Part A – 2 Marks Questions

1. What is Maven?

Maven is a powerful build automation and project management tool developed by Apache.

It simplifies compiling, testing, packaging, and deploying Java projects by using a standard lifecycle.

Maven also handles dependency management automatically.

Thus, it reduces the complexity of builds in large projects.

---

2. Define POM file in Maven.

POM (Project Object Model) is the core XML file in Maven, named `pom.xml`.

It contains project details like groupId, artifactId, version, dependencies, and build settings.

POM acts as the blueprint for the entire project.

It drives the Maven build lifecycle.

---

3. List any two Maven build phases.

- **Compile** – Compiles source code into bytecode.
- **Package** – Bundles the compiled code into JAR/WAR for deployment.

---

4. What are Maven profiles used for?

Maven profiles allow customization of builds for different environments.

They let you define environment-specific configurations (Dev, Test, Prod).

Profiles override default settings in POM.

This helps achieve flexibility in deployment.

---

5. Differentiate between local and central Maven repositories.

- **Local Repository**: Stored on the developer's machine (`~/.m2` folder).
- **Central Repository**: Public repository hosted online (`repo.maven.apache.org`).

  Local caches artifacts, while central provides global access.

  If not found locally, Maven fetches from central.

---

6. What is a Maven plugin? Give an example.

A plugin is a collection of goals that extend Maven's functionality.

They are used for compiling, testing, packaging, or deploying.

Example: **maven-compiler-plugin** compiles Java code.

Without plugins, Maven cannot execute lifecycle phases.

---

7. Define "Dependency management" in Maven.

Dependency management refers to handling project libraries in a structured way.

It ensures consistent versions across different modules.

Maven automatically downloads required JARs from repositories.

This reduces manual effort and avoids version conflicts.

---

8. What is Gradle?

Gradle is an advanced open-source build automation tool.

It supports multi-language projects (Java, C++, Python).

It uses Groovy/Kotlin DSL instead of XML.

Gradle is faster and more flexible than Maven.

---

9. Mention one advantage of Gradle over Maven.

Gradle supports **incremental builds** (only recompiles changed files).

This makes it much faster than Maven for large projects.

It also provides better caching and parallel execution.

Hence, it improves developer productivity.

---

10. What is Jenkins?

Jenkins is an open-source automation server.

It is widely used for Continuous Integration (CI) and Continuous Delivery (CD).

It automates build, test, and deployment pipelines.

Jenkins supports hundreds of plugins for tool integration.

---

11. List two features of Jenkins.

1. Supports distributed builds with master-agent architecture.

2. Integrates easily with tools like Git, Maven, Docker, and Kubernetes.

---

12. Define Jenkins plugin.

A Jenkins plugin extends Jenkins functionality.

It helps integrate external tools and services.

Example: **Git Plugin** for integrating Git repositories.

Plugins make Jenkins highly customizable.

---

13. What is Jenkins workspace?

A Jenkins workspace is a directory allocated for each job.

It stores source code, configuration, and build output.

Each build runs in its own workspace.

This prevents conflicts between different jobs.

---

14. What is the use of Git Plugin in Jenkins?

The Git Plugin allows Jenkins to connect to Git repositories.

It fetches source code for builds.

It also supports webhooks to trigger jobs on commits.

Thus, it enables CI with Git.

---

15. Mention any two steps to configure Jenkins with Maven.

1. Configure Maven installation in Jenkins Global Tool Configuration.
2. Add a build step “Invoke top-level Maven targets” in the job.

Part B – 12 Marks Questions

Q1. Explain the architecture of Maven and the role of POM file with an example.

Introduction:

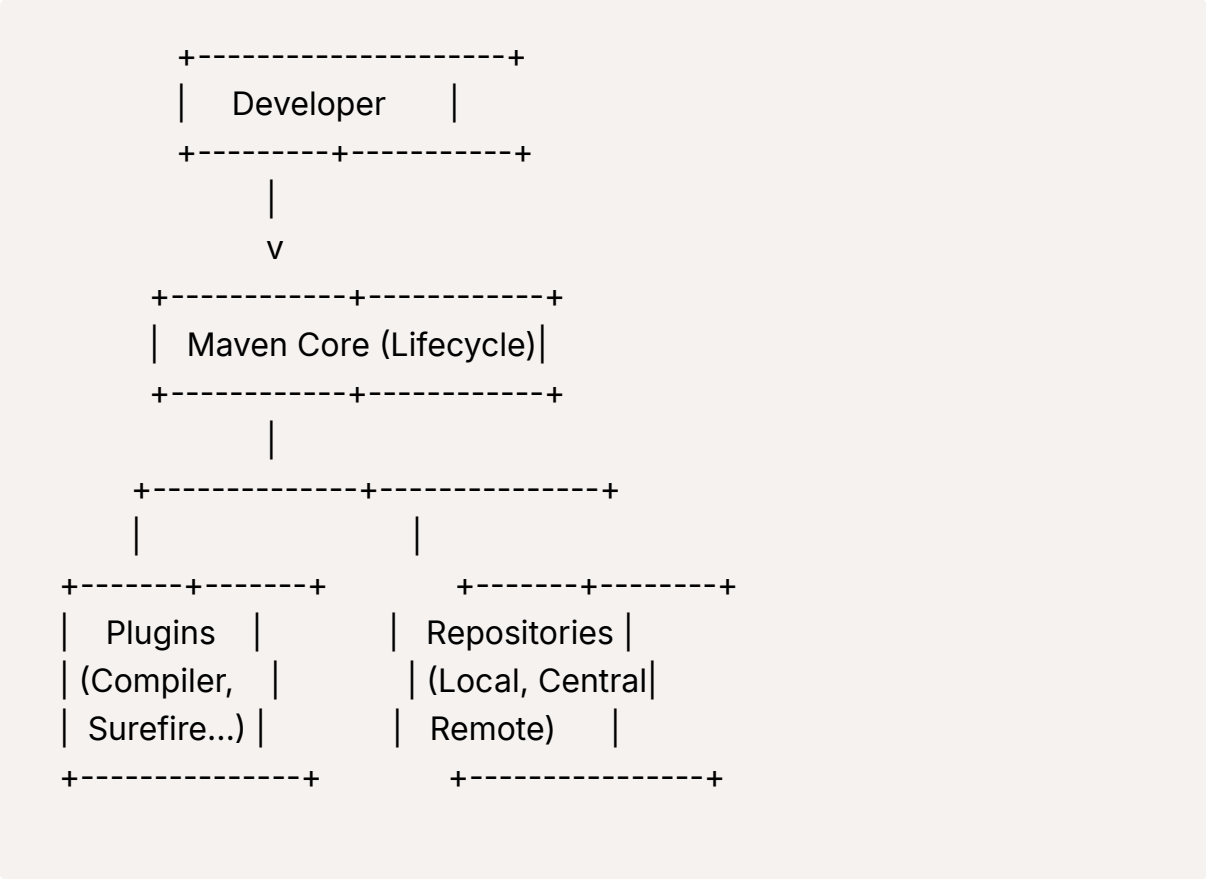
Maven is a project management and build automation tool that uses a centralized model for handling builds, dependencies, and project lifecycles. Its architecture is designed around a standard directory structure, plugins, repositories, and the Project Object Model (POM).

Architecture of Maven:

The Maven architecture consists of:

1. **Core** – Provides fundamental functionalities such as lifecycle management and dependency resolution.
2. **Plugins** – Extend the core functionalities. Each plugin contains goals that can be executed in phases (e.g., compiler plugin).
3. **Repositories** – Store libraries, plugins, and project artifacts. There are local, central, and remote repositories.
4. **Lifecycle** – Defines the sequence of steps in building and managing a project.
5. **POM (Project Object Model)** – The XML file (`pom.xml`) that contains configuration details about the project.

Diagram – Maven Architecture:



Role of POM file:

- Defines **project coordinates** (groupId, artifactId, version).
- Manages **dependencies** (automatic download and versioning).

- Configures **plugins** (compiler, surefire for testing, jar/war packaging).
- Defines **profiles** for environment-specific builds.

Example – pom.xml:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>myapp</artifactId>
  <version>1.0.0</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Conclusion:

The Maven architecture ensures standardization and automation in project builds. The POM file plays a central role by defining the structure, dependencies, and build configurations, making projects portable and maintainable.

Q2. Describe Maven build lifecycle with suitable phases.

Introduction:

A build lifecycle in Maven defines the sequence of steps required to build and distribute a project. Each lifecycle is composed of a set of **phases**, and each phase contains **plugin goals**.

Types of Lifecycles in Maven:

1. **Default Lifecycle** – Used for project deployment.
2. **Clean Lifecycle** – Cleans project directory before a new build.
3. **Site Lifecycle** – Generates project documentation.

Phases of Default Lifecycle:

- **Validate** – Validates project structure.
- **Compile** – Compiles the source code.
- **Test** – Executes unit tests using frameworks like JUnit.
- **Package** – Packages compiled code into JAR/WAR.
- **Verify** – Runs integration tests or checks.
- **Install** – Installs the package into the local repository.
- **Deploy** – Deploys the package to a remote repository.

Diagram – Maven Lifecycle:

```
Validate → Compile → Test → Package → Verify → Install → Deploy
```

Example:

If a developer runs:

```
mvn install
```

Maven executes all phases up to **install** → validate, compile, test, package, verify, install.

Conclusion:

The Maven build lifecycle automates the software development process, reducing manual errors and ensuring consistent builds across environments.

Q3. Discuss Maven repositories in detail (local, central, global).

Introduction:

Maven repositories are storage locations for project dependencies, plugins, and artifacts. They allow Maven to fetch required libraries during the build process.

Types of Repositories:

1. **Local Repository**
    - Located on the developer's system (`~/.m2/repository`).
    - Acts as a cache of downloaded dependencies.
    - Faster as it avoids repeated downloads.
2. **Central Repository**
    - Publicly available repository maintained by Apache (`repo.maven.apache.org`).
    - Contains a huge collection of open-source libraries.
    - Default source if dependency not found locally.
3. **Remote/Global Repository**
    - Company-specific or third-party hosted repositories (e.g., Nexus, Artifactory).
    - Useful for storing private libraries.
    - Provides enterprise-level dependency control.

Diagram – Maven Repository Structure:

```
Local Repository (~/.m2) ←→ Central Repository (repo.maven.org)
                ↑
                |
        Remote/Global Repository
```

Conclusion:

Maven repositories form the backbone of dependency management. Local repositories improve efficiency, central provides universal access, and global repositories enable enterprise-level customization.

Q4. Explain dependency management in Maven with an example.

Introduction:

Dependency management is one of Maven's most powerful features. It ensures that external libraries used by a project are automatically downloaded, versioned, and configured correctly without manual intervention.

Need for Dependency Management:

- Reduces the hassle of manually downloading JAR files.
- Ensures consistent versions of libraries across different modules.
- Avoids “JAR Hell” (version conflicts and duplicate libraries).
- Promotes modular development by reusing dependencies.

How Dependency Management Works in Maven:

1. A dependency is declared in `pom.xml` with groupId, artifactId, and version.
2. Maven first checks the **local repository** for the dependency.
3. If not found, it fetches it from the **central repository**.
4. If configured, Maven can fetch from **remote repositories**.
5. Dependencies can be **transitive** (a dependency can itself have dependencies).

Example – Dependency Declaration:

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>5.3.20</version>
  </dependency>
</dependencies>
```

Transitive Dependency Example:

If your project depends on **spring-core**, Maven automatically pulls other required dependencies like **commons-logging**.

Dependency Scope:

- **compile** (default) – Available at compile time.
- **test** – Available only for testing.
- **provided** – Provided by runtime environment (e.g., servlet API in Tomcat).
- **runtime** – Required for execution but not compilation.

Diagram – Dependency Flow:

```
POM.xml → Local Repository → Remote/Central Repository
                ↓
        Fetch Required JAR Files Automatically
```

Conclusion:

Maven's dependency management eliminates manual library handling, ensures consistent builds, and resolves transitive dependencies, thereby improving project stability and maintainability.

Q5. Compare Maven and Gradle with suitable examples.

Introduction:

Maven and Gradle are popular build automation tools. While Maven is older and XML-based, Gradle is newer, flexible, and script-based. Both serve the same purpose but differ in speed, flexibility, and ease of use.

Comparison Between Maven and Gradle:

Feature	Maven	Gradle
Build Script	XML (pom.xml)	Groovy/Kotlin DSL (build.gradle)
Performance	Slower, full builds	Faster, supports incremental builds
Flexibility	Fixed lifecycle phases	Flexible, task-based graph
Dependency Mgmt	Excellent, uses repositories	Uses same repositories as Maven
Learning Curve	Easier for beginners	Slightly steeper

Example – Maven Dependency (pom.xml):

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

Example – Gradle Dependency (build.gradle):

```
dependencies {
    testImplementation 'junit:junit:4.12'
}
```

Diagram – Conceptual Difference:

```
Maven → Lifecycle (Fixed Phases) → Compile, Test, Package
Gradle → Task Graph (Dynamic) → Execute Only Required Tasks
```

Conclusion:

Maven is best for simple, structured projects, while Gradle excels in large, enterprise-level projects where performance and flexibility matter. Most modern companies prefer Gradle for Android and complex builds, but Maven remains reliable for Java projects.

Q6. Explain the installation and configuration of Jenkins.

Introduction:

Jenkins is an open-source automation server used for CI/CD. Setting up Jenkins requires installation, configuration of system tools, and integration with version control systems like Git and build tools like Maven.

Steps for Installation:

1. **Download Jenkins:**
   - From jenkins.io.
   - Available for Windows (.msi), Linux (.war/.rpm), or Docker.
2. **Run Jenkins WAR file:**

   ```
   java -jar jenkins.war
   ```

   Jenkins starts on `http://localhost:8080`.
3. **Unlock Jenkins:**
   - Initial admin password is located in `.jenkins/secrets/initialAdminPassword`.

- Paste it into the web UI to continue.

4. **Install Plugins:**
   - Recommended plugins (Git, Maven, Gradle, Docker, etc.).

Configuration After Installation:

1. **Configure Global Tools:**
   - Set JDK, Git, Maven, and Gradle paths under *Manage Jenkins → Global Tool Configuration*.
2. **Configure Security:**
   - Create admin users.
   - Enable Role-based Authorization.
3. **Setup Jobs:**
   - Create Freestyle jobs or Pipeline jobs.
   - Define SCM (Git URL).
   - Add build steps (Maven/Gradle commands).

Diagram – Jenkins Installation Flow:

```
Download Jenkins → Start Jenkins → Unlock → Install Plugins → Configure Tools → Create Jobs
```

Conclusion:

Jenkins installation is straightforward, requiring Java setup and plugin configuration. Once installed, it becomes a powerful CI/CD server that automates builds, testing, and deployments.

Q7. Describe Jenkins architecture with a neat diagram.

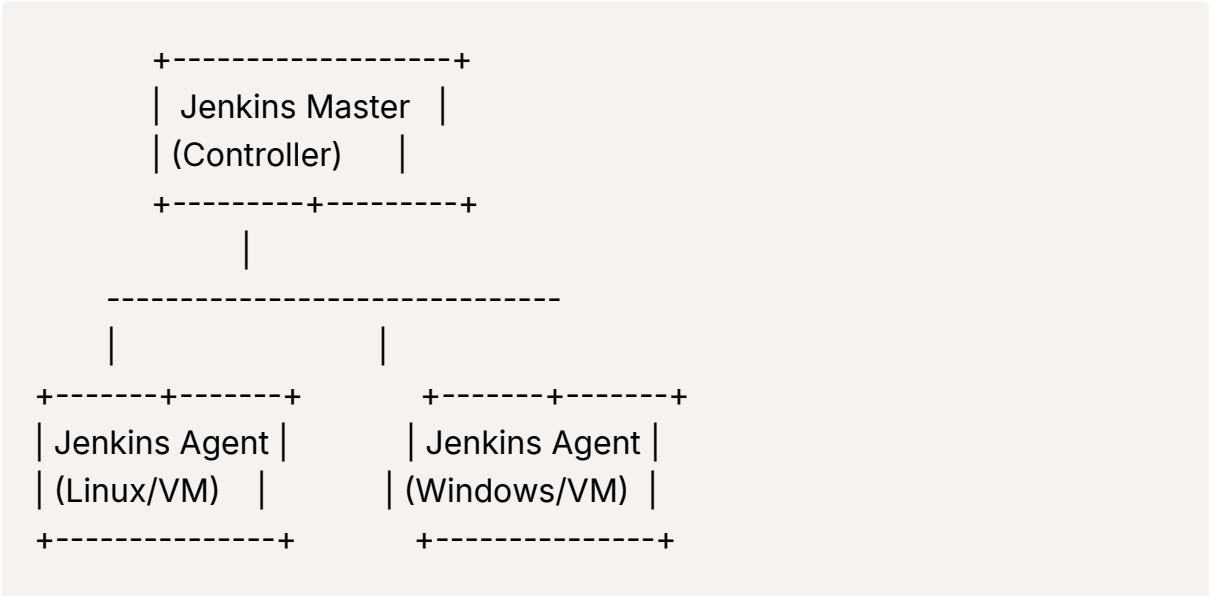
Introduction:

Jenkins follows a **Master-Agent (previously called Master-Slave)** architecture that allows distributed builds. The architecture provides scalability, flexibility, and efficient resource utilization in Continuous Integration (CI) and Continuous Delivery (CD) pipelines.

Components of Jenkins Architecture:

1. **Jenkins Master (Controller):**
    - Responsible for scheduling build jobs, monitoring agents, and recording results.
    - Provides a web-based UI for configuration and management.
    - Distributes tasks to agents.
2. **Jenkins Agents (Nodes):**
    - Execute build jobs assigned by the master.
    - Can be physical or virtual machines, Docker containers, or Kubernetes pods.
    - Allow parallel and distributed builds.
3. **Build Jobs:**
    - Defined tasks (e.g., compile code, run tests, deploy application).
    - Configured in Freestyle or Pipeline mode.
4. **Plugins:**
    - Extend functionality by integrating Jenkins with Git, Maven, Docker, Kubernetes, etc.
5. **Workspace:**
    - Each agent has its own workspace where code is checked out and build results are stored.

Diagram – Jenkins Architecture:



```
|                    |
Executes Builds      Executes Builds
```

Conclusion:

Jenkins architecture enables distributed builds, high scalability, and integration with multiple tools. This makes it suitable for enterprise-level CI/CD pipelines.

Q8. Write the steps to create and configure a Jenkins Job.

Introduction:

Jenkins Jobs define the specific tasks that Jenkins executes, such as building, testing, or deploying a project. Jobs can be Freestyle, Pipeline, or Multibranch.

Steps to Create and Configure a Job:

1. **Create a New Job:**
   - Go to Jenkins dashboard → *New Item*.
   - Enter a name for the job.
   - Select job type (Freestyle/Pipeline).
2. **Configure Source Code Management (SCM):**
   - Select Git as SCM.
   - Provide repository URL and credentials.
   - Configure branch (e.g., `main` or `develop`).
3. **Configure Build Triggers:**
   - Options include:
     - *Build periodically* (CRON expressions).
     - *Poll SCM* to check repository changes.
     - *Git Webhook* to trigger builds on commits.
4. **Add Build Steps:**
   - Choose *Invoke top-level Maven targets* or *Execute shell/Gradle tasks*.
   - Example: `mvn clean install`.
5. **Post-Build Actions:**

- Publish JUnit test reports.
- Deploy artifacts to remote servers.
- Send build notifications.

Diagram – Job Configuration Flow:

```
New Job → SCM Configuration → Build Triggers → Build Steps → Post-Buil
d Actions
```

Conclusion:

Jenkins Jobs are the backbone of CI/CD pipelines. By configuring SCM, build steps, and post-build actions, Jenkins automates repetitive tasks and ensures consistent delivery.

Q9. Discuss the role of Jenkins plugins with examples (Git, Parameter, HTML Publisher, etc.).

Introduction:

Jenkins is lightweight by default but can be extended with plugins. Plugins integrate Jenkins with external tools, SCMs, testing frameworks, and deployment environments.

Key Plugins and Their Roles:

1. **Git Plugin:**
   - Integrates Jenkins with Git repositories.
   - Enables source code checkout and webhook-based builds.
2. **Parameter Plugin:**
   - Adds build parameters (e.g., String, Choice, Boolean).
   - Enables dynamic and configurable builds.
3. **HTML Publisher Plugin:**
   - Publishes HTML reports generated during builds.
   - Useful for test coverage reports and documentation.
4. **Maven Integration Plugin:**

- Allows executing Maven goals directly from Jenkins.
- Useful for Java project builds.

5. **Docker Plugin:**
    - Builds and runs Docker containers from Jenkins.
    - Helps in containerized CI/CD pipelines.

Diagram – Jenkins with Plugins:

```
[Git Plugin] → Source Control
[Parameter Plugin] → Dynamic Builds
[HTML Publisher] → Reports
[Docker Plugin] → Containerized Deployments
```

Conclusion:

Plugins are the strength of Jenkins, transforming it into a flexible automation server. They allow Jenkins to integrate seamlessly with version control, build tools, and deployment environments.

Q10. Explain how to configure Jenkins to work with Java, Git, and Maven.

Introduction:

For Jenkins to automate builds, it must be integrated with essential tools like Java (JDK), Git, and Maven.

Steps for Configuration:

1. **Configure JDK in Jenkins:**
    - Go to *Manage Jenkins → Global Tool Configuration*.
    - Add a new JDK installation (point to JDK path or let Jenkins install automatically).
2. **Configure Git in Jenkins:**
    - Ensure Git is installed on the system.
    - In Jenkins configuration, set the Git executable path.
    - Verify by creating a job and pulling code from a Git repository.

3. **Configure Maven in Jenkins:**
    - Install Maven Integration Plugin.
    - Add Maven installation under *Global Tool Configuration*.
    - Define environment variables (`M2_HOME`).
4. **Create a Job Using These Tools:**
    - SCM → Git repository.
    - Build Step → *Invoke top-level Maven targets*.
    - Run command: `mvn clean install`.

Diagram – Jenkins Integration Flow:

```
[Git Repository] → [Jenkins SCM] → [Maven Build] → [JDK for Compilation]
```

Conclusion:

By configuring Java, Git, and Maven within Jenkins, developers can automate the entire pipeline—from fetching code to compiling and deploying artifacts—making CI/CD efficient and reliable.

Q1. Explain the Maven build lifecycle in detail with all build phases (compile, test, package, install, deploy) and suitable examples.

Introduction:

Maven follows a well-defined **build lifecycle** that automates the process of compiling, testing, packaging, and deploying Java applications. Instead of writing custom scripts, developers rely on predefined phases, which ensure consistency across projects.

Maven Lifecycles:

Maven defines **three lifecycles**, each containing phases:

1. **Default Lifecycle (Core Build Process)**

    Handles compilation, testing, packaging, installation, and deployment.

2. **Clean Lifecycle**

    Removes previous build files (`target/` directory).

3. **Site Lifecycle**

   Generates project documentation and reports.

Phases of the Default Lifecycle:

1. **Validate Phase**
   - Ensures the project structure and required configurations are correct.
   - Example: Checking if `pom.xml` is valid.
2. **Compile Phase**
   - Compiles the project's source code into bytecode.
   - Example: `mvn compile`.
3. **Test Phase**
   - Runs unit tests using JUnit/TestNG.
   - Example: `mvn test`.
4. **Package Phase**
   - Bundles compiled code into **JAR/WAR** files.
   - Example: `mvn package`.
5. **Verify Phase**
   - Ensures the package is valid and passes integration checks.
6. **Install Phase**
   - Installs the artifact into the **local repository** (`~/.m2/repository`).
   - Example: `mvn install`.
7. **Deploy Phase**
   - Copies final artifacts to a **remote repository** for sharing with other developers.
   - Example: `mvn deploy`.

Diagram – Maven Lifecycle:

```
Validate → Compile → Test → Package → Verify → Install → Deploy
```

Example Project Build:

Suppose a Java web app has the following POM dependency:

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>4.0.1</version>
  <scope>provided</scope>
</dependency>
```

Running `mvn install` will:

- Validate → Compile → Run JUnit tests → Package into WAR → Install in local repo.

Real-World Example:

- **E-commerce application** – Maven ensures code is compiled, tested, packaged into a WAR file, and deployed to Tomcat.
- Developers just run:

  ```
  mvn clean install
  ```

  and Maven handles everything.

Conclusion:

Maven's build lifecycle standardizes software builds, improves automation, and ensures reliable deployments. By handling compile → test → package → deploy seamlessly, Maven eliminates manual steps and reduces human error in enterprise applications.

Q2. Describe in detail Maven plugins, creation of artifacts, and how dependency management is handled.

Introduction:

Maven's real power lies in **plugins** and **dependency management**. Plugins extend Maven's lifecycle, while dependency management ensures all external

libraries are available with correct versions.

---

1. Maven Plugins

- A plugin is a collection of goals executed in phases.
- Without plugins, Maven cannot perform tasks.
- **Types:**
  - **Build Plugins** → Used during the build process (compiler, surefire).
  - **Reporting Plugins** → Used for generating reports (Surefire report, Javadoc).

Examples of Plugins:

- **maven-compiler-plugin** – Compiles Java code.
- **maven-surefire-plugin** – Runs unit tests.
- **maven-jar-plugin** – Packages JAR files.

Diagram – Maven with Plugins:

```
Lifecycle → Phases → Plugin Goals → Execution
```

---

2. Artifact Creation in Maven

- An **artifact** is the output of a Maven project (JAR, WAR, EAR, ZIP).
- Defined by **coordinates** (groupId, artifactId, version).
- Example – Packaging a JAR:

```
<packaging>jar</packaging>
```

- Example – Packaging a WAR:

```
<packaging>war</packaging>
```

Running `mvn package` creates the artifact inside the `/target` directory.

---

3. Dependency Management

- Dependencies declared in `pom.xml`.
- Maven downloads from local/central/remote repositories.
- Supports **transitive dependencies** (A → B → C).
- Supports **scopes** (compile, test, provided, runtime).

Example – Dependency Declaration:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>5.3.20</version>
</dependency>
```

Diagram – Dependency Resolution:

```
POM.xml → Local Repo → Central Repo → Remote Repo
```

Real-World Example:

- A Spring Boot project uses 20+ dependencies. Maven handles them automatically.
- Developers only declare a few, Maven resolves the rest.

Conclusion:

Maven plugins automate build tasks, artifacts package applications, and dependency management ensures reliable builds. Together, they make Maven a complete DevOps-friendly build tool.

Q3. Discuss the installation of Gradle and explain how to build a project using Gradle step by step with an example.

Introduction:

Gradle is an advanced build tool used in Java, Android, and enterprise projects. It combines flexibility (Groovy/Kotlin DSL) with speed (incremental builds).

---

1. Installation of Gradle:

- **Step 1:** Install **Java JDK** (Gradle requires Java).
- **Step 2:** Download Gradle from [gradle.org].
- **Step 3:** Extract and set environment variables:
  - `GRADLE_HOME` → Gradle folder.
  - Add to PATH.
- **Step 4:** Verify installation:

```
gradle -v
```

---

2. Creating a Gradle Project:

- Create a folder → Run `gradle init`.
- Generates files: `build.gradle`, `settings.gradle`.

Example build.gradle:

```
plugins {
    id 'java'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework:spring-core:5.3.20'
    testImplementation 'junit:junit:4.12'
}
```

---

3. Build Process in Gradle:

- **Step 1:** Write source code in `/src/main/java`.
- **Step 2:** Run `gradle build`.

- **Step 3:** Gradle compiles, tests, packages into a JAR.

Diagram – Gradle Workflow:

```
Source Code → Gradle Build Script → Tasks → Artifacts
```

Real-World Example:

- Android Studio uses Gradle as the default build system.
- Every APK is built and tested via Gradle tasks (`assembleDebug`, `assembleRelease`).

Conclusion:

Gradle installation and project build are simple and powerful. Its flexibility and speed make it the preferred choice for Android and enterprise applications.

Q4. Explain Jenkins architecture in detail with a neat diagram.

Introduction:

Jenkins follows a distributed **Master-Agent** architecture that allows multiple builds across platforms, enabling scalability and parallel execution.

---

1. Jenkins Master (Controller):

- Provides the web interface.
- Handles job scheduling.
- Assigns jobs to agents.

2. Jenkins Agents (Workers):

- Execute jobs on different machines.
- Support multiple OS (Linux, Windows).
- Report back results to the master.

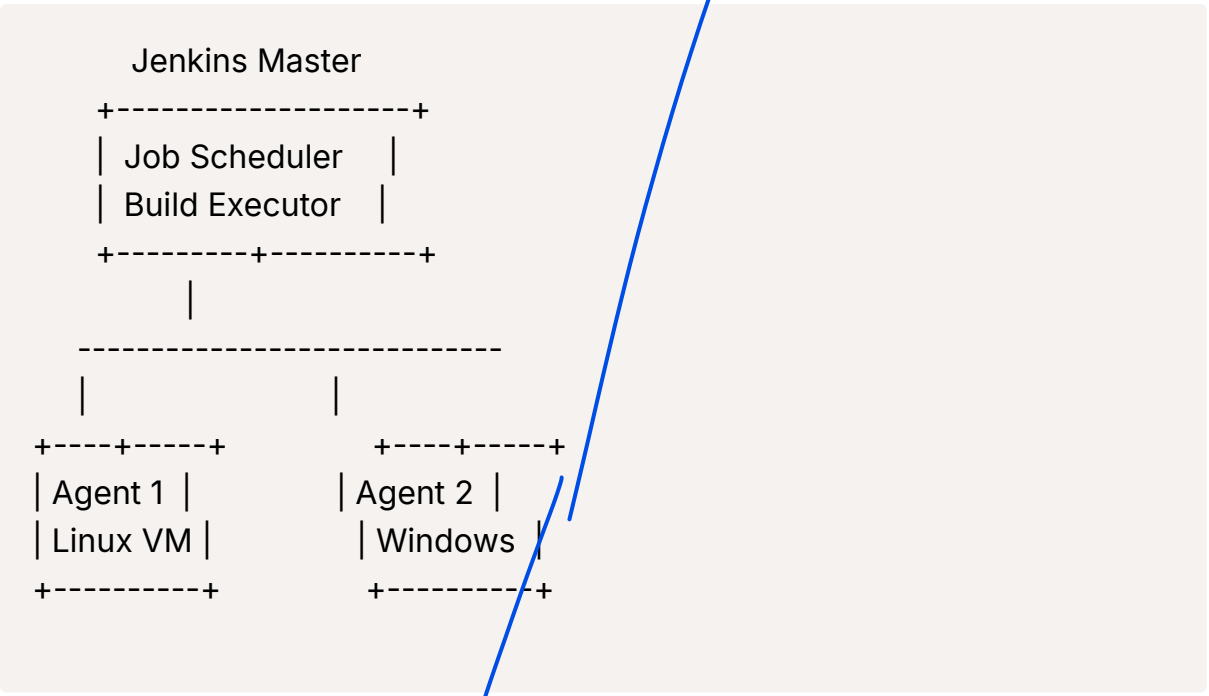
3. Communication:

- Secure channel (SSH, JNLP, WebSocket).
- Master controls execution.

4. Plugins:

- Extend Jenkins functionality.
- Integrate tools like Git, Maven, Docker.

Diagram – Jenkins Architecture:



Real-World Example:

- A company runs Jenkins master on a server, with agents on AWS EC2 for Linux builds and Windows agents for .NET builds.

Conclusion:

Jenkins architecture ensures efficient CI/CD with distributed builds, making it scalable and enterprise-ready.

Q5. Discuss step-by-step installation, configuration, and job creation in Jenkins, including integration with Git and Maven.

Introduction:

Jenkins setup involves installation, configuring tools, and integrating with Git and Maven for automated builds.

Step 1 – Installation:

- Download Jenkins WAR or installer.
- Start with:

```
java -jar jenkins.war
```

Step 2 – Unlock Jenkins:

- Use the initial admin password from `.jenkins/secrets/`.
- Install suggested plugins.

Step 3 – Configure Tools:

- Add JDK, Git, and Maven paths in *Global Tool Configuration*.

Step 4 – Create a Job:

- New Item → Freestyle or Pipeline.
- Configure SCM → Git repository.
- Build Step → `mvn clean install`.

Step 5 – Post-Build Actions:

- Archive artifacts.
- Publish reports.

Diagram – Jenkins CI/CD Workflow:

```
Git Commit → Jenkins → Build (Maven) → Test → Deploy
```

Conclusion:

Jenkins integrates seamlessly with Git and Maven, enabling fully automated pipelines from code commit to deployment.

Q6. Explain commonly used Jenkins plugins (Git Plugin, Parameter Plugin, HTML Publisher, Copy Artifact, Extended Choice Parameters) with examples.

Introduction:

Plugins extend Jenkins functionality, making it adaptable to diverse project needs.

---

1. Git Plugin:

- Integrates with Git.
- Example: Clone repo `https://github.com/project.git`.

2. Parameter Plugin:

- Adds build parameters.
- Example: String parameter for selecting build environment.

3. HTML Publisher Plugin:

- Publishes HTML test reports.
- Example: JUnit report shown in Jenkins UI.

4. Copy Artifact Plugin:

- Copies artifacts from one job to another.
- Useful in multi-stage pipelines.

5. Extended Choice Parameter Plugin:

- Provides advanced parameters like multi-select checkboxes.

Diagram – Jenkins Plugins Integration:

```
[Git Plugin] → Source Code
[Parameter Plugin] → Dynamic Builds
[HTML Publisher] → Reports
[Copy Artifact] → Reuse Artifacts
```

Conclusion:

Plugins are the backbone of Jenkins, allowing it to adapt to any CI/CD workflow.

Q7. Write a detailed note on Continuous Integration using Jenkins with a real-world workflow example.

Introduction:

Continuous Integration (CI) is the practice of merging code changes frequently and testing them automatically. Jenkins is the most widely used CI server.

Jenkins CI Workflow:

1. Developer pushes code to Git.
2. Git webhook triggers Jenkins job.
3. Jenkins pulls latest code.
4. Jenkins builds (Maven/Gradle).
5. Jenkins runs unit & integration tests.
6. Reports are published.
7. If successful → deploy.

Diagram – CI Workflow:

```
Developer → Git → Jenkins → Build & Test → Deploy → Feedback
```

Real-World Example:

- **E-commerce app pipeline:**
  - GitHub push triggers Jenkins.
  - Maven compiles and tests code.
  - Docker image built.
  - Deployed on Kubernetes.

Conclusion:

Jenkins makes CI possible by automating build, test, and deployment pipelines, ensuring faster delivery and higher code quality.

Q8. How do Jenkins Builds work? Explain Jenkins workspace, build triggers, and build execution process with an example.

Introduction:

A Jenkins build is the execution of a job configured in Jenkins. It involves source checkout, build execution, testing, and reporting.

---

1. Jenkins Workspace:

- Each job gets a dedicated workspace.

- Stores source code, configuration, and build output.

2. Build Triggers:

- **Manual** – Triggered by user.
- **SCM Polling** – Jenkins polls Git for changes.
- **Webhooks** – Trigger builds on Git push.
- **Scheduled** – CRON jobs.

3. Build Execution Process:

1. Jenkins checks out source into workspace.
2. Executes build steps (Maven/Gradle).
3. Runs unit/integration tests.
4. Publishes artifacts and reports.

Diagram – Build Flow:

```
Trigger → Workspace → Build Execution → Reports → Artifacts
```

Example:

- Job configured with Git + Maven.
- Trigger: Git commit.
- Execution: `mvn clean test package`.
- Result: JAR file + test reports.

Conclusion:

Jenkins builds automate the entire software pipeline, ensuring consistency, speed, and reliability in enterprise development.