# CT417

## Software Engineering III

Name: Andrew Hayes
E-mail: a.hayes18@universityofgalway.ie
Student ID: 21321503

2024–10–23

# Contents

**1 Introduction** **1**
1.1 Lecturer Contact Details . . . 1
1.2 Grading . . . 1

**2 Revision** **1**
2.1 What is Software? . . . 1
2.2 Functional vs Non-Functional Requirements . . . 1
2.3 What is Software Engineering? . . . 2
2.4 What are Software Development Life Cycles? . . . 2
2.5 What is a Framework? . . . 2
2.6 Agile & DevOps . . . 2
2.6.1 What is Agile? . . . 2
2.6.2 Agile Principles . . . 2
2.6.3 Agile Frameworks . . . 2
2.6.4 What is DevOps? . . . 3
2.6.5 DevOps Core Practices . . . 3
2.6.6 Key Differences between Agile & DevOps . . . 3
2.6.7 Why DevOps Complements Agile . . . 3
2.6.8 Benefits of Agile & DevOps . . . 3

**3 Version Control** **4**
3.1 What is Version Control? . . . 4
3.2 What files should be checked in to Version Control? . . . 4
3.3 Centralised Version Control – Subversion . . . 4
3.4 Distributed Version Control – Git . . . 7
3.4.1 GitHub . . . 7
3.4.2 Git Commands . . . 7
3.4.3 Pull Requests . . . 7

**4 Build Tools** **8**
4.1 Build Tools in CI/CD Pipelines . . . 9
4.2 Maven . . . 9
4.2.1 Plugins . . . 10
4.2.2 Build Lifecycle . . . 11
4.2.3 Dependency Management . . . 12
4.2.4 Local & Remote Repository . . . 12
4.3 Spring Boot . . . 12
4.3.1 Spring vs Spring Boot . . . 12
4.4 GitHub Actions . . . 13

**5 Containerisation** **13**
5.1 Docker . . . 14
5.1.1 Building & Running Applications in Docker Containers . . . 16
5.1.2 Docker Best Practices . . . 16
5.1.3 Common Docker Commands . . . 17
5.1.4 Practical Docker Example with Spring Boot . . . 17
5.2 Container Orchestration . . . 19
5.2.1 Kubernetes . . . 20

**6 DevSecOps** **21**
6.1 Security Vulnerabilities in Code . . . . . 23
6.1.1 SQL Injection . . . . . 23
6.1.2 Cross-Site Scripting . . . . . 24
6.1.3 Cross-Site Request Forgery . . . . . 24
6.2 Static Code Analysis . . . . . 24

**7 Software Security** **26**
7.1 Threat Classification . . . . . 27
7.1.1 Requirement-Level Threats . . . . . 27
7.1.2 Hardware-Level Threats . . . . . 27
7.1.3 Code-Level Threats . . . . . 27
7.1.4 Design-Level Threats . . . . . 28
7.1.5 Architectural-Level Design Threats . . . . . 28

# 1 Introduction

## 1.1 Lecturer Contact Details

- Dr. Effirul Ramlan.
- Email: `effirul.ramlan@universityofgalway.ie`.
- Will attempt to reply to emails immediately between the hours of 09:00 & 20:00 from Week 01 to Week 12.
- Discord server: `https://discord.gg/CRAtHv9uNg`.

## 1.2 Grading

- Continuous Assessment: 40%.
  - You will work in pairs on a software project with three key submissions across the 12 weeks. Each deliverable will align with the topics covered in the course up to that point, allowing for continuous progress assessment.
  - AS-01: Set up musicFinder and configure the CI/CD pipeline (Week 4).
  - AS-02: Testing, Security, & Expanded Application (Week 8).
  - AS-03: Refactoring & Application Deployment.
- Final Exam: 60%.
  - Typical 2-hour exam paper covering materials from Week 1 to Week 12, with nothing out of the ordinary – "You can be sure of that".
  - "There is a question on Agile on your final" – key differences between agile & DevOps table.

# 2 Revision

## 2.1 What is Software?

**Software** consists of:

i. Instruction (computer programs) that when executed provide desired features, function, & performance.

ii. Data structures (Arrays, Objects, Lists, Dictionaries, Maps, etc.) that enable programs to manipulate information.

iii. Descriptive information in both hard copy & virtual format describing the operation & use.

## 2.2 Functional vs Non-Functional Requirements

| **Functional Requirement** | **Non-Functional Requirement** |
|---|---|
| Describes the actions with which the user's work is concerned | Describes the experience of the user while doing the work |
| A feature or function that can be captured in use-cases | A global constraint (and therefore difficult to capture in use-cases) |
| A behaviour that can be analysed via sequence diagrams or state machines | A software quality |
| can be usually traced back to a single module / class / function | Usually cannot be implemented in a single module or even program |

Table 1: Functional vs Non-Functional Requirements

Typical non-functional requirements include: availability, maintainability, performance, privacy, reliablility, scalability, & security.

## 2.3 What is Software Engineering?

**Software Engineering** is the field of computer science that deals with the building of software systems that are so large or so complex that they are built by a team or teams of engineers. Software Engineering encompasses a process, a collection of methods, & an array of tools that allow professionals to build high-quality software.

**DevOps** outlines a software development process that increases the delivery of higher quality software by integrating the efforts of the development & IT operation teams.

$$\text{DevOps} = \text{Software Engineering} + \text{IT Operations}$$

## 2.4 What are Software Development Life Cycles?

**Software Development Life Cycles (SDLC)** refers to a process used by software engineers to design, develop, & test software. Each approach focuses on a different aspect of development, from planning to continuous improvement.

## 2.5 What is a Framework?

A **software framework** is an abstraction in which common code providing generic functionality can be selectively overridden or specialised by user providing specific functionality.

**Low-code** is a method of designing & developing applications using an intuitive GUI & embedded functionality that reduce traditional professional code writing requirements. **No-code** is similar to low-code, but for non-technical business users as it allows them to develop software / applications without having to write a single line of code.

## 2.6 Agile & DevOps

### 2.6.1 What is Agile?

**Agile** is a method of software development consisting of:

- **Iterative & Incremental Development:** Software is developed in small, workable increments.
- **Customer-Centric:** Constant feedback from customers to refine requirements.
- **Frequent Delivery:** Rapid releases of smaller, functional product versions.
- **Adaptability:** Agile responds to change quickly

### 2.6.2 Agile Principles

- **Individuals & Interactions:** over processes & tools.
- **Working Software:** over comprehensive documentation.
- **Customer Collaboration:** over contract negotiation.
- **Responding to Change:** over following a plan.
- **Quote:** "The highest priority is to satisfy the customer through early & continuous delivery of valuable software."

### 2.6.3 Agile Frameworks

Agile methodologies & frameworks include:

- **Scrum:** Divides work into sprints (2-4 weeks) with regular stand-ups & reviews.
- **Kanban:** Focuses on visualising workflow & limiting Work-In-Progress (WIP).
- **XP (eXtreme Programming):** Emphasises technical excellence & frequent releases.
- **Lean Development:** Focuses on minimising waste & maximising value.

### 2.6.4 What is DevOps?

**DevOps** is a culture & set of practices that integrated development (Dev) & operations (Ops). It involves collaboration & automation between developers & IT operations for faster delivery of high-quality software. It also involves continuous integration/continuous delivery (CI/CD) to automate code testing & deployment.

$$\text{DevOps} = \text{Development} + \text{Operations}$$

### 2.6.5 DevOps Core Practices

DevOps core practices include:

- **CI/CD Pipelines:** Automating the building, testing, & deployment of code.
- **Infrastructure as Code (IaC):** Managing infrastructure through code (e.g., Terraform, Ansible).
- **Monitoring & Logging:** Ensures system reliability through real-time tracking & analysis.
- **Collaboration & Communication:** Cross-functional teams sharing ownership of development & operations tasks.

### 2.6.6 Key Differences between Agile & DevOps

| **Agile** | **DevOps** |
|---|---|
| Focus on frequent customer feedback | Focus on collaboration between Dev & Ops teams |
| Iteration done through iterative cycles | Iteration done through rapid feedback loops |
| Scope of smaller, incremental changes | Scope of large-scale projects |
| Uses task management software (e.g. Jira) | Uses automation tools (e.g. Jenkins) |
| Scrum, XP frameworks | Kanban, DevOps lifecycle frameworks |

Table 2: Key Differences between Agile & DevOps

**Agile** focuses on iterative development & customer feedback, with a short feedback loop. **DevOps** focuses on automating delivery, collaboration, & integration between Dev & Ops teams, integrating the entire process for faster releases.

### 2.6.7 Why DevOps Complements Agile

Agile improvements development velocity, but DevOps extends the concept to deployment & maintenance. Both are customer-focused, but DevOps ensures rapid & reliable deployment in addition to development. DevOps fills gaps Agile doesn't cover, like operations, infrastructure, & automation. Agile helps development teams iterate & adapt to changing requirements, while DevOps bridges the gap between developers & IT operations.

### 2.6.8 Benefits of Agile & DevOps

- Faster, more frequent delivery of features.
- Improved communication & collaboration between teams.
- Reduced risk of deployment errors.
- Ability to adapt to customer feedback & market changes rapidly.
- Higher-quality software & reduced time-to-market.

# 3 Version Control

## 3.1 What is Version Control?

**Version Control** is a system that records changes to a file or set of files over time, allowing you to recall or access specific versions at a later date. It is also known as *revision control* or *source control*. It allows you to keep track of changes, by whom, & when they occurred. Some of the popular version control programs include Git, CVS, Subversion, Team Foundation Server, & Mercurial.

It allows us to:

- Backup the source code and be able to rollback to a previous version.
- Keep a record of who did what and when (know who to praise & who to fire).
- Collaborate with the team (know who to praise & who to fire).
- Troubleshoot issues by analysing the change history to figure out what caused the problem.
- Analyse statistics such as who is being the most productive etc.

## 3.2 What files should be checked in to Version Control?

Any file that influences the software build should be checked into version control. This includes configuration files, file encodings, binary settings, etc. Furthermore, anything that is needed to setup the project from a clean checkout / fork should also be included in the version control, such as source code, documentation, manuals, image files, datasets, etc.

You should not check in any binary files such as JAR files or any other "build" files, any intermediate files from build / compilation such as `.pyc` or `.o` files, any files which contain an absolute path, or personal preference / personal settings files.

## 3.3 Centralised Version Control – Subversion

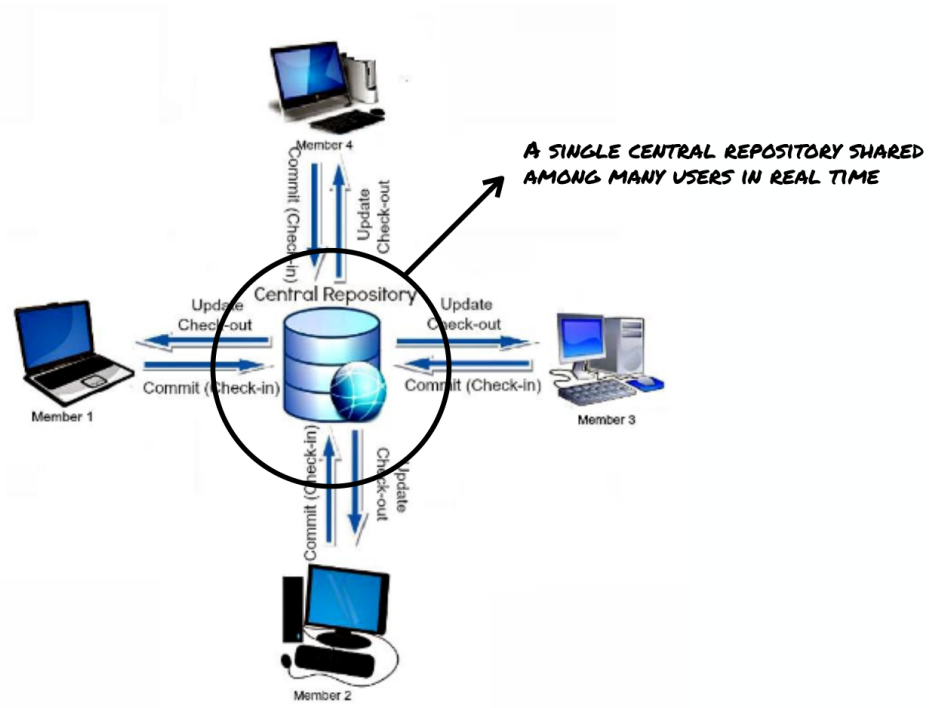


Figure 1: Centralised Version Control System Diagram

**Subversion** is a centralised version control system in which code is centralised in a repository which can be checked out to get a working copy on your local machine. In general, you don't have the entire repository checked out in Subversion, just a specific branch. Changes are committed back to the central repository, "normally" with useful comments, and a change log is maintained of who did what & when.

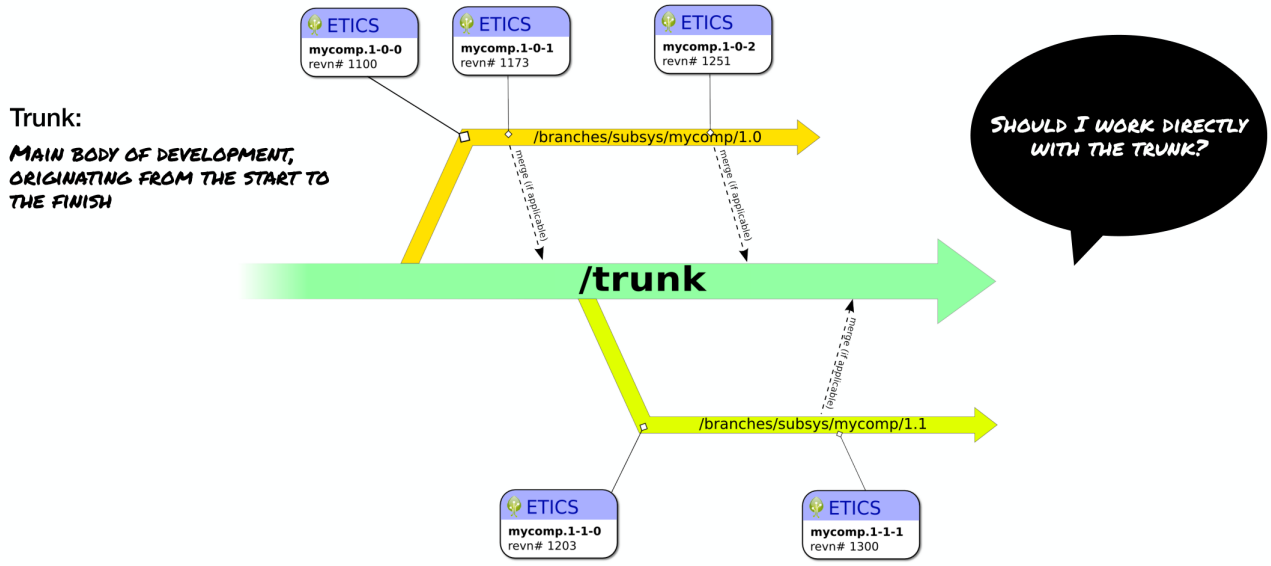


Figure 2: Trunk in Subversion

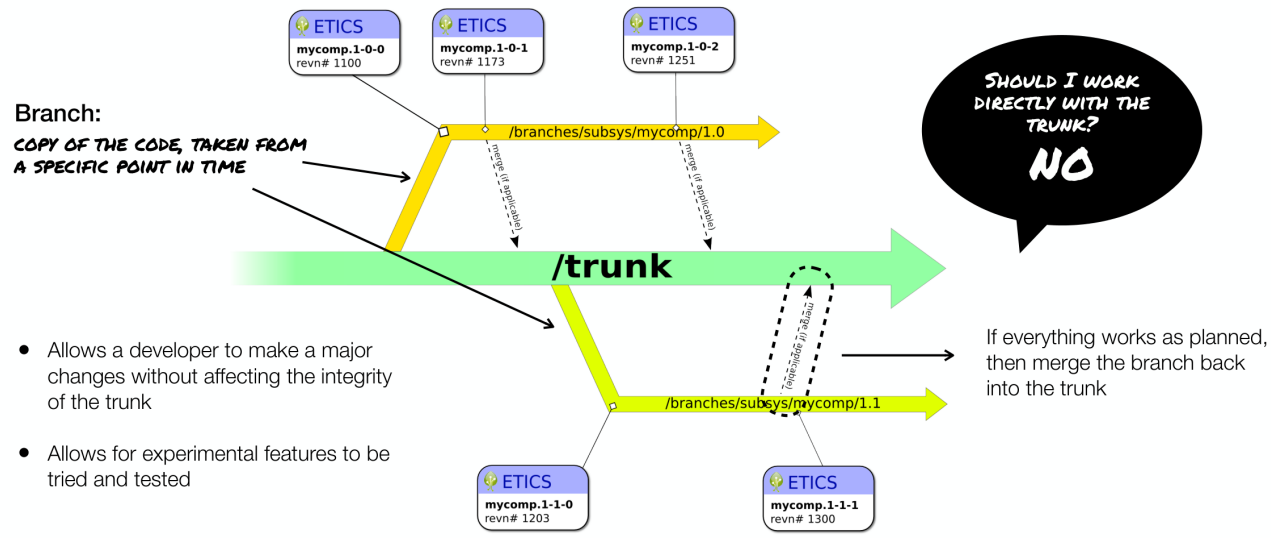


Figure 3: Branching in Subversion

When you check out the project in subversion, you will get the `HEAD` revision. When you invoke the command `svn update`, you are updating your local copy to the `HEAD` version as well. Branches should eventually be merged back into the trunk with `svn commit`. The trunk must build afterwards. The commit is a process of storing changes from your private workplace to the central server. After the commit, your changes are made available to the rest of the team; other developers can retrieve these changes by updating their working copy. Committing is an **atomic operation**: either the whole commit succeeds, or it is entirely rolled back – users never see a half-finished commit.

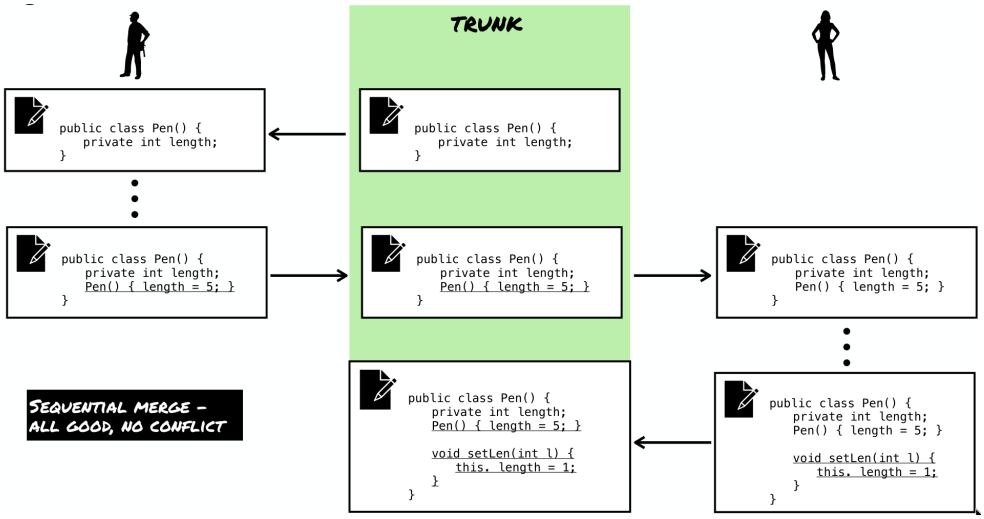


Figure 4: Sequential Merge in Subversion

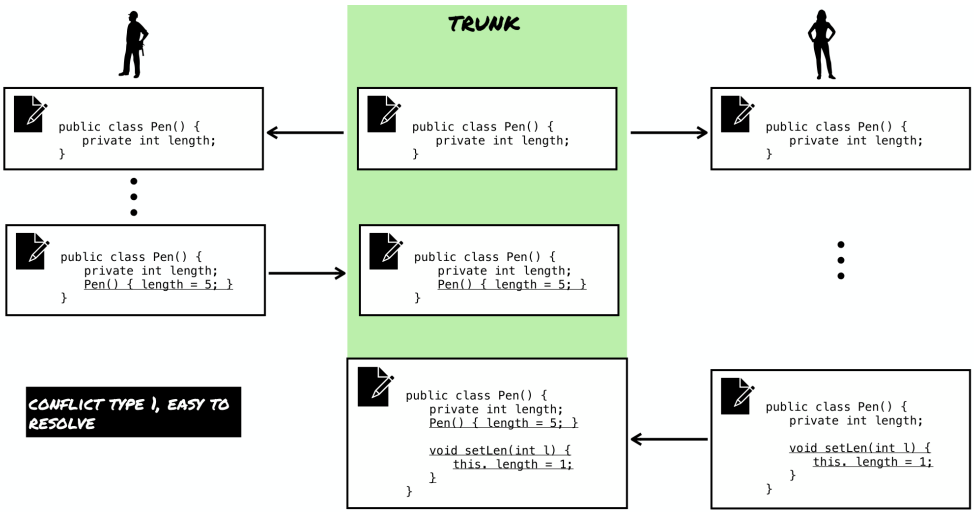


Figure 5: Type 1 Merge Conflict in Subversion

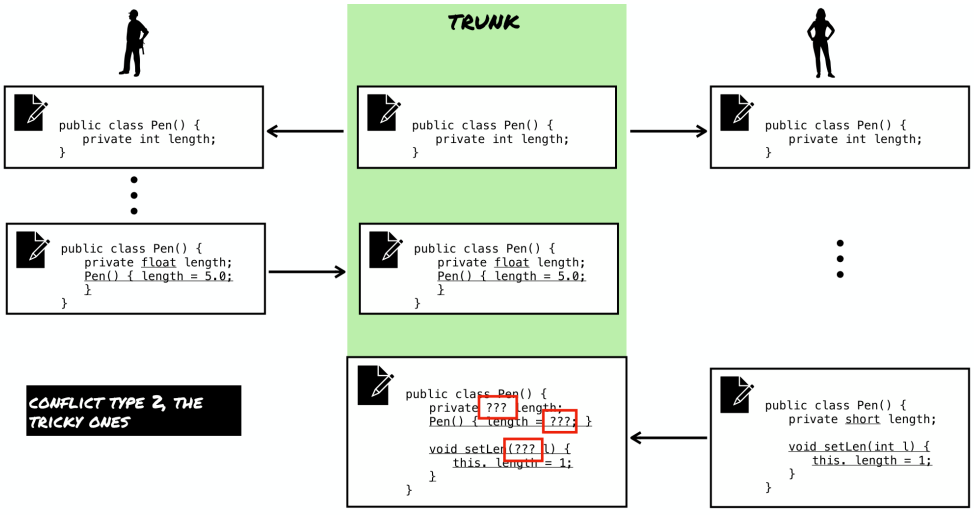


Figure 6: Type 2 Merge Conflict in Subversion

## 3.4 Distributed Version Control – Git

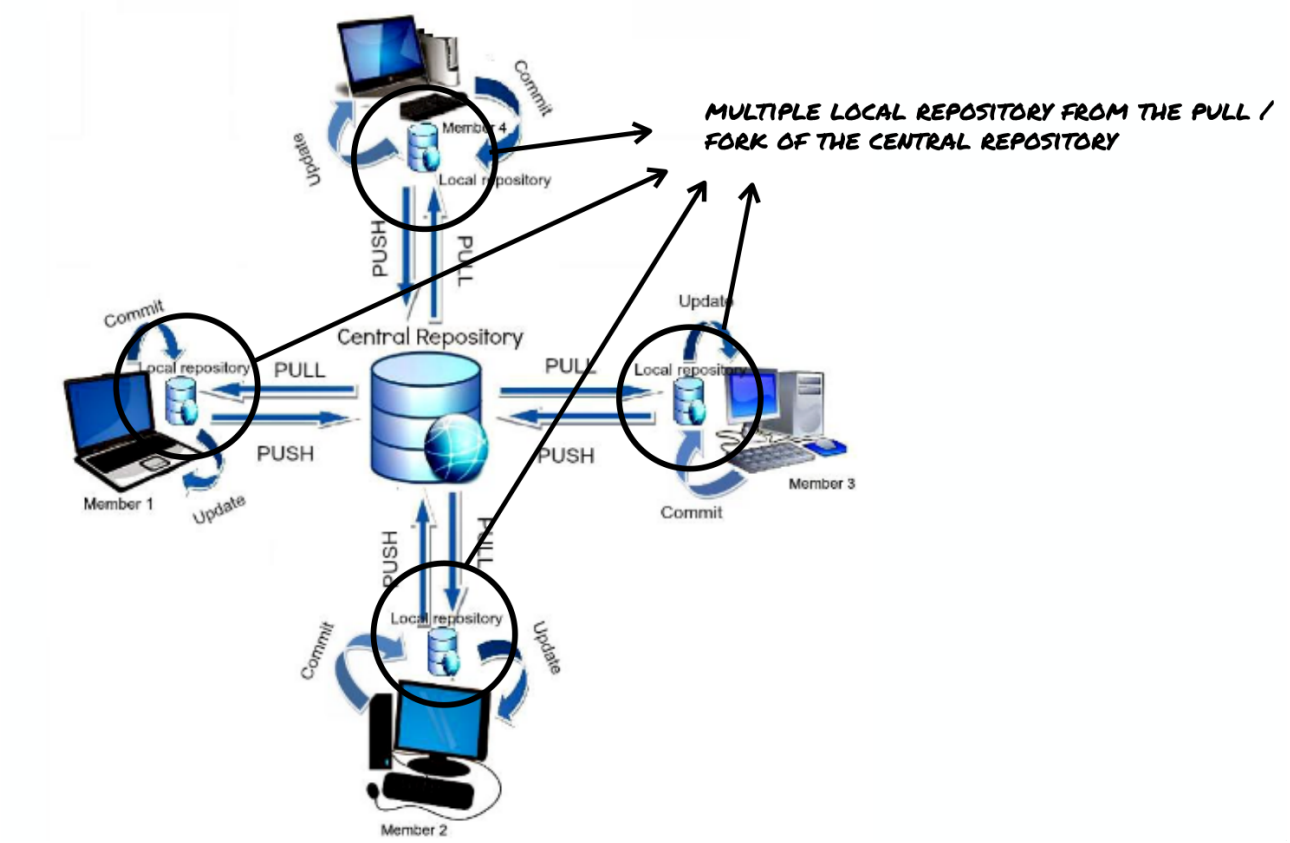


Figure 7: Distributed Version Control System Diagram

Git encourages branching for every feature, regardless of size. After successful completion of the new feature, the branch is merged into the trunk. Each developer gets their own local repository, meaning that developers don't need a network connection to commit changes, inspect previous version, or compare `diffs`. If the production trunk / branch is broken, developers can continue working uninhibited.

### 3.4.1 GitHub

**GitHub** is a web-based hosted service for Git repositories. It allows you to host remote Git repositories and has a wealth of community-based services that makes it ideal for open-source projects. It is a publishing tool, a version control system, & a collaboration platform.

### 3.4.2 Git Commands

- `git clone`: download ("clone") the source code from the remote repository.
- `git fetch`: fetches the latest version from the repository that you've cloned but doesn't synchronise with all commits in the repository.
- `git pull`: pulls the latest version from the repository that you've cloned and synchronises with all commits in the repository. Equivalent to running `git fetch` & `git merge`.
- `git push`: pushes the changes that you have committed to your local branch to the remote repository.

### 3.4.3 Pull Requests

A **pull request** is when you ask another developer to merge your feature branch into their repository. Everyone can review the code & decide whether or not it should be included in the master branch. The pull request is an invitation to discuss pulling your code into the master branch, i.e. it is a forum for discussing changes.

# 4 Build Tools

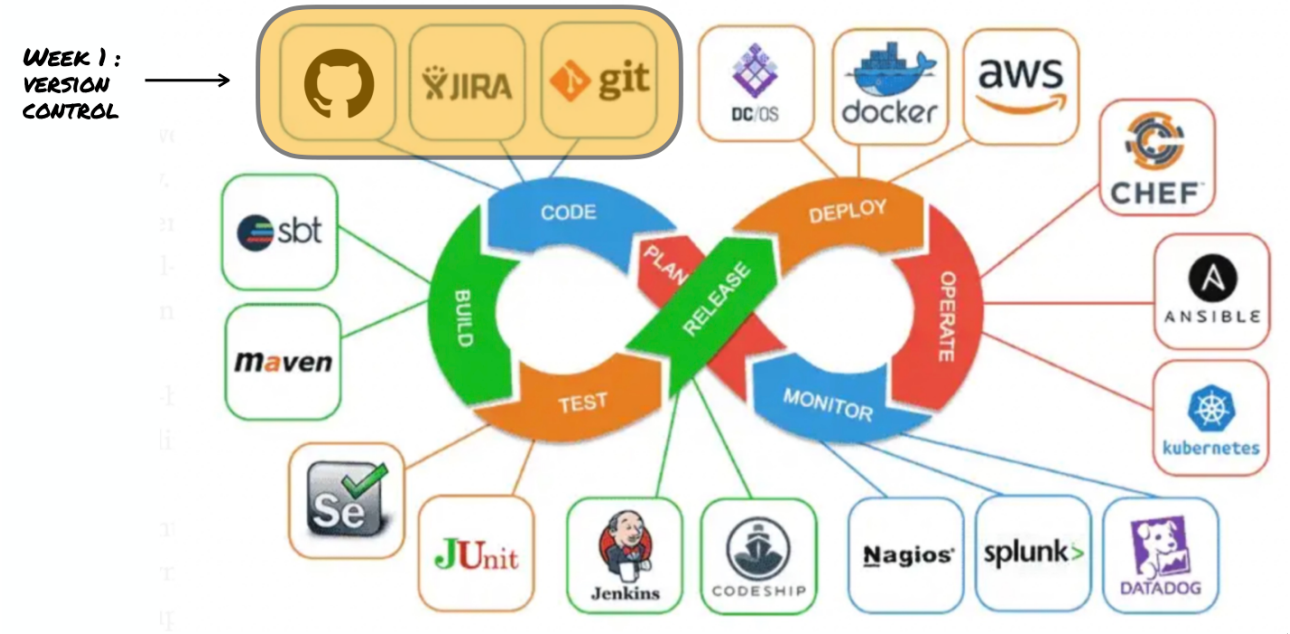


Figure 8: Example of a Continuous Software Development System

The **build** is a process which covers all the steps required to create a deliverable of your software. E.g., in Java:

1. Generating sources.
2. Compiling sources.
3. Compiling test sources.
4. Executing tests.
5. Packaging (`.JAR`, `.WAR`, `EJB`).
6. Running health checks.
7. Generating reports.

**Build tools** are software utilities that automate the tasks of compiling, linking, & packaging source code into executable application or libraries. Build tools help with:

- **Automation:** build tools help to automate repetitive tasks such as compiling code, running tests, packaging binaries, & even deploying applications.
- **Consistency:** build tools ensure that every build process (e.g., dev, test, prod) is identical, minimising human error.
- **Efficiency:** build tools speed up development by automating builds whenever code is pushed or merged into a repository.

Popular build tools include:

- **Maven** is a software build tool which can manage the project build, reporting, & documentation, primarily used for Java development and supported by most Java IDEs. It features dependency management, project structure standardisation, & automatic builds.

- **Gradle** is a build tool that supports multiple languages including Java, Kotlin, & Groovy. Its features include being highly customisable and being faster than Maven due to its incremental builds & caching. It's preferred for modern Java-based CI/CD pipelines and supports both Android & general Java applications.
- **Node Package Manager (NPM)** is a build tool for JavaScript / Node.js that features dependency management & building for JavaScript applications. It is used to build web-based frontend applications or backend applications in a CI/CD pipeline.

Other popular build tools include Yarn for JavaScript, PyBuilder or tox for Python, MSBuild for C#/.NET, & Rake for Ruby.

## 4.1 Build Tools in CI/CD Pipelines

**Continuous Integration (CI)** automatically integrates & tests code on each commit. **Continuous Deployment/Delivery (CD)** automatically deploys tested code to production or staging. Build tools serve many roles in CI/CD pipelines:

- **Integration:** when changes are pushed to the repository, the CI tool (e.g., GitHub Actions) triggers the build tool to compile & package the application.
- **Build Automation:** the build tool automatically handles downloading dependencies, compiling the code, & running tests. It ensures that the same version of the application is built every time.
- **Testing:** many build tools, such as Maven, integrate with testing frameworks (e.g., JUnit, Selenium) to run automated tests after the build.
- **Deployment:** the packaged application can be deployed to a server, containerised (e.g., with Docker), or distributed using CD tools.

Build tools automate the process of building & testing code with each integration to the repository, i.e. **Continuous Integration**. They ensure new changes don't break existing code by running automated tests as part of the build process and exhibit *fail-fast* behaviour: if a test or build fails, the developer is notified immediately.

After a successful build & testing, the build tools package the code, ready for deployment, i.e. **Continuous Deployment**. **Artifact creation** is when the builds create deployable artifacts (e.g., JAR files, WAR files, Docker images). The pipeline can then engage in **automated deployment** by deploying the artifact to a server, cloud, or container.

An example build tool workflow in a CI/CD pipeline with GitHub Actions may look like the following:

1. **Code Push:** a developer pushes new code to the GitHub repository.
2. **CI Tool Trigger:** GitHub Actions detects the change and triggers the pipeline.
3. **Dependency Resolution:** the build tool (e.g., Maven) fetches dependencies from repositories.
4. **Compile & Build:** the build tool compiles & packages the code into executable binaries (e.g., JAR, WAR).
5. **Testing:** run unit & integration tests automatically.
6. **Package & Deploy:** the build tool creates the package, and the CI/CD pipeline deploys it to staging or production.

## 4.2 Maven

**Maven** is a software build tool which can manage the project build, reporting, & documentation, primarily used for Java development and supported by most Java IDEs. It is widely used in Spring Boot projects.

1. Compile source code.
2. Copy resources.

3. Compile & run tests.
4. Package projects.
5. Deploy project.
6. Cleanup files.

Developers wanted:

- to make the build process easy.
- a standard way to build projects.
- a clear definition of what the project consists of.
- an easy way to publish project information and a way to share JARs across several projects.

The result is a tool that developers can use to build & manage any Java-based project. It embraces the idea of "convention over configurations".

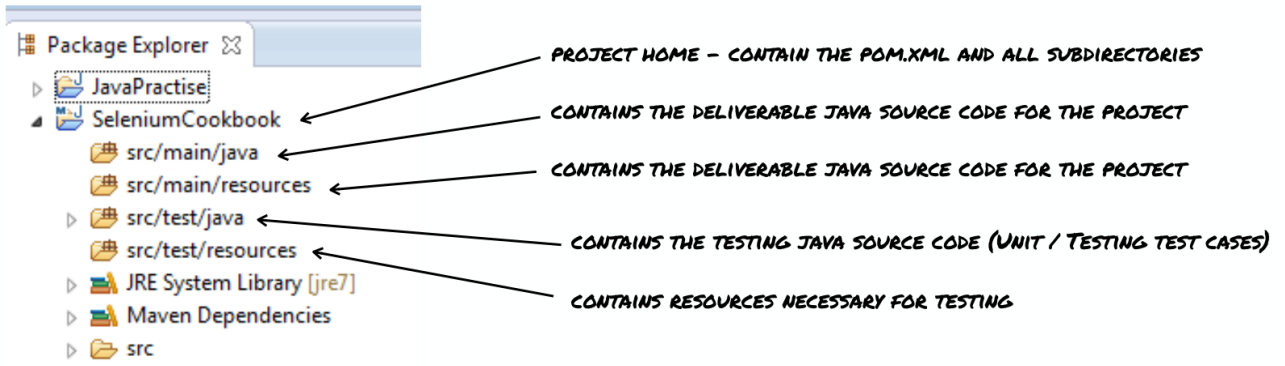


Figure 9: Maven Default Directory Structure

The command `mvn package` compiles all the Java files, runs any tests, and packages the deliverable code & resources into `target/my-app-1.0.jar**`.

The `pom.xml` file is an XML document that contains all the information that Maven requires to automate a build of your software. The `pom.xml` is automatically updated on demand, but can be manually configured as well. The POM provides all the configuration for a single project:

- General configuration covers the project's name, its owner, & its dependencies to other projects.
- One can also configure individual phases of the build process, which are implemented as plugins. E.g., one can configure the `compiler-plugin` to use Java 1.5 for compilation, or specify packaging the project even if some unit tests fail.

Larger projects should be divided into several modules, or sub-projects each with its own POM. All root POM can compile all the modules with a single command. POMs can also inherit configuration from other POMs; all POMs inherit from the super-POM by default. The super-POM provides default configuration, such as default source, default directories, default plugins, etc.

### 4.2.1 Plugins

Maven build projects based on convention; it expects files to be in a certain place, which is very useful when developing in teams.

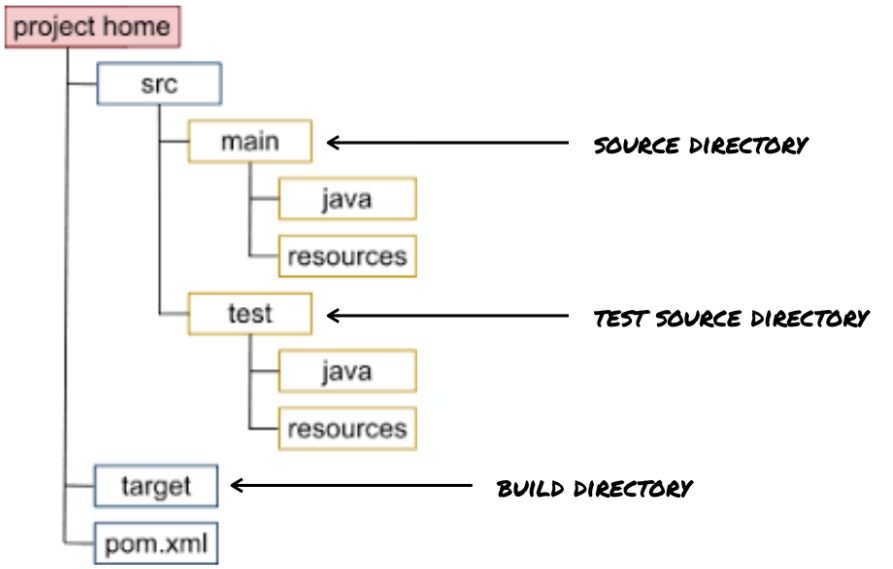


Figure 10: Maven Conventional Directory Structure

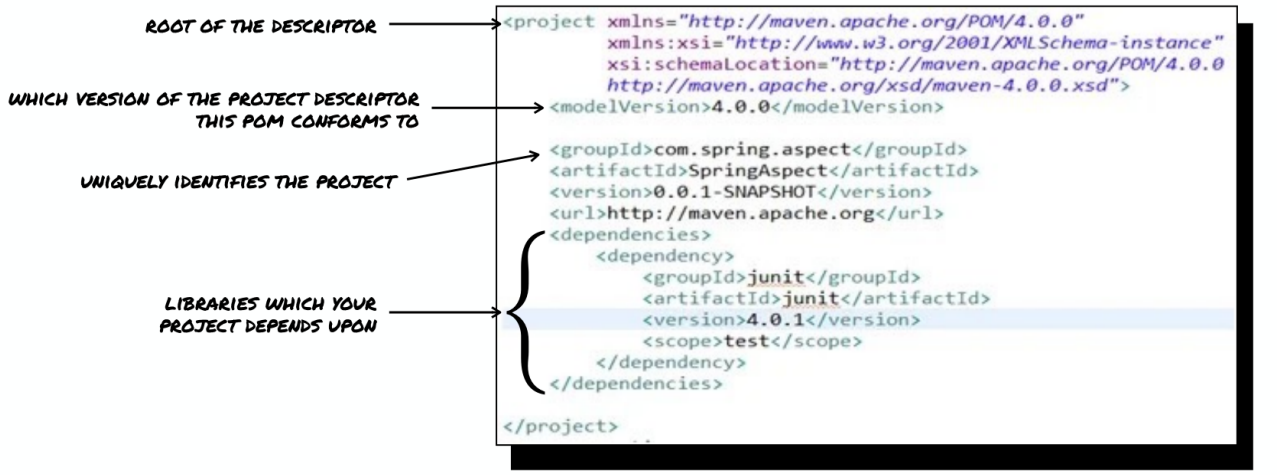


Figure 11: Maven `pom.xml`

A Maven **plugin** is an extension or add-on module that enhances the functionality of Apache Maven. Maven plugins provide additional capabilities & tasks that can be executed during the build process or as part of project lifecycle management. These plugins are typically packages as JAR (Java Archive) files and can be easily added to a Maven project's configuration.

The **core plugins** are plugins corresponding to default core phases (e.g., clean, compile). They may also have multiple goals.

### 4.2.2 Build Lifecycle

The process for building & distributing a particular project is clearly defined. It comprises a list of named phases that can be used to give order to goal execution. Goals provided by plugins can be associated with different phases of the lifecycle. e.g., by default, the goal `compiler:compile` is associated with the compile phase, while the goal `surefire:test` is associated with the test phase. `mvn test` will cause Maven to run all goals associated with each of the phases up to & including the test phase.

### 4.2.3 Dependency Management

**Dependency management** is a central feature in Maven. The dependency mechanism is organised around a co-ordinate system identifying individual artefacts such as software libraries or modules (e.g., JUnit). If your project depends on a JAR file, Maven will automatically retrieve it for you, and store it in the user's local repository. If the JAR file depends on other libraries, Maven will ensure these are also included: these are known as **transitive dependencies**. This wasn't always a part of Maven, so it's a huge benefit.

Dependency management features supported include:

- Management: you can specify library version that transitive dependencies should use.
- Scope: include dependencies appropriate for the current stage of the build, i.e., compile, test, run, etc.
- Exclude dependencies: if project X depends on Project Y, and Project Y depends on Project Z, you can choose to exclude Project Z from your build.

### 4.2.4 Local & Remote Repository

The remote repository `https://mvnrepository.com/` contains libraries for almost everything: cloud computing, date & time utilities, HTML parsers, mail clients, etc. Once you specify the correct details in the POM file, Maven will automatically fetch the library for you on build.

The local Maven repository can be found at `~/.m2/`. Maven will search the local repository first and then search third-party repositories if it does not find what it's looking for. You can create your own repository and share it within your organisation.

## 4.3 Spring Boot

**Spring Boot** is a framework for building standalone Java applications with embedded servers that streamlines Java application development. It provides pre-configured, out-of-the-box functionality to avoid boilerplate code. It reduces configuration & setup, focuses on *convention over configuration*, and is compatible with microservices architecture, REST APIs, & monolithic apps.

### 4.3.1 Spring vs Spring Boot

**Spring** is a comprehensive framework for building any Java application requiring more manual configuration & management of dependencies & application context. Spring Boot is an extension of the Spring framework aimed at simplifying development, configuration, & deployment, especially for microservices & cloud-based applications.

| Spring | Spring Boot |
|---|---|
| Requires an external embedded server (e.g, Tomcat, Jetty, etc.) | Comes with an embedded server (Tomcat/Jetty) |
| Highly flexible but requires more setup effort | Simplifies Spring projects, reducing setup time |
| Best for complex, large-scale applications | Ideal for microservices & fast prototypes |
| Requires WAR file and deployment on external server | Packaged as JAR with an embedded server for easy deployment |

Table 3: Technical Differences between Spring & Spring Boot

Choose Spring when:

- Your project requires extensive customisations.
- You're building a complex enterprise application where flexibility & modularity are necessary.
- You have a team experiences in managing detailed configurations.

Choose Spring Boot when:

- You're building microservices or need quick iterations in development.
- You want an all-in-one solution with auto-configuration.
- Your focus is on simplicity & speed without worrying about configuration details.

| **Spring** | **Spring Boot** |
|---|---|
| Framework for building complex enterprise-level Java applications | Simplified framework to quickly build microservices or standalone apps |
| Slower to set up due to configuration | Faster development with minimal setup |
| Provides maximum flexibility & customisation | Less flexibility, focuses on ease of use |
| Suitable for large-scale, complex, highly customised apps | Suitable for small/medium projects, microservices, & rapid development |

Table 4: High-Level Differences between Spring & Spring Boot

### 4.4 GitHub Actions

**GitHub Actions** automates your workflow by triggering events such as `push`, `pull_request`, & `release`. It easily integrates with other tools like Docker, AWS, Heroku, etc. Key components of GitHub Actions include:

- **Workflows** are defined in YAML in the `.github/workflows/` folder and are triggered by events such as `push`, `pull_request`, etc.
- **Jobs** define units of work that run on a **runner** (e.g., `ubuntu_latest`, `macos-latest`). They can be run sequentially or in parallel.
- Each job consists of a series of **steps**, e.g., checking out the code, building, testing, etc.
- **Runners** include GitHub-hosted runners (e.g., Ubuntu, macOS) but you can also have self-hosted runners to run on your own infrastructure.
- **GitHub Secrets** can be used to securely store sensitive information (e.g., API keys) and are accessible in workflows as `secrets.MY_SECRET_KEY`.

**Composite actions** can be used to define reusable workflows.

## 5 Containerisation

**Containerisation** is a method of packaging software & all of its dependencies so that it can run uniformly & consistently across any infrastructure. Containers are isolated environments in which applications run, ensuring consistent behaviour across different environments (e.g., development, testing, production).

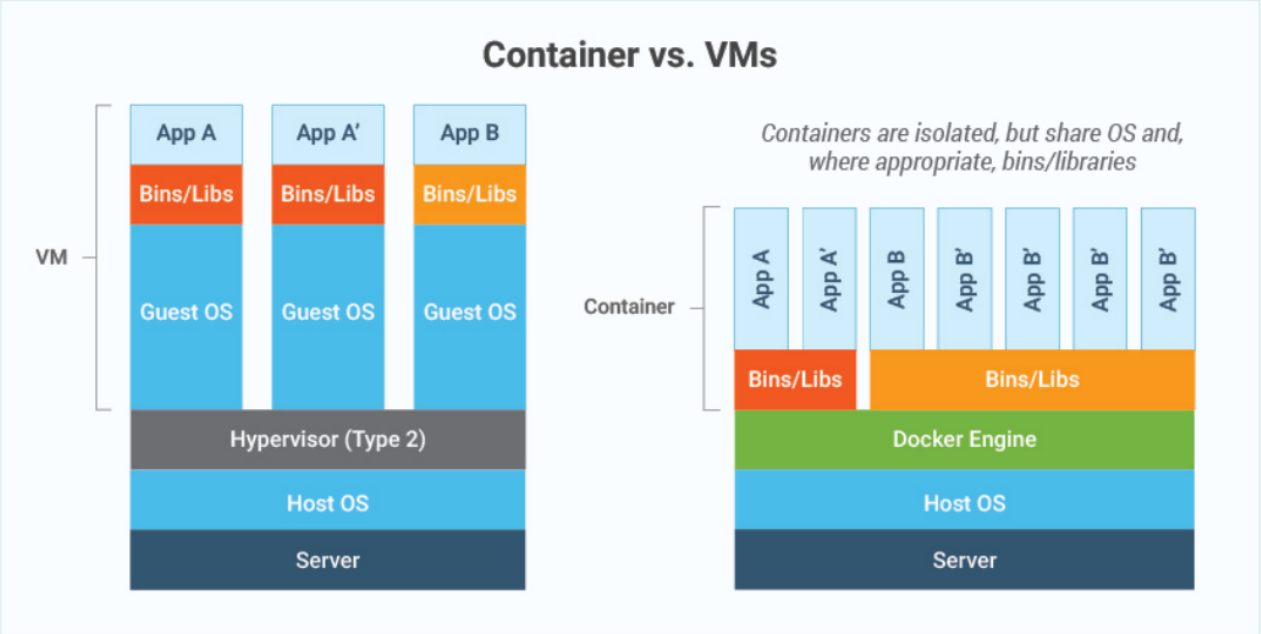


Figure 12: Virtual Machines vs Containers

Containers virtualise the operating system, while virtual machines virtualise the hardware. Containers share the OS kernel but isolate the application & its dependencies.

| **Virtual Machines** | **Containers** |
|---|---|
| Heavy & resource-intensive | Lightweight, share the OS kernel |
| Requires an entire OS for each VM | Faster startup, less overhead |
| Slower startup & resource usage | More efficient resource usage |

Table 5: Virtual Machines vs Containers

Reasons for using containerisation include:

- **Portability:** containers ensure that applications run the same regardless of the environment.
- **Scalability:** containers can be scaled easily, making them ideal for microservice architectures.
- **Efficiency:** containers are lightweight and use fewer resources than traditional VMs.
- **Isolation:** each container is isolated, meaning multiple containers can run on the same host without interference.
- **Faster deployment:** containers can be started in seconds, enabling fast deployments & rollbacks.

## 5.1 Docker

**Docker** is an open-source platform for building, deploying, & managing containerised applications. It simplifies container creation & management, and provides a consistent environment for development, testing, & production. Advantages of using Docker include:

- **Consistency across environments:** Docker ensures that the application runs the same regardless of where it's deployed. Eliminates the "works on my machine" problem by providing a consistent environment across all stages of development.
- **Isolation:** containers provide process isolation, so multiple applications or microservices can run side-by-side without interfering with each other.
- **Scalability:** containers are lightweight and can be easily scaled horizontally (i.e., replicating containers to handle more traffic).

- **Efficiency:** compared to virtual machines, containers use fewer resources since they share the host machine's kernel, making them much faster to start and stop.
- **Portability:** applications packaged in Docker containers can be easily moved across environments, cloud platforms, & OSes, ensuring smooth & reliable deployments.
- **Easy integration:** seamlessly integrates with CI/CD tools & workflows.

Key terms in Docker include:

- **Image:** a lightweight, standalone, executable package that includes everything an application needs (code, runtime, libraries, dependencies).
- **Container:** a runtime instance of an image. While images are static, containers are dynamic and can be started, stopped, or moved across environments.
- **Dockerfile:** a text file with instructions to build a docker image. It defines the steps to configure an environment, install dependencies, and set up the application.
- **Docker Hub:** a cloud-based repository for finding & sharing container images, both public & private.

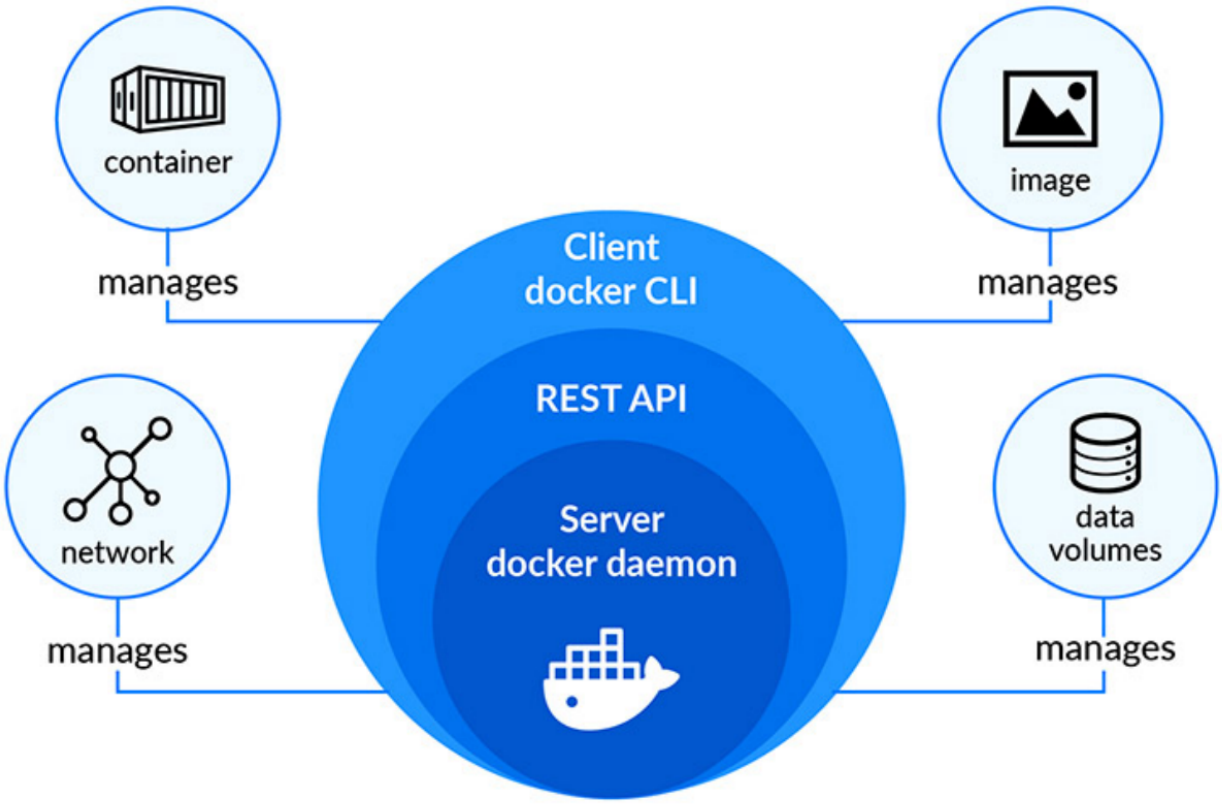


Figure 13: Docker Architecture Overview

Key Docker components include:

- **Docker Engine:** the runtime that runs & manages containers.
- **Docker Hub:** a public repository where users can publish & share container images.
- **Dockerfile:** a text files that contains instructions on how to build a Docker image.

### 5.1.1 Building & Running Applications in Docker Containers

The dockerfile structure is as follows:

- **`FROM`**: the base image (e.g., `openjdk:17` for Java applications).
- **`WORKDIR`**: the directory inside the container where the application will reside.
- **`COPY`**: copies files from the host system into the container.
- **`RUN`**: executes commands (e.g., installing dependencies).
- **`CMD`**: defines the default command to run when the container starts (e.g., `java -jar app.jar`).

```
FROM openjdk:17-jdk-slim
WORKDIR /app
COPY target/musicFinder-1.0 jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Listing 1: Example Dockerfile for a Spring Boot Application

To build the Docker image from the Dockerfile:

```
docker build -t my-app .
```

The Docker container can then be run, mapping the container's port 8080 to the host's port 8080:

```
docker run -p 8080:8080 my-app
```

### 5.1.2 Docker Best Practices

- **Keep images lightweight:** use minimal base images (e.g., `alpine`) to reduce the size of the final image, leading to faster build times & fewer security vulnerabilities.
- **Multi-stage builds:** separate the build environment from the final image to reduce size & improve performance.

```
FROM maven:3.8-jdk-11 AS builder
WORKDIR /build
COPY . .
RUN mvn clean package

FROM openjdk:11-jre-slim
WORKDIR /app
COPY --from=builder /build/target/app.jar /app.jar
CMD ["java", "-jar", "/app.jar"]
```

- **Use `.dockerignore`:** similar to `.gitignore`, it prevents unnecessary files from being copied into the container, optimising build times.
- **Tagging:** tag your images `docker build -t my-app:v1` for version control & easier management of deployments.
- Security best practices include:
  - Regularly update base images to avoid security vulnerabilities.
  - Avoid running containers as root.
  - Scan your Docker images for vulnerabilities using tools like Clair or Anchore.

### 5.1.3 Common Docker Commands

- Listing containers:
  - `docker ps`: list running containers.
  - `docker ps -a`: list all containers, including stopped ones.
- Stopping / removing containers:
  - `docker stop container_id`: stops a running container.
  - `docker rm container_id`: removes a stopped container.
- Viewing logs:
  - `docker logs container_id`: shows the logs of a container.
- Entering a running container:
  - `docker exec -it container_id /bin/bash`: opens a shell inside the running container.

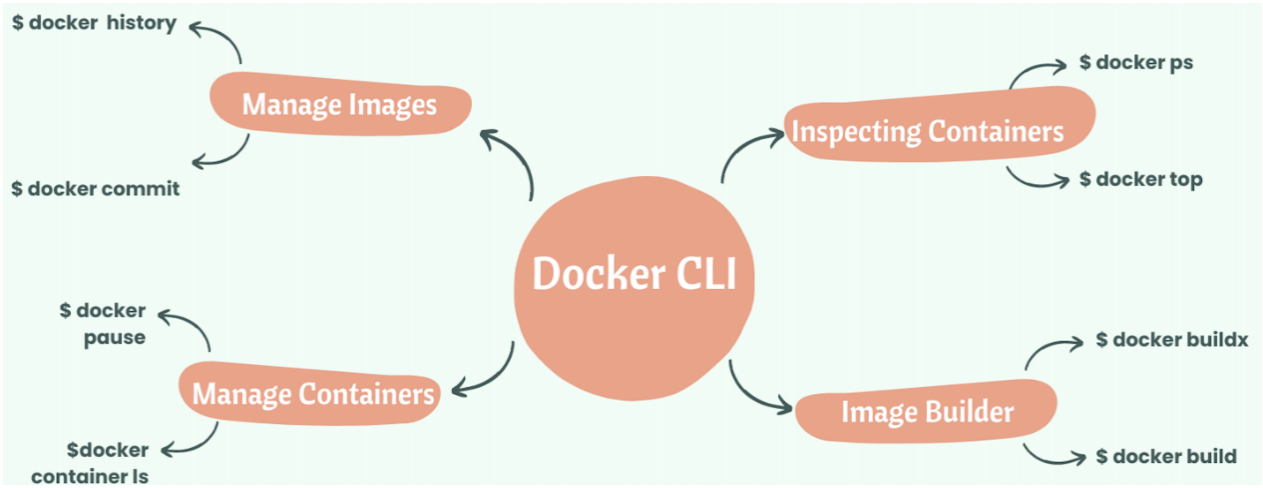


Figure 14: Some Common Docker Commands

### 5.1.4 Practical Docker Example with Spring Boot

Prerequisites:

- Java 17 SDK.
- Docker (running).
- Maven.
- Sprint Boot CLI (Optional, but useful for scaffolding projects).

1. Initialise a Spring Boot project with Maven using Spring Initializr to generate the project structure.
2. Create the Song Suggester Logic in `src/main/java/com/example/songsuggester/SongSuggesterController.java`:

```
1 package com.example.songsuggester;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.RestController;
```

```
import org.springframework.web.client.RestTemplate;
import org.json.JSONObject;
import java.util.Random;

@RestController
public class SongSuggesterController {
    @GetMapping("/suggest")
    public String suggestSong() {
        String apiUrl = "https://itunes.apple.com/search?term=pop&limit=10";
        RestTemplate restTemplate = new RestTemplate();
        String result = restTemplate.getForObject(apiUrl, String.class);

        // Parse the JSON response
        JSONObject jsonObject = new JSONObject(result);
        var tracks = jsonObject.getJSONArray("results");

        // Randomize the selection
        Random rand = new Random();
        int randomIndex = rand.nextInt(tracks.length());
        var randomTrack = tracks.getJSONObject(randomIndex);

        // Extract the song and artist name
        String song = randomTrack.getString("trackName");
        String artist = randomTrack.getString("artistName");

        return "Today's song suggestion: " + song + " by " + artist;
    }
}
```

3. Build and run the Spring Boot application:

   i. Navigate to the project folder.

   ii. Run the Maven build command:

   ```
   mvn clean install
   ```

   iii. Start the Spring Boot application:

   ```
   mvn spring-boot:run
   ```

   iv. Access the application by navigating to `http://localhost:8080/suggest` in your browser.

4. In the root of your project directory, create a Dockerfile:

```
# Use an official OpenJDK runtime as a parent image
FROM openjdk:17-jdk-slim

# Set the working directory inside the container
WORKDIR /app

# Copy the project JAR file into the container
COPY target/song-suggester-0.0.1-SNAPSHOT.jar app.jar

# Expose the port the app runs on
EXPOSE 8080
```

```
12
13 # Run the JAR file
14 ENTRYPOINT ["java", "-jar", "app.jar"]
```

5. Build the Docker image from the Dockerfile:

```
1 docker build -t song-suggester .
```

6. Run the Docker container:

```
1 docker run -p 8080:8080 song-suggester
```

## 5.2 Container Orchestration

**Container orchestration** automates the management, deployment, scaling, & networking of containers. It's crucial when dealing with a large number of containers running across multiple environments because as the number of containers grows, manually managing them becomes unfeasible.

Key components of container orchestration include:

- **Scheduling:** automatically assigns containers to hots machines based on resource availability.
- **Scaling:** dynamically adds or removes containers based on demand.
- **Networking:** manages the communication between containers and ensures that they can interact securely.
- **Load balancing:** distributes traffic across multiple containers to optimise resource usage.
- **Service discovery:** automatically detects and connects services running in different containers.

Orchestration benefits DevOps in the following ways:

- **Automation:** simplifies repetitive tasks such as deployment, scaling, & rollback.
- **High availability:** distributes workloads across different machines, ensuring that services remain available.
- **Fault tolerance:** automatically restarts or replaces failed containers and reroutes traffic to healthy containers.
- **Scalability:** orchestrators can dynamically scale the number of running containers to handle increased traffic.

Challenges with container orchestration include:

- **Complexity:** orchestration platforms can introduce signification complexity, especially for small teams.
- **Learning curve:** tools like Kubernetes have a steep learning curve for new users.
- **Resource overhead:** orchestrators can consume considerable resources, particularly when managing large-scale systems.
- **Networking:** configuring secure & reliable networking between containers can be challenging.

Popular container orchestration tools include:

- **Kubernetes:**
  - Most widely used container orchestration platform.
  - Manages containerised application across clusters of machines.
  - Handles self-healing, automated rollouts, & scaling.
- **Docker Swarm:**

  - Built-in Docker tool for orchestration.
  - Easier to set up but less feature-rich than Kubernetes.
  - Ideal for smaller setups with Docker-native capabilities.

- **Apache Mesos:**
  - General-purpose distributed systems platform that supports container orchestration.
  - Suitable for large-scale environments requiring both container & non-container workloads.

### 5.2.1 Kubernetes

The Kubernetes architecture consists of a master node & worker nodes:

- **Master Node:** manages the Kubernetes cluster.
  - **API Server:** entry point for REST operations.
  - **Scheduler:** assigns containers to nodes.
  - **Controller Manager:** ensures the desired state of the system.
- **Worker Nodes:** hosts running containerised applications.
  - **Kubelet:** ensures containers are running on a node.
  - **Pod:** smallest deployable unit consisting of one or more containers, with shared storage & network resources.
  - **Kube-Proxy:** handles networking within Kubernetes.

Key Kubernetes concepts include:

- **Service:** an abstraction that defines a logical set of pods & a policy for accessing them.
- **Deployment:** manages pod scaling & rolling updates for your application.
- **Namespace:** provides scope for resources within a Kubernetes cluster, helping organise & manage resources.

The steps to run Kubernetes locally are as follows:

1. **Install Minikube:** Minikube allows you to run Kubernetes on a single node.

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

2. Install `kubectl`:

```
sudo apt-get install -y apt-transport-https
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee -a
↪ /etc/apt/sources.list.d/kubernetes.list
sudo apt-get update
sudo apt-get install -y kubectl
```

   Verify the installation by running `kubectl version`.

3. Running Kubernetes:
   - Use the following commands to start interacting with your Kubernetes cluster:

```
kubectl cluster-info
kubectl get nodes
```

You can deploy containers & pods using:

```
kubectl apply -f <your-deployment-file>.yaml
```

4. Manage Kubernetes with Helm (optional):
   - **Helm** is a package manager for Kubernetes that makes deployment easier.
   - To install Helm:

```
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
```

   - You can then start deploying applications using Helm Charts.

To deploy the `song-suggester` app with Kubernetes:

1. Create a Docker image for the `suggest-music` app:

```
docker build -t song-suggester .
```

2. Deploy the Docker container in a Kubernetes pod:

```
kubectl create deployment song-suggester --image=song-suggester
```

3. Expose the app using a service to make it accessible outside the Kubernetes cluster:

```
kubectl expose deployment song-suggester -type=LoadBalancer -- port=8080
```

4. Scale the application to run multiple instances:

```
kubectl scale deployment song-suggester --replicas=5
```

# 6 DevSecOps

Traditional development cycles considered security at the end, leading to costly vulnerabilities in production. Modern applications involve complex microservices, and frequent releases that increase attack surfaces. Key risks in modern development include:

- Faster development leads to higher risks; without security baked into the process, vulnerabilities can go unnoticed until late stages.
- Complex architectures such as containerised environments & cloud infrastructure create new attack vectors.
- Increasing rates of cyberattacks: 2023 saw a rise in supply chain attacks, phishing, & ransomware incidents

**DevSecOps** involves integrating security throughout the entire DevOps lifecycle. It involves **shift-left security** which consists of moving security practices earlier in the development process to catch vulnerabilities before deployment. Doing so makes detecting vulnerabilities cheaper & easier to fix due to earlier detection, reduces attack vectors from the start of the development process, and gives real-time visibility into security risks during development, not just post-deployment.

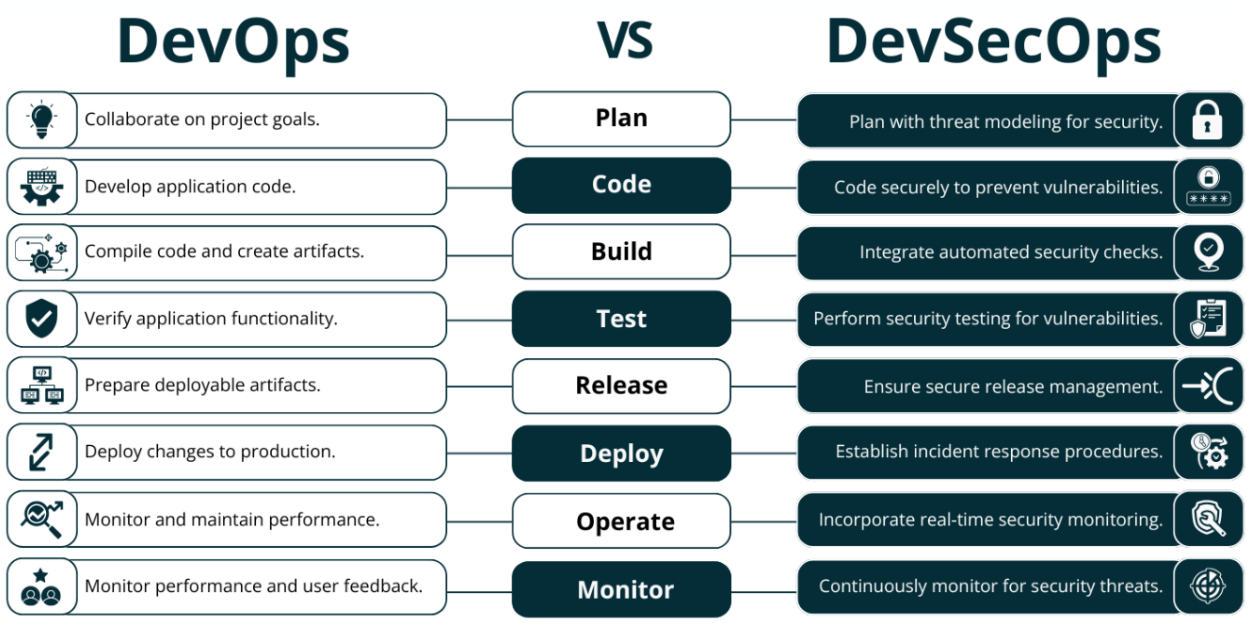


Figure 15: DevOps vs DevSecOps

Benefits of DevSecOps include:

- Reduced time to fix bugs: fixing vulnerabilities earlier in development is faster & cheaper.
- Continuous Security: automated tests & monitoring ensure security across the pipeline.
- Better Compliance: ensures adherence to industry standards (e.g., GDPR, PCI-DSS) through continuous security checks.
- Improved Collaboration: security becomes a shared responsibility, promoting teamwork.

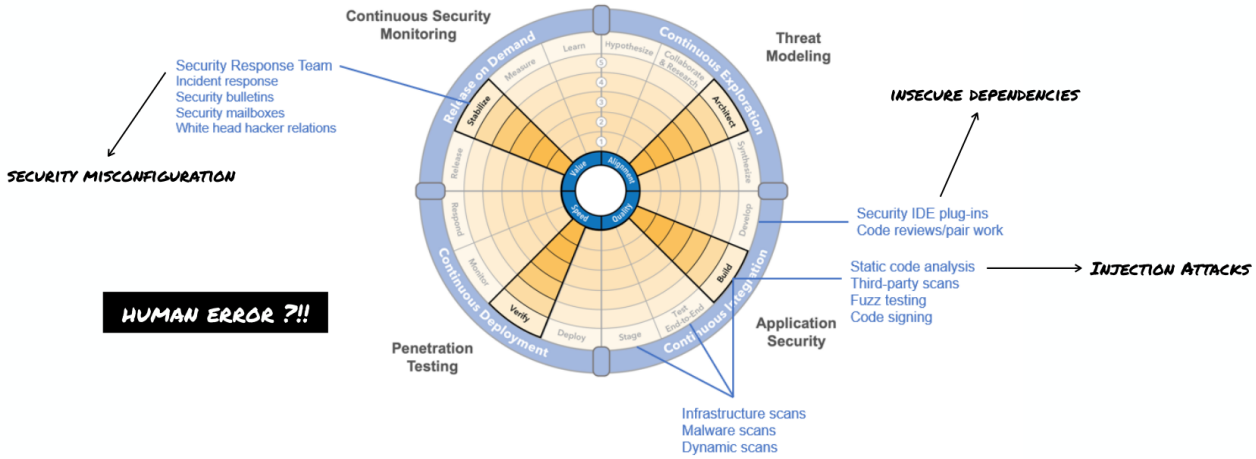


Figure 16: Key Vulnerabilities

Key security metrics include:

- **Mean Time To Detection:** how fast can you detect security vulnerabilities?
- **Mean Time To Remediation:** how quickly can you fix vulnerabilities once detected?
- **Mean Time To Failure:** average amount of time a non-repairable system is expected to function before it fails.

$$\text{MTTF} = \frac{\text{Total Operational Time}}{\text{Number of Failures}}$$

- **Mean Time Before Failures:** similar metric to MTTF but for repairable systems that includes the time to failure and the time it takes to repair the system.

$$\text{Mean Time Between failures} = \text{Mean Time To Detection} + \text{Mean Time To Remediation}$$

DevSecOps best practices include:

- **Security as Code:** treat security policies & tests like code. Use version control, collaboration, & automation reduces attack vectors from the start of the development process.
- **Automated Testing:** integrate automated security testing into CI/CD pipelines (static, dynamic, & dependency checks).
- **Continuous Monitoring:** implement tools for real-time monitoring of security events in production.
- **Infrastructure as Code (IaC):** automate secure configurations of infrastructure to avoid security misconfigurations.
- **Training & Awareness:** regularly train teams on the latest security practices & vulnerabilities.

## 6.1 Security Vulnerabilities in Code

**Security vulnerabilities** are weaknesses in a system or application that attackers can exploit to compromise confidentiality, integrity, or availability. Understanding common vulnerabilities is key in building more secure applications, particularly in a DevSecOps pipeline. Common vulnerabilities include:

- SQL injection.
- Cross-Site Scripting (XSS).
- Cross-Site Request Forgery (CSRF).
- Insecure deserialisation.
- Impoper input validation.

### 6.1.1 SQL Injection

**SQL injection** is a technique wherein attackers manipulate an application's SQL queries by injecting malicious SQL code through user inputs (e.g., forms or URL parameters). If user input is not properly sanitised, the attacker can execute arbitrary queries to retrieve, modify, or delete sensitive data. Types of SQL injection include:

- **Classic SQL injection:** occurs by inserting SQL queries in user inputs.
- **Blind SQL injection:** SQL queries are injected without knowing the output.
- **Error-Based SQL injection:** an attacker retrieves information from error messages returned by the database.

SQL injection can be prevented with:

- **Input validation:** ensure all user inputs are properly validated.
- **Parameterised queries:** use placeholders for query parameters to prevent direct user input into SQL queries.
- **Stored procedures:** use database-stored procedures instead of dynamic SQL queries.
- **ORM Frameworks:** use Object-Relational Mapping tools such as JPA which generate secure queries.
- **Least privilege:** ensure that the database user has minimal privileges.

### 6.1.2 Cross-Site Scripting

**Cross-Site Scripting (XSS)** is a type of injection attack wherein malicious scripts are injected into otherwise benign & trusted websites. The attacker tricks the victim's browser into executing malicious scripts, potentially compromising sensitive information such as cookies or login tokens. Types of XSS include:

- **Stored XSS:** a malicious script is permanently stored on a target server, e.g. in a database. When users visit the page, the malicious script is delivered to their browser from the server.
- **Reflected XSS:** the injected script is reflected off a web server, often via an error message or search result. The user is tricked into clicking a malicious link, where the script is injected and executed.
- **DOM-based XSS:** the vulnerability occurs in the browser rather than the server. The malicious script is part of the Document Object Model.

Dangers of XSS include:

- **Data theft:** attackers can steal cookies, local storage data, & other sensitive information.
- **Session hijacking:** attackers can impersonate users by stealing session tokens.

XSS can be mitigated with:

- **Input validation:** always validate & sanitise user inputs.
- **Escaping user output:** escape any user input before rendering it in the HTML.
- **Content Security Policy (CSP):** use a strict CSP to block inline scripts, and only allow trusted scripts to be executed.
- **HTTPOnly Cookies:** prevent JavaScript from accessing cookies.

### 6.1.3 Cross-Site Request Forgery

**Cross-Site Request Forgery (CSRF)** is an attack wherein a user is tricked into performing actions they didn't intend to by sending an unauthorised request to a trusted web application. This usually happens when the user is already authenticated and their browser automatically includes credentials (e.g., cookies, tokens). For example, a malicious website could contain a form that sends a request to a bank's API to transfer money; if the user is already logged into their bank account and visit this webpage, the browser will send the request with the user's bank credentials without them knowing.

## 6.2 Static Code Analysis

**Static code analysis** is the process of analysing source code without executing it to find potential errors, vulnerabilities, & coding violations. It helps to identify weaknesses early in the development process, preventing security vulnerabilities, bugs, & performance issues before they reach production. Key metrics include:

- **Code Complexity:** how difficult is it to understand, test, & maintain a codebase. It's often measured using **cyclomatic complexity** which counts the number of independent paths through a program's source code by counting the number of decision points (e.g., `if`, `while`, & `for` statements in the code. The higher the complexity, the more testing & resources are required to ensure stability.

  High complexity indicates that a piece of code has many decision points, which can increase the likelihood of bugs, make it harder to test, & make the code more prone to errors. Low complexity is often associated with cleaner, more maintainable code.

  It is considered best practice to refactor large functions into smaller, more manageable functions and to keep functions & classes single-responsibilty.

- **Code Duplication:** this occurs when the same block of code appears multiple times in a codebase. This leads to redundancy, increased maintenance costs, & the potential for inconsistency. It is a code smell that increases the risk of errors during future modifications, as if a bug is found in one piece of duplicated code, all other copies of that code need to be fixed as well. It leads to code that is harder to refactor, test, & maintain.

  Best practice to avoid code duplication includes following the DRY (Don't Repeat Yourself) principle, refactoring duplicated code into reusable functions, and to use inheritance or composition to remove repeated logic between classes.

- **Code Smells:** these are symptoms in the source code that *might* be indicative of deeper problems. They aren't necessarily bugs, but they make code harder to maintain and evolve over time. They often point to violations of coding principles or patterns that might lead to future issues. Examples include:
  - **Long methods:** methods that do too much, making them harder to read, test, & maintain.
  - **Large classes:** classes that handle too many responsibilities, making them complex.
  - **Primitive obsession:** overuse of basic data types instead of creating more meaningful classes.
  - **Feature envy:** a method in one class that heavily relies on the internal data of another class.
  - **God object:** a class that knows too much or does too much, violating the Single Responsibility principle.

  Code smells indicate areas that need refactoring. While they may not be immediate bugs, they make the code harder to manage in the long run are are often signs of technical debt. Best practice includes refactoring large classes & methods into smaller, focused components, applying design patterns like Strategy, Factory, or Observer to fix specific code smells, and following SOLID (Single Responsibility, Open-Closed, etc.) principles to avoid common smells.

Static code analysis works by performing:

1. **Scanning:** automated tools analyse source code for predefined patterns & rules.
2. **Analysis:** tools break down the code and check for common vulnerabilities.
3. **Reporting:** the analysis generates reports categorising findings as bugs, code smells, vulnerabilities, or performance issues.

Key benefits of static code analysis include:

- **Early bug detection:** reduces future costs by catching bugs early.
- **Improved code quality:** ensures adherence to coding standards & best practices.
- **Security improvements:** identifies security vulnerabilities.
- **Efficiency:** automated & continuous scanning of large codebases.
- **CI/CD integration:** seamless integration with DevOps pipelines.

Types of static code analysis include:

- **Lexical analysis:** analyses tokens & basic structures like variables & operators.
- **Syntax analysis:** examines the structure of the code & syntax rules.
- **Semantic analysis:** ensures that the logic of the code makes sense.
- **Data flow analysis:** examines how data flows through the program to identify unused variables or null pointer exceptions.

Tools for static code analysis include:

- **SonarQube:** supports Java, JavaScript, Python, etc. and features bug detection, vulnerability detection, code smells, & technical debt.
- **ESLint:** supports JavaScript and features style guide enforcement & best practices.
- **Checkmarx:** focuses on security and features vulnerability detection & OWASP Top 10 compliance.
- **FindBugs:** supports Java and features detection of bug patterns in Java bytecode.

Best practices for static code analysis include:

- **Integrate early:** apply static analysis from the beginning of the development cycle.
- **Automate in CI/CD Pipelines:** ensure continuous scanning for every commit or build.
- **Review findings regularly:** don't ignore reports, address issues promptly.
- **Tune rules:** customise analysis rules based on the project requirements.
- **Complement with manual code reviews:** tools can't detect everything, manual reviews are still necessary.

Limitations of static code analysis include:

- **False positives:** not all flagged issues are real problems, requiring manual inspection.
- **No runtime error detection:** unlike dynamic analysis, it can't identify errors that only occur during execution.
- **Context ignorance:** might not understand the broader context of the code.
- **Performance overhead:** large codebases can take time to analyse thoroughly.

# 7 Software Security

**Software security** is the concept of implementing mechanisms & adopting best development practices to protect software against malicious attacks, i.e. to make it resistant to attacks and to keep it functional when attacked. In traditional software design & development practices, software security was almost an afterthought. Secure software is defined as software engineered in such a way that its operation and functionality continues as normal even when subjected to malicious attacks.

In computer security, a **threat** is a potential negative action or event facilitated by a vulnerability that results in an unwanted impact to a computer system or application. A threat can either be a negative *intentional* event such as a cyberattack, or an *accidental* even such as an earthquake. Different definitions from different authorities for what constitutes a threat include:

- **ISO27005:** a potentital cause of an incident that may result in harm of systems & organisation.
- **NIST:** any circumstance or event with the potential to adversely impact organisation operations, organisational assets, or individuals through an information system via unauthorised access, destruction, disclosure, modification of information, and / or denial of service.
- **ENISA:** any circumstance or event with the potential to adversely impact an asset through unauthorised access, destruction, disclosure, modification of data, and / or denial of service.

## 7.1 Threat Classification

Microsoft classifies threats into the following categories:

- **Spoofing of user identity:** e.g., an attacker takes on the identity of an administrator.
- **Tampering:** e.g., an attacker changes an account balance.
- **Repudiation:** e.g., a user denies performing an action without either parties having any way to prove otherwise.
- **Information disclosure:** privacy breach or data leak.
- **Elevation of privilege:** an attacker elevates their own security level to an administrator.
- **Denial of Service (DOS):** a cyber attack in which the perpetrator seeks to make a machine or network resource unavailable to its intended users by temporarily or indefinitely disrupting services of a host connected to the internet.

The term **threat agent** is used to indicate an individual, thing, or a group that can manifest a threat. These incldue:

- Non-target specific, e.g. a computer virus, worms, trojans, & logic bombs.
- Employees, e.g. disgruntled staff or contractors.
- Organised crime & criminals.
- Corporation, e.g. partners or competitors.
- Human (unintentional), e.g., accidents, carelessness.
- Human (intentional), insider, outsider.
- Natural, e.g., flood, fire, lighting, meteor, earthquakes.

### 7.1.1 Requirement-Level Threats

Expertise in requirements engineering & information system security is a rare combination: customers & users don't know what they want with respect to security, and requirement engineers don't know what questions to ask to elicit security requirements. This combined lack of security expertise leads to missing or unidentified security requirements, resulting in security vulnerabilities.

### 7.1.2 Hardware-Level Threats

| Threat | Countermeasure |
|---|---|
| Eavesdropping devices (e.g., keyloggers) | Physical Security |
| Power outage | Uninterruptible Power Supply |
| Natural Disasters | Geographically dispersed redundancy to avoid a single point of failure |
| Sabotage | Physical Security |

Table 6: Hardware-Level Threats & Countermeasures

### 7.1.3 Code-Level Threats

Code-level threats include **unintentional** threats, which are mainly caused by a lack of secure coding knowledge. These can be mitigated with software security education & training, automatic static & dynamic code analysis, and peer code review.

**Intentional** threats can be prevented with peer code review, job rotation, & mandatory vacation.

### 7.1.4 Design-Level Threats

**Design-level threats** relate to weaknesses in principal OO design & object interaction, therefore secure design is more fundamental than secure coding, e.g., object attributes being public rather than private. Best OO design practices are captures in design patterns for security. Code implementation without a solid design is dangerous & costly.

### 7.1.5 Architectural-Level Design Threats

**Architectural design** decisions entail overarching design decisions. Widely accepted solutions to these recurring architectural design problems are referred to as **architectural patterns**.