) ➔ one of the most popular code quality scanners; free to use for open-source projects hosted on GitHub/GitLab.

b. SonarQube CSUI (https://sonarqube.cs.ui.ac.id) ➔ self-hosted, open-source version of SonarCloud.
c. PMD (https://github.com/marketplace/actions/pmd) ➔ can be run locally using command-line tool or Gradle plugin
    i. Make sure you set the PMD version to 7.0.0-rc4. The current stable version (version 6.x) mentioned in the action's documentation does not support Java 21 yet.

We recommend you to use either **SonarCloud** or **PMD**.

- Create a new workflow that triggers the new code scanning/analysis on every `push` events to every branch. Don't forget to save and commit the new workflow file into the `module-2-exercise` branch.
- Run the workflows related to code scanning/analysis and take note of the detected code quality issues.
- Try to address at least one code quality issue by fixing the codebase. Save and commit the fix into the `module-2-exercise` branch. You may fix more than one issue. Make sure each fix is saved as independent commits from other fixes.
- Verify that the fixed issue(s) does not come up in the next workflow run.
- Implement an auto-deploy mechanism to a PaaS in your online Git repository using either push- or pull-based approach. Push-based approach can be implemented by using a workflow that "pushes" your codebase to the deployment environment, whereas pull-based approach lets the deployment environment to "pull" the code base into the environment. Usually PaaS that follow a pull-based approach require the codebase to be associated with the PaaS and will monitor changes of your codebase remotely.

  Several free PaaS that you can use:

  1. Koyeb ➔ Either push- or pull-based approach. See the documentation at: https://www.koyeb.com/docs/deploy/java
  2. Render ➔ Pull-based approach. See the documentation at: https://docs.render.com/github and https://docs.render.com/docker#docker-vs-native-runtimes

  To help you package the application as a Docker image, which may be required in some PaaS, you can refer to the following `Dockerfile` for building Spring Boot application into Docker image:

```dockerfile
FROM docker.io/library/eclipse-temurin:21-jdk-alpine AS builder

WORKDIR /src/advshop
COPY . .
RUN ./gradlew clean bootJar

FROM docker.io/library/eclipse-temurin:21-jre-alpine AS runner

ARG USER_NAME=advshop
ARG USER_UID=1000
ARG USER_GID=${USER_UID}

RUN addgroup -g ${USER_GID} ${USER_NAME} \
    && adduser -h /opt/advshop -D -u ${USER_UID} -G ${USER_NAME} ${USER_NAME}

USER ${USER_NAME}
WORKDIR /opt/advshop
COPY --from=builder --chown=${USER_UID}:${USER_GID} /src/advshop/build/libs/*.jar app.jar

EXPOSE 8080

ENTRYPOINT ["java"]
CMD ["-jar", "app.jar"]
```

- Write the new workflow to deploy the codebase to a PaaS of your choice. Don't forget to save and commit it to `module-2-exercise`.
- Merge the `module-2-exercise` branch to the `main`/`master` branch.
- Let the workflows run and wait for deployment to be finished.
- Take note of the URL where your app is deployed.
- *(Optional/Challenge)* Try to add new test cases to increase your code coverage to 100%. Make sure the new test cases are **meaningful**. That is, the new test cases verify the correctness of the codebase.
    a. Hint: You need to explore topics beyond the coverage of this module, especially if you want to test the controller(s) of your Web application as unit tests. To give you some pointers, try to learn about *testing using mock*.

4.2 Reflection

You have implemented a CI/CD process that automatically runs the test suites, analyzes code quality, and deploys to a PaaS. Try to answer the following questions in order to reflect on your attempt completing the tutorial and exercise.

1. List the code quality issue(s) that you fixed during the exercise and explain your strategy on fixing them.
2. Look at your CI/CD workflows (GitHub)/pipelines (GitLab). Do you think the current implementation has met the definition of Continuous Integration and Continuous Deployment? Explain the reasons (minimum 3 sentences)!

Write your reflection in the `README.md` file of your repository.

4.3 Grading Rubric

Grading Scheme

- Students should follow the instructions in the given module.
- Do the tutorial and exercise in the module.
- Make sure to do all the commits as requested.
- Do not squash your commit during the merge process. It may destroy your Git commits history and the teaching assistants cannot evaluate it.
- Write down your personal reflection notes as markdown in the readme.md on your main/master branch.
- Push your work to the repository.
- Give access to the teaching assistants.
- Submit the link to your repository on SCELE.

Scale

All components will be scored on discrete scale 0,1,2,3,4. Grade 4 is the highest grade, equivalent to A. No Partial score Score 0 is for not submitting.

Components

- 60% - Commits
- 10% - Reflection
- 30% - Correctness
- Bonus 10% - 100% code coverage

Rubrics

	Score 4	**Score 3**	**Score 2**	**Score 1**
Correctness (one time final result)	Score 3 criteria + can access the deployed app via the public Internet AND fixes 75% of code quality issues	All working CI/CD workflows/pipelines AND there is an attempt to deploy the app AND fixed at least one code quality issue	Partially working CI/CD workflows/pipelines AND (no proof of deployment OR no proof of fixing code quality issue)	Broken CI/CD workflows/pipelines AND no proof of deployment AND no proof of fixing code quality issue
Reflection (for each reflection)	More than 5 sentences. The description is	Less than or equal 5 sentences. The	The description is not sound although still	The description is not sound. It is not related to the

	sound.	description is sound.	related.	topics.
Commit (evaluated on all commits)	All requested commits are completely registered and correct.	At least 75% of the requested commits are registered and correct.	At least 50% of the requested commits are registered and correct.	Less than 50%.
Bonus (100% Code Coverage)	Score 3 criteria + achieved 100% code coverage	Achieved 99% - 90% code coverage AND all tests contain verification/assertion to meaningful expected output.	Achieved < 90% code coverage OR some tests do not contain any verification/assertion to a meaningful expected output.	Attempted to add new test case(s) but the code coverage did not increase.