Session: Understanding Software Engineering Workflows – SDLC, Agile, and DevOps

---

Software Process Models

1. Software Process Models – Overview

A **software process model** defines the **structured sequence of activities** involved in the Software Development Life Cycle (SDLC), such as requirement analysis, design, development, testing, deployment, and maintenance. It acts as a **framework** that guides how a software project is planned, executed, monitored, and delivered.

Different projects have different characteristics—such as requirement stability, project size, risk level, customer involvement, and technology complexity. Because of this, **no single process model fits all projects**. Choosing the right process model is critical for achieving **quality, cost control, timely delivery, and customer satisfaction**.

---

2. Waterfall Model

The **Waterfall model** is a **linear and sequential** software development process where each phase must be completed before the next phase begins. The phases typically include:

1. Requirement Analysis
2. System Design
3. Implementation (Coding)
4. Testing
5. Deployment
6. Maintenance

In this model, progress flows in one direction—**like a waterfall**—with very little overlap between phases. Documentation is created extensively at each stage, and changes are discouraged once a phase is completed.

Advantages

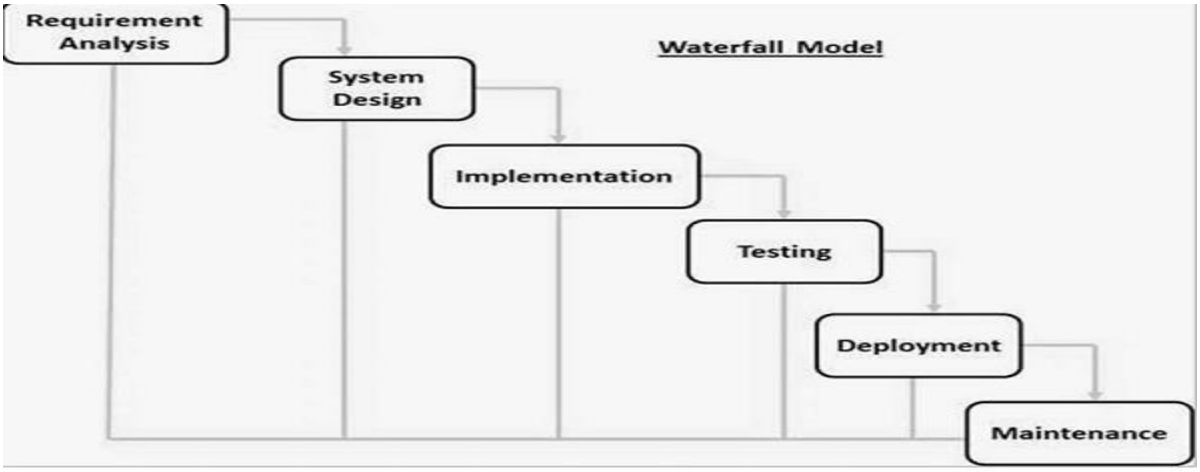
- Simple and easy to understand
- Well-structured with clear milestones
- Strong documentation and control
- Suitable for projects with stable and well-defined requirements

Limitations

- No working software until late in the lifecycle
- Very costly to accommodate changes
- Late user feedback
- High risk if requirements are misunderstood

When to Use

The Waterfall model is best suited for **government, defense, banking, and compliance-driven projects** where requirements are fixed and documentation is mandatory.



---

3. Incremental Model

The **Incremental model** divides the system into **small, manageable increments**, where each increment delivers a **working version of the software** with added functionality. Each increment follows its own mini-SDLC cycle: requirements → design → coding → testing.

The first increment delivers core features, and subsequent increments add enhancements until the complete system is built. Users can start using the software early while development continues.

Advantages

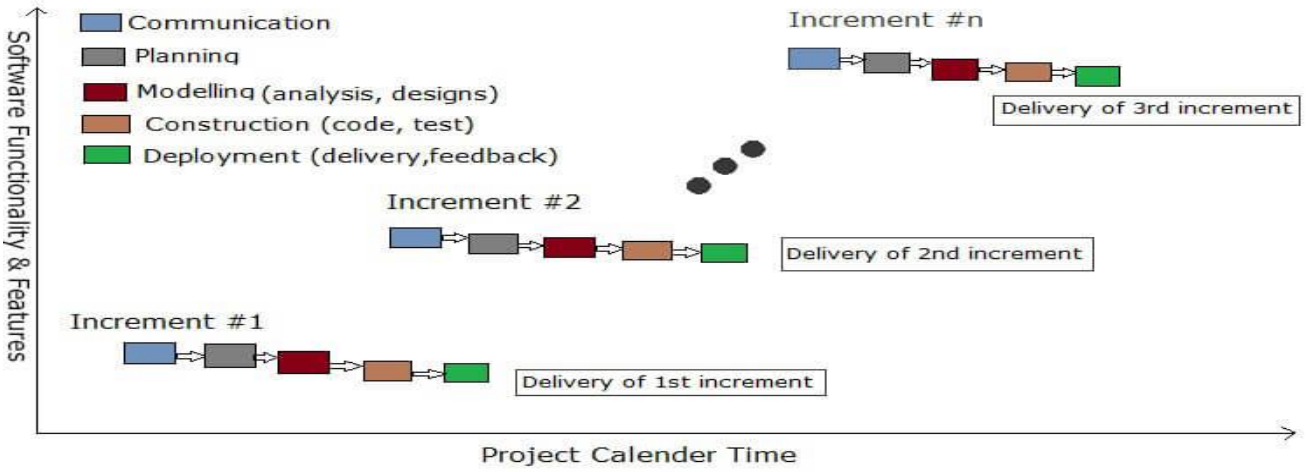
- Early delivery of working software
- Reduced risk compared to Waterfall
- Easier testing and debugging
- User feedback can be incorporated in later increments

Limitations

- Requires good planning and modular design
- Integration of increments can be complex
- Overall system architecture must be well-defined upfront

When to Use

The Incremental model is ideal for **business applications, enterprise systems, and projects where early release and gradual enhancement are important**.



---

4. Prototyping Model

The **Prototyping model** is used when **requirements are unclear, incomplete, or likely to change**. Instead of waiting for full requirement clarity, developers build a **quick prototype** (a working mock-up) to help users visualize the system.
Users interact with the prototype and provide feedback, which helps refine requirements. The prototype may either be discarded (throwaway prototype) or evolved into the final system (evolutionary prototype).

Advantages

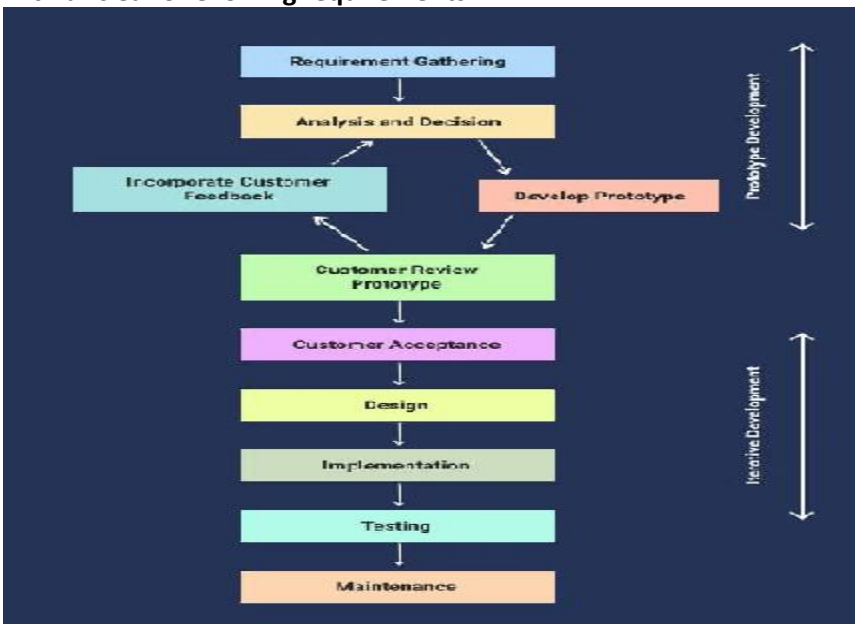
- Improves requirement understanding
- Enhances user involvement and satisfaction
- Reduces misunderstandings early
- Useful for UI-intensive systems

Limitations

- Poorly designed prototypes may become final systems
- Can increase development cost if uncontrolled
- May lead to unrealistic user expectations

When to Use

The Prototyping model is best suited for **user-interface-heavy applications, interactive systems, and projects with unclear or evolving requirements**.



---

5. Spiral Model

The **Spiral model** is a **risk-driven, iterative process model** that combines elements of Waterfall and Prototyping. Development proceeds in a series of **spiral loops**, and each loop represents a phase of the project.
Each spiral iteration includes four key activities:

1. Planning
2. Risk Analysis
3. Engineering (development & testing)
4. Evaluation

The core strength of the Spiral model is **early identification and mitigation of risks**, making it highly suitable for complex systems.

Advantages

- Excellent risk management
- Flexible and iterative

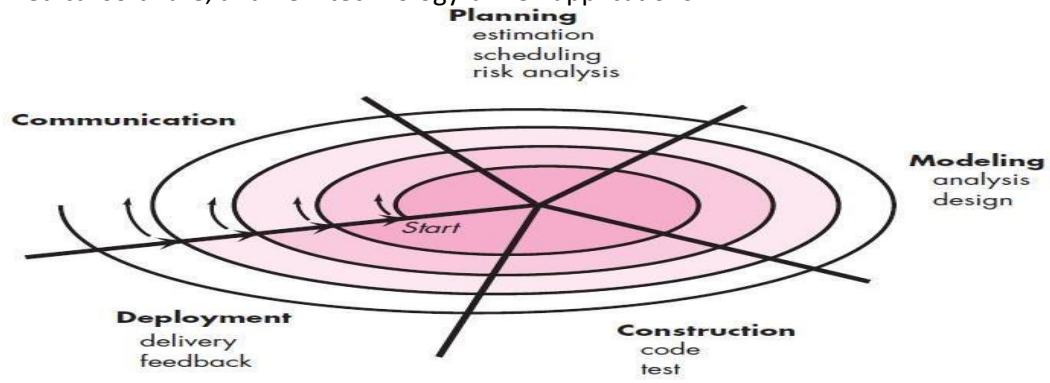
- Early detection of issues
- Suitable for large and complex projects

Limitations

- Requires expertise in risk analysis
- Expensive and time-consuming
- Not suitable for small or low-risk projects

When to Use

The Spiral model is ideal for **large-scale, mission-critical, and high-risk projects**, such as aerospace systems, medical software, and new technology-driven applications.



---

6. Agile Model

The **Agile model** is a **modern, iterative, and incremental approach** that emphasizes **flexibility, customer collaboration, and rapid delivery**. Instead of long development cycles, Agile uses **short iterations (sprints)** that deliver small but usable pieces of software.

Agile promotes continuous communication between developers and customers, frequent testing, and quick response to change. Popular Agile frameworks include **Scrum, Kanban, and XP**.

Core Principles

- Working software over documentation
- Customer collaboration over contract negotiation
- Responding to change over following a rigid plan

Advantages

- Fast delivery and early value
- High customer satisfaction
- Easy to accommodate changing requirements
- Continuous testing and improvement

Limitations

- Requires strong team collaboration
- Less documentation
- Difficult to scale without experience

When to Use

Agile is best suited for **web applications, mobile apps, cloud systems, startups, and projects with rapidly changing requirements**.

---

7. Comparative Summary (Exam-Friendly)

Model	Best For	Key Strength	Main Limitation
Waterfall	Stable requirements	Simplicity & documentation	No flexibility
Incremental	Early delivery	Risk reduction	Integration complexity
Prototyping	Unclear requirements	Better requirement clarity	Poor design risk

Model	Best For	Key Strength	Main Limitation
Spiral	High-risk projects	Risk management	High cost
Agile	Dynamic environments	Flexibility & speed	Needs discipline

---

Agile Software Development

Agile Software Development is an **iterative and flexible approach** that emphasizes **continuous delivery of working software, collaboration, and quick adaptation to change**. It works well for projects with **changing or unclear requirements**.

Agile is implemented using **frameworks** that structure Agile principles:

- **Scrum:** Organizes development into short **sprints** (2–4 weeks). Defines roles like **Product Owner, Scrum Master, and Development Team**. Uses **Sprint Planning, Daily Stand-up, Sprint Review, and Retrospective** for transparency and continuous improvement. Ideal for **complex projects needing frequent feedback**.
- **Kanban: Visual workflow-based** framework focusing on **continuous delivery**. Work items are shown on a **Kanban board**, and **WIP limits** improve efficiency. Commonly used in **maintenance, support, and DevOps**.
- **Extreme Programming (XP):** Focuses on **technical excellence and high-quality code**. Promotes **TDD, pair programming, and continuous integration**. Suitable for projects with **frequent requirement changes**.
- **Lean Software Development:** Applies **Lean principles** to eliminate waste, improve flow, and deliver value quickly. Often used to **increase process efficiency**.

Conclusion: Agile frameworks help teams **deliver high-quality software faster**, remain **flexible**, and stay **customer-focused**, making Agile the preferred choice in modern software development.

Scrum Framework

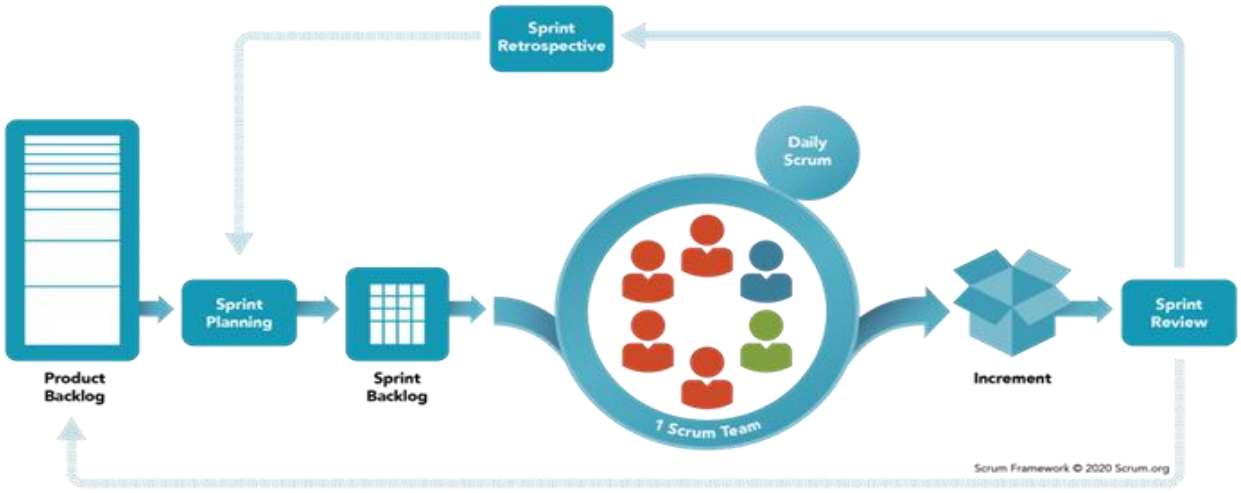
The **Scrum procedure** begins with the **Product Backlog**, which is a complete, prioritized list of software requirements written as **user stories** and maintained by the **Product Owner**. The Product Backlog is a **living artifact**, continuously updated as business needs change and feedback is received from stakeholders. From this backlog, the team conducts **Sprint Planning**, selecting a set of high-priority items that can realistically be completed within the sprint. These items form the **Sprint Backlog**, and a **Sprint Goal** is defined to ensure shared understanding and commitment among team members.

During **Sprint Execution**, which typically lasts **two to four weeks**, the Development Team designs, codes, tests, and integrates the selected features. The sprint scope remains **fixed** to maintain focus, and Agile engineering practices are followed to ensure **quality and continuous progress**. Every day, the team conducts a **Daily Scrum (Stand-up)**, a short meeting where members discuss what they completed yesterday, what they

plan to do today, and any obstacles blocking their work. This practice ensures **transparency, coordination, and quick resolution of issues**.

At the end of the sprint, the team delivers a **potentially shippable product increment**, which is reviewed during the **Sprint Review**. Stakeholders provide feedback on the completed work, and new requirements or changes may be added to the Product Backlog, maintaining continuous alignment with business goals. Finally, the **Sprint Retrospective** allows the team to reflect on the sprint process itself, evaluating what went well, what did not, and how improvements can be made in the next sprint.

The Scrum process then repeats in an **iterative cycle**: **Product Backlog → Sprint Planning → Sprint → Increment → Review → Retrospective**, enabling teams to deliver high-quality software in a **controlled, flexible, and feedback-driven manner**.



DevOps

DevOps is a modern **software engineering practice** that focuses on **integrating Development (Dev) and Operations (Ops)** to enable **faster, reliable, and continuous software delivery**. In traditional environments, development teams wrote code while operations teams handled deployment and maintenance, often working in isolation. This separation caused **delays, deployment failures, and poor communication**. DevOps emerged to eliminate these issues by promoting **collaboration, automation, and shared responsibility** across the entire **Software Development Life Cycle (SDLC)**.

The **need for DevOps** arises from the limitations of both traditional and Agile approaches when software must be delivered **frequently and at scale**. While Agile improved development speed through **iterative releases**, it often stopped at the development or testing phase. Deployment and monitoring were still largely manual, slow, and error-prone. DevOps extends Agile principles beyond development by introducing

automation, continuous integration, continuous delivery, and continuous monitoring, ensuring that software moves smoothly from **code to production**.

In real-world practice, DevOps uses **automated pipelines** to build, test, deploy, and monitor applications continuously. This reduces **human error**, improves **release frequency**, and ensures **system stability**. Feedback from production systems is continuously collected and shared with development teams, enabling **faster issue resolution and continuous improvement**. DevOps is especially critical for **cloud-based, microservices, and large-scale applications**, where frequent updates and high availability are mandatory.

Difference Between Traditional, Agile, and DevOps

Traditional Software Development (Waterfall Model) follows a **linear and sequential process**, where each phase such as requirements, design, development, testing, and deployment is completed one after another. Changes are difficult to accommodate once a phase is finished, and the software is delivered only at the end of the project. Deployment is mostly manual, making releases **slow and risky**, and this model works best when requirements are **stable and clearly defined**.

Agile Software Development uses an **iterative and incremental approach**, delivering software in small cycles with frequent customer feedback. It improves **flexibility and development speed**, allowing teams to handle changing requirements effectively. However, Agile mainly focuses on **development and testing**, while deployment and operations are often managed separately, limiting full continuous delivery.

DevOps extends Agile by integrating **development, testing, deployment, operations, and monitoring** into a single automated workflow. It emphasizes **CI/CD and continuous monitoring**, enabling frequent and reliable releases, faster recovery from failures, and improved system performance. DevOps represents both a **cultural and technical shift**, with shared responsibility from development to production.

Legacy Software

Legacy software consists of systems built on **older technologies** that still support **critical business functions**, especially in areas like **banking, healthcare, government, and insurance**. These systems remain in use because they are **stable, reliable, and deeply integrated** into business processes.

Replacing legacy systems is difficult due to **high cost, risk, and operational disruption**, so organizations prefer to **maintain and modernize** them. Modern software engineering enables this through **APIs, system wrapping, and partial migration**, allowing legacy systems to work with new technologies while preserving essential business logic.

Session: Source Code Management with Git & GitHub – From Local Repos to Team Collaboration

---

What are Requirements in Software Engineering?

Requirements define **what the system should do** and **how well it should perform**.
They act as a **contract between stakeholders and developers** and serve as the foundation for **design, development, testing, and maintenance**.

☞ remember:
"Incorrect requirements lead to project failure, even with perfect coding."

Functional Requirements (FR)

◈ **Definition**

Functional requirements describe the specific functions, features, or services the system must provide.

◈ **What they answer:**

- **What should the system do?**
- **How should it respond to inputs?**
- **What operations should it perform?**

◈ **Examples**

- User should be able to **register and log in**
- System should **generate monthly reports**
- Admin can **approve or reject requests**
- ATM should **dispense cash after PIN validation**

Non-Functional Requirements (NFR)

◈ **Definition**

Non-functional requirements define quality attributes of the system rather than functionality.

◈ **What they answer:**

- **How well should the system perform?**
- **Under what constraints should it operate?**

◈ **Common Types (VERY IMPORTANT for interviews)**

Category	**Example**
Performance	Response time < 2 seconds
Scalability	Support 10,000 concurrent users
Security	Data encrypted using AES
Reliability	99.9% uptime
Usability	Easy navigation, user-friendly UI
Maintainability	Modular, easy to update
Portability	Runs on Windows & Linux

Software Requirements Specification (SRS)

◈ **What is SRS?**

An **SRS document** is a **formal, structured document** that clearly describes:

- Functional requirements
- Non-functional requirements
- System constraints
- Interfaces and assumptions

☞ It acts as a **single source of truth** for the entire project.

◈ **Importance of SRS (Interview Gold Points)**

- Reduces **ambiguity**

- Helps in **cost and time estimation**
- Acts as reference for **design, coding, testing**
- Prevents **scope creep**
- Legal agreement between client and vendor

🔹 **Typical SRS Structure (IEEE Style)**

1. Introduction
2. Overall Description
3. Functional Requirements
4. Non-Functional Requirements
5. System Interfaces
6. Constraints & Assumptions
7. Use Cases / Diagrams (optional)

🔹 **Qualities of a Good SRS (VERY IMPORTANT)**

A good SRS should be: **Correct, Complete, Unambiguous, Consistent, Verifiable, Modifiable, Traceable**

Requirements Engineering Process (Short & Interview Focused)

🔹 **1. Feasibility Study**

Checks whether the project is: Technically feasible, Economically viable, Operationally possible

🔹 **2. Requirements Elicitation**

Process of **gathering requirements** from stakeholders using: Interviews, Questionnaires, Workshops, Observation

---

Source Code Management (SCM) and Version Control Systems

hhnnk Below is the **same content rewritten in a shorter, crisp, exam-ready format**, while **preserving key ideas** and using **clear points**.
This is suitable for **quick revision, 5–8 mark answers, interviews, and notes**.

Source Code Management (SCM) and Version Control Systems

Source Code Management (SCM) is a fundamental practice in software engineering that helps teams **track, manage, and control changes** made to source code over time. In real-world projects where multiple developers work on the same codebase, SCM enables **collaboration, version tracking, rollback, and parallel development**. It ensures **code safety, accountability, consistency**, and helps maintain **high-quality software delivery** across different environments.

Version Control Systems (VCS)

Version Control Systems (VCS) are tools used to implement SCM. They allow developers to **store multiple versions of code**, track changes, compare versions, and restore previous states when needed.

Types of Version Control Systems

Centralized Version Control Systems (CVCS)

- Uses a **central server** to store all versions of the code
- Developers **check out and commit** changes to the central repository
- **Examples:** SVN, CVS
- **Pros:** Simple to understand, centralized control
- **Cons:** Single point of failure, requires network connection

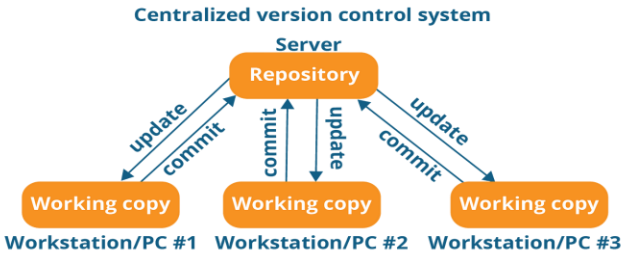
---

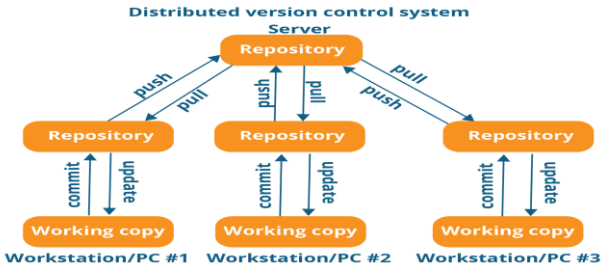
Distributed Version Control Systems (DVCS)

- Each developer has a **complete copy** of the repository with full history
- Supports **offline work** and later merging
- **Examples:** Git, Mercurial
- **Pros:** Parallel development, strong branching and merging, high reliability
- **DVCS (especially Git) is the industry standard**

Git – Most Popular SCM Tool

Git is the most widely used SCM tool because it supports **distributed workflows**, offers **high performance**, and provides **powerful branching and merging** features. It enables teams to work independently and safely merge changes, making it ideal for **modern collaborative software development**.





Git as a Distributed Version Control System

Git is a **distributed version control system (DVCS)**, meaning every developer has a **complete local copy** of the repository and its history. This architecture allows developers to work **offline**, experiment safely, and commit changes locally before sharing them. Git efficiently tracks file changes and enables **safe merging of work** from multiple developers. Commonly used commands such as **git init**, **git add**, **git commit**, and **git log** help manage the project lifecycle locally. Because of its speed, reliability, and flexibility, Git is widely adopted in **Agile and DevOps environments**.

Difference Between Git and GitHub

Git and **GitHub** serve different but complementary roles. **Git** is a **command-line tool** that manages version control on a local system, while **GitHub** is a **cloud-based hosting platform** for Git repositories. GitHub enables **remote collaboration**, **code reviews**, and **access control** through features like pull requests and issues.

Developers use **git clone** to copy repositories from GitHub, **git push** to upload changes, and **git pull** to synchronize updates. In practice, Git handles version control, and GitHub enables **team collaboration and DevOps integration**. Think of it like this:

- **Git** → A *tool* that tracks changes in your files
- **GitHub** → A *platform* that stores Git projects online and enables collaboration

GitHub is a **cloud-based hosting service for Git repositories** that provides:

- Online backup of code
- Collaboration tools (Pull Requests, Issues)
- Open-source contribution workflows

Real-World Analogy:
Git = Notebook
GitHub = Google Drive

Initial Git Configuration

Before making commits, Git must be configured with the developer's identity. This ensures every change is traceable to an individual. The configuration is done using

Configure Username & Email (One-Time Setup)

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

✓ This information appears in **every commit** you make.
⚠ If someone else configured Git earlier, you can later **unset** it using:

```
git config --global --unset user.name
git config --global --unset user.email
```

Proper configuration is critical in professional environments for **audit trails and accountability**.

Git Local Workflow and Repository Management

Git follows a **three-stage workflow** consisting of the Git follows a **3-step workflow**:
Working Directory → Staging Area → local Repository.
Developers modify files in the working directory, prepare selected changes using **git add**, and permanently store them using **git commit -m "message"**. The status of files can be checked using **git status**, while **git diff** helps review changes before staging. This structured workflow provides precise control over code changes.
Initialize a New Project
Create project folder and initialize Git.

```
mkdir ecommerce-app
cd ecommerce-app
git init
```

Converts folder into a **Git repository**

Add Files to Staging Area

Create a file:

touch index.html

Check status:

git status

Add files:

git add index.html # single file

git add . # all files

📂 Working Directory → 📄 Staging Area

Commit Changes

Save staged changes permanently.

git commit -m "Initial project setup" 📜 Commit = snapshot of your code

---

View Commit History

git log

git log –oneline ✓ Used for auditing & rollback

Change Commit Message

If the message is wrong:

git commit --amend -m "Updated initial project structure" 📌 Modifies **latest commit only**

Create & Work with Branches

Create a feature branch:

git branch feature-login

git checkout feature-login

OR shortcut:

git checkout -b feature-login ✓ Enables parallel development ✓ Keeps main branch stable

---

Merge Branch into Main

Switch to main:

git checkout main

Merge:

git merge feature-login ✓ Combines feature work into main

Undo & Restore Changes

Discard file changes:

git restore file.txt

Unstage file:

git restore --staged file.txt ✓ Safe undo without deleting commits

---

Revert a Commit (Safe Undo)

Undo a committed change safely:

git revert <commit_hash> ✓ Creates new commit ✓ Safe for shared branches

---

Branching & Merge Conflicts

Branches allow **parallel development**, safe feature implementation, and isolated bug fixes without affecting the main code

◈ **Benefits of Branching**

- Feature development
- Bug fixes
- Safe experimentation
- Team collaboration

◈ **Create & Switch Branch** git checkout -b feature-branch

◈ **Merge Branch** git merge feature-branch

⚠ **What is a Merge Conflict?**

Answer:

A merge conflict occurs when:

- Two branches modify the **same file**
- Git cannot decide which change to keep

Git actually does during a conflict

- Git **pauses the merge process**
- Marks the conflicting sections in the file with special symbols:

resolve a merge conflict

1. Open the conflicted file
2. Manually choose or combine the correct changes
3. Remove the conflict markers
4. Stage the resolved file:
5. git add file.txt

git commit -m "Resolved merge conflict"

Importance in real projects

Merge conflicts ensure that **no code is overwritten silently**, forcing developers to review and resolve differences carefully, which maintains code correctness and stability.

Using .gitignore for Clean Repositories

In real-world projects, certain files should not be tracked by Git. The **.gitignore** file specifies files and directories that Git must ignore, such as build outputs and environment files. After creating the .gitignore file, it is added and committed using **git add .gitignore** and **git commit**. This practice prevents accidental exposure of sensitive data and keeps repositories clean.

Use .gitignore to exclude files

*.gitignore specifies files/folders Git should **ignore** and not track.*

Example .gitignore for Node.js:

node_modules/ dist/ .env *.log

Common commands:

Check which files are ignored

git status --ignored

```
# Add .gitignore to repository
git add .gitignore
git commit -m "Add .gitignore"
```

💡 **Tip:** Always add .gitignore **before committing** to avoid accidentally tracking sensitive or unnecessary files

---

Working with Remote Repositories (GitHub)

To collaborate with others, a local repository is connected to a **remote repository** on GitHub using **git remote add origin <url>**. The connection can be verified using **git remote -v**. Code is uploaded using **git push -u origin main**, and updates from others are retrieved using **git pull origin main**. For safer workflows, **git fetch** is used to review changes before merging.

✅ **Quick Command Summary**

```
cd my-project
git init
git add .
git commit -m "Initial commit"
git remote add origin https://github.com/your-username/my-project.git
git push -u origin main
```

Undoing Changes and Correcting Mistakes

Git provides multiple ways to undo mistakes safely. Files can be unstaged using **git restore --staged file.txt**, and local changes can be discarded using **git checkout -- file.txt**. Commits can be safely undone using **git revert <commit_hash>**, which is preferred in shared repositories. The most recent commit can be modified using **git commit --amend**, allowing updates to commit messages or included files.

Cherry-Picking Specific Commits

Cherry-picking is used to apply a specific commit from one branch to another using **git cherry-pick <commit_hash>**. This is especially useful for **hotfixes** in production environments, where merging an entire branch is unnecessary or risky.

GitHub Collaboration Workflow

GitHub supports team collaboration by enabling developers to clone repositories, create feature branches, commit changes, and push them back to the remote repository. Pull requests are used to review and merge code changes safely. Regular synchronization using **git pull** ensures all team members stay aligned.

`git fetch origin` ✓ Downloads changes ✗ Does not merge

git pull origin main ✓ Fetch + Merge

📌 DevOps Rule
✓ Fetch for production
✓ Pull for development

Git Patches and Restricted Workflows

In environments where direct push access is not allowed, Git patches are used to share changes. Patch files are generated using **git format-patch** and applied using **git apply** or **git am**. This workflow is common in **open-source and regulated projects**.

Generate Patch Files

📦 Git Patch – Short Notes

- **Generate patch files**

```
git format-patch origin/main
```

Creates files like:
0001-fix-issue.patch

- **Latest commit**

```
git format-patch -1 HEAD
```

- **Last N commits**

```
git format-patch -N
```

- **Share patch**
  Email / file share / issue comment
- **Apply patch**

```
git am file.patch      # apply with commit
git apply file.patch   # apply without commit
```

👉 **Used when push access is not available (open-source / restricted repos).**

Advanced Git Commands for DevOps Engineers

DevOps engineers frequently use **git fetch** for safe updates, **git reset** to undo commits carefully, **git rebase** to maintain a clean history, and **git stash** to temporarily save work. These commands help manage **high-frequency commits** and **production stability** in CI/CD pipelines.

From Concept to Deployment – Software Design, Architecture, UML, and CI/CD Pipelines

---

Design Engineering

Design Engineering is a core activity in software engineering that transforms **software requirements into a detailed blueprint** for construction and implementation. It acts as a bridge between **requirements analysis**

and coding and ensures that the final system satisfies both **functional and non-functional requirements**. A good design focuses on **correctness, efficiency, maintainability, scalability, and reliability** while minimizing complexity and risk.

The Design Process and Design Quality

The **design process** is a systematic sequence of activities that converts requirements into a structured representation of the software system. It begins with understanding the problem and proceeds through architectural design, interface design, data design, and component-level design. During this process, design decisions are made to balance constraints related to **performance, cost, schedule, and quality**.

Design quality refers to how well the design meets the stated and implied requirements of the system. A high-quality design is easy to understand, flexible to change, reusable, and efficient. Design quality is often evaluated using the **FURPS quality model**, which ensures that both functional and non-functional attributes are addressed.

FURPS and Design Quality

- **Functionality**: Completeness and correctness of features
- **Usability**: Ease of use and user interaction
- **Reliability**: Fault tolerance and availability
- **Performance**: Response time, throughput, resource usage
- **Supportability**: Maintainability, adaptability, scalability

By incorporating FURPS during the design process, designers can proactively reduce quality-related risks and ensure a balanced design.

Design Concepts

Design concepts are fundamental principles that guide software designers in creating effective and manageable systems. These concepts help in controlling complexity and improving clarity.

Important Design Concepts

- **Abstraction**: Focuses on essential features while hiding unnecessary details.
- **Modularity**: Divides the system into smaller, manageable modules.
- **Information Hiding**: Internal details of a module are hidden from others.
- **Separation of Concerns**: Different aspects of the system are handled independently.
- **Refinement**: Gradual elaboration of the design from high level to low level.
- **Functional Independence**: Each module performs a single well-defined task.

Applying these concepts leads to designs that are **robust, reusable, and easy to maintain**.

The Design Model

The **design model** is a collection of representations that describe the software from different perspectives. It provides a **blueprint for implementation** and helps stakeholders understand how the system will be built.

Elements of the Design Model

- **Architectural Design**: Overall system structure
- **Data Design**: Data structures and databases
- **Interface Design**: User and component interfaces
- **Component-Level Design**: Internal logic of components

The design model ensures traceability from requirements to implementation and supports quality evaluation using models such as FURPS.

Architectural Design

Architectural design defines the **overall structure of the software system**, including its components, their relationships, and interactions. It establishes the system's foundation and significantly influences **performance, reliability, scalability, and maintainability**. Architectural decisions are critical because they are difficult to change later and directly affect the success of the project.

Software Architecture

Software architecture refers to the high-level organization of a system, including architectural components, connectors, and constraints. It provides a conceptual model that guides detailed design and implementation. A well-defined architecture supports **reuse, ease of maintenance, and efficient communication** among development teams.

Architectural Styles and Patterns

Architectural styles and patterns are proven solutions to common design problems. They define how components are organized and how they communicate.

Common Architectural Styles

- **Layered Architecture**: System organized into layers with specific responsibilities. Ex:- A **web application** with presentation layer, business logic layer, and database layer.
- **Client–Server Architecture**: Clients request services from servers.ex:- A **web browser (client)** accessing a **web server** to retrieve web pages.
- **Pipe-and-Filter Architecture**: Data flows through a series of processing components.ex:- A **compiler**, where source code passes through lexical analysis, syntax analysis, and code generation stages.
- **Event-Driven Architecture**: Components respond to events asynchronously.Ex:- An **online shopping system** where placing an order triggers payment, inventory update, and notification events.

- **Microservices Architecture**: System composed of small, independent services. Ex:-A **food delivery application**, where order management, payment, and delivery tracking run as separate services.

UML (Unified Modeling Language)

UML (Unified Modeling Language) is a **standardized visual modeling language** used to **specify, visualize, construct, and document** software systems. It helps developers and stakeholders understand both the **structure and behavior** of a system using graphical notations.

Basic Building Blocks of UML

The basic building blocks of UML consist of **Things, Relationships, and Diagrams**. These elements together form the foundation for modeling a software system.

UML Things

Things are the **basic modeling elements** in UML and represent real-world or conceptual entities.
Types of UML Things

- **Structural Things**: Represent static parts of the system
  Example: Class, Interface, Component, Node
- **Behavioral Things**: Represent dynamic behavior
  Example: Interaction, State
- **Grouping Things**: Used to organize elements
  Example: Package
- **Annotational Things**: Provide explanations
  Example: Note

UML Relationships
Relationships show how UML elements are **connected or related** to each other.
Types of Relationships

- **Association**: Represents a connection between classes
- **Aggregation**: Represents a “has-a” relationship (weak ownership)
- **Composition**: Strong “has-a” relationship (strong ownership)
- **Generalization**: Represents inheritance (is-a relationship)
- **Dependency**: Shows that one element depends on another

UML Diagrams
Diagrams are graphical representations that show different views of the system. UML diagrams are broadly classified into **structural diagrams** and **behavioral diagrams**.
Short Notes on Common UML Diagrams
Class Diagram
A **Class Diagram** shows the **static structure** of a system, including classes, attributes, methods, and relationships. It is mainly used during the **design phase**.
Use Case Diagram
A **Use Case Diagram** represents the **functional requirements** of the system by showing interactions between **actors** and the system. It helps in requirement analysis
Sequence Diagram

A **Sequence Diagram** illustrates **object interactions over time** and shows the sequence of message exchanges. It is useful for understanding system behavior.

Component Diagram

A **Component Diagram** depicts the **organization and dependencies of software components**. It is useful for architectural and deployment planning

Activity Diagram

An **Activity Diagram** models the **workflow or flow of activities** in a system. It is similar to a flowchart and is useful for business process modeling.

What is a Build Tool

Definition:

A **build tool** is a software utility that **automates converting source code into a runnable application**. It handles tasks like **compiling code, running tests, packaging, and preparing deployment**, ensuring the process is **automatic, consistent, and reliable**.

Need for Build Tools:

Manual builds are **time-consuming and error-prone**, especially in projects with **multiple developers**. Build tools save time, **reduce human errors**, ensure **consistency across environments**, and enable **team collaboration** by standardizing the build process.

Common Tasks Performed:

- **Compile source code** into bytecode or machine code
- **Manage dependencies** by downloading required libraries
- **Run automated tests** to ensure quality
- **Package applications** into deployable formats (JAR, WAR, EAR)
- **Deploy artifacts** to servers or repositories
- Generate **documentation and reports**

Examples:

- **Java:** Maven, Gradle, Ant (Maven also handles project and dependency management)
- **JavaScript:** npm scripts, Webpack, Gulp
- **Python:** Setuptools, Poetry
- **Note:** Maven is widely used in enterprise Java projects for its **standardized structure and dependency management**.

Maven Overview

Maven is a **powerful build and project management tool** used in Java development to automate the process of turning source code into running applications. It handles **compilation, testing, packaging, and deployment** consistently, reducing manual effort and errors. Maven also serves as a **dependency management tool**, automatically downloading and managing external libraries required by a project. By using Maven, developers can focus on writing code while the tool ensures a reliable and repeatable build process.

---

pom.xml – The Heart of Maven

The **pom.xml (Project Object Model)** file is the blueprint of a Maven project. It defines the **project structure, dependencies, plugins, and build instructions**. Dependencies are external libraries that Maven automatically downloads from repositories, avoiding the need to manually manage JAR files. Plugins in Maven execute tasks such as compiling code, running tests, packaging JARs or WARs, and generating documentation. For example, the maven-compiler-plugin allows you to specify the Java version for compilation. Overall, **pom.xml centralizes project configuration**, making builds reproducible and maintainable.

Maven Repositories
Maven uses repositories to store libraries and artifacts. The **local repository** (~/.m2/repository) caches dependencies and plugins downloaded from remote sources, reducing repeated downloads. Maven supports multiple types of repositories: **internal** (hosted within the organization), **central** (the official Maven Central Repository), and **remote** (external or third-party repositories). This flexible system allows developers to use both internal corporate libraries and external open-source dependencies seamlessly.

Standard Maven Project Structure
A typical Maven project follows a **convention-based layout**, making it easier for teams to understand and maintain projects. The standard structure includes src/main/java for source code, src/main/resources for resources, src/test/java for unit tests, and a target folder where compiled classes and packaged artifacts (JARs/WARs) are placed. Maven automatically handles these conventions, ensuring consistency across projects.

Maven Multi-Module Projects
Maven supports **multi-module projects**, allowing large applications to be organized into smaller, modular sub-projects with independent functionality. A **parent POM** manages common configurations, dependencies, and plugin settings, while each **module** has its own POM file for specific tasks. This structure promotes **code reuse, better maintainability, and easier dependency management**. When building the parent project, Maven automatically builds all modules in the correct order based on dependencies, streamlining complex project workflows.

Maven Archetypes
An **archetype** is a **template for creating new Maven projects**. Archetypes can come from the **internal repository** (custom templates within the organization), the **central repository** (predefined templates available in Maven Central), or **remote repositories** (external templates). Using archetypes ensures **standardized project setup**, reducing boilerplate work and maintaining consistency across teams. For example, the maven-archetype-quickstart template creates a simple Java project with preconfigured src directories and a basic POM file.

Integration with Eclipse IDE
Maven is tightly integrated with IDEs like **Eclipse**, either via built-in support or plugins. Developers can create a Maven project by selecting **File → New → Maven Project**, choosing an archetype, and configuring groupId, artifactId, and version. Eclipse generates the Maven structure and POM automatically, allowing developers to focus on coding while Maven handles the build lifecycle.

Common Maven Plugins
Maven plugins perform tasks within specific **build phases**. Examples include:

- maven-compiler-plugin → Compiles Java code.
- maven-surefire-plugin → Runs unit tests and generates reports.
- maven-jar-plugin → Packages code into JARs.
- maven-war-plugin → Packages web applications into WARs.
- maven-shade-plugin → Creates fat JARs including all dependencies.
- maven-dependency-plugin → Analyzes and manages project dependencies.

These plugins allow Maven to automate every step of the build process, from compilation to deployment.

Jenkins CI/CD – Complete Notes with Build Triggers (End-to-End Flow)
What is Jenkins?

Jenkins is an **open-source automation server** used to implement **Continuous Integration (CI)** and **Continuous Delivery/Deployment (CD)**. It automates the process of **building, testing, and deploying applications** whenever changes are made to the source code. Jenkins works with many programming languages and tools such as **Java, Maven, Git, Docker, Kubernetes, AWS**, etc., making it a core tool in DevOps pipelines.

Why Do We Use Jenkins?

Jenkins is used to **reduce manual effort**, **increase development speed**, and **improve software quality**. In modern software development, multiple developers push code frequently. Jenkins automatically validates every change, ensuring early detection of errors. It enables faster releases, reliable deployments, and consistent environments across development, testing, and production.

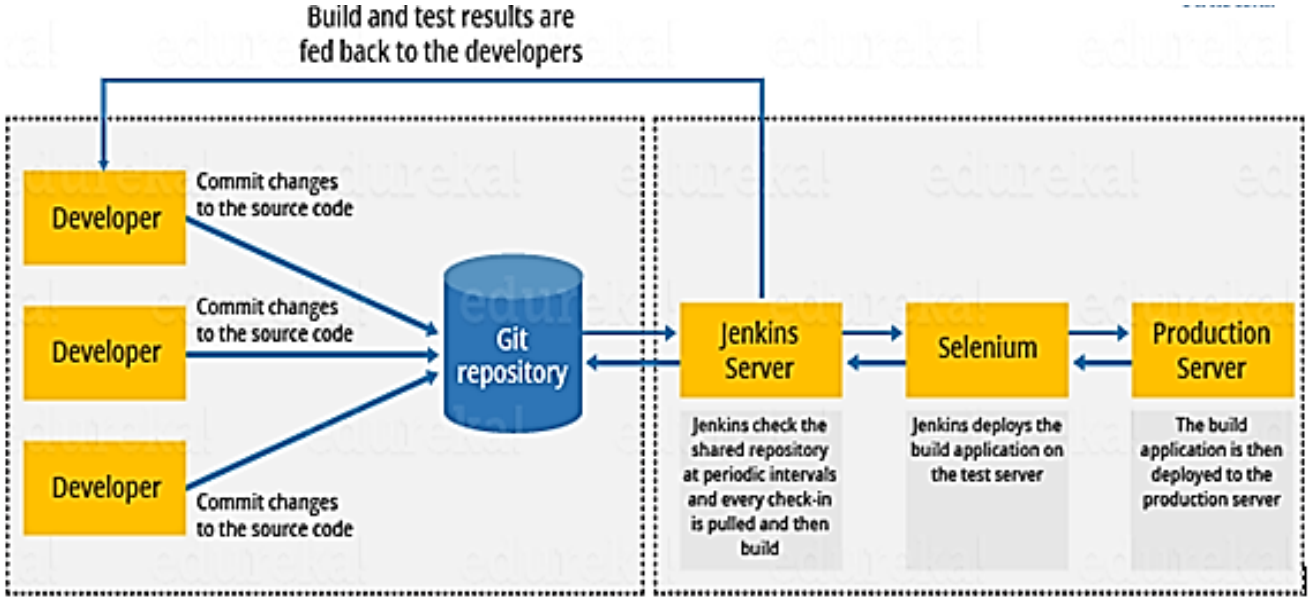
Key Features of Jenkins

Jenkins automates build, test, and deployment workflows. It supports **Pipeline as Code**, allowing CI/CD steps to be written and version-controlled. With **1800+ plugins**, Jenkins integrates easily with almost any tool in the DevOps lifecycle. It supports distributed builds using agents and provides detailed logs and reports for troubleshooting.

Jenkins Architecture (High-Level)

Jenkins follows a **master–agent architecture**.
The **Jenkins Controller (Master)** schedules jobs, manages configurations, and coordinates builds.
The **Agent Nodes (Workers)** execute the actual build, test, and deployment tasks.
This architecture allows Jenkins to scale horizontally and run multiple builds in parallel.

Jenkins workflow:-



Jenkins Plugins (Maven Project Focus)

Plugins extend Jenkins functionality. For Maven-based Java projects, the **Maven Integration Plugin** is essential. It allows Jenkins to execute Maven goals such as clean, compile, test, and package using the pom.xml.
Other commonly used plugins include:

- **Git Plugin** – to clone repositories
- **JUnit Plugin** – to publish test reports
- **Pipeline Plugin** – to define CI/CD pipelines as code
- **Email/Slack Plugins** – for notifications

Plugins make Jenkins adaptable to enterprise-level CI/CD workflows.

Tool Configuration in Jenkins (Java, Maven, Git)
Before running jobs, Jenkins must know where required tools are installed. This is done through **Global Tool Configuration**.

- **JDK Configuration** defines which Java version is used for builds.
- **Maven Configuration** specifies the Maven installation for executing build goals.
- **Git Configuration** enables Jenkins to interact with Git repositories.

Tool configuration ensures builds are **consistent, repeatable, and independent of developer machines**.

Continuous Integration (CI) Using Jenkins
Continuous Integration means **automatically building and testing code whenever changes are pushed**. Jenkins fetches the latest code from Git, compiles it using Maven, and runs automated tests. If any step fails, the build is marked failed and notifications are sent. CI helps teams identify issues early, reducing integration problems and improving overall code quality.

Continuous Delivery / Deployment (CD) Using Jenkins
Continuous Delivery and Deployment automate the release process after successful builds. Jenkins can deploy applications to **Tomcat servers, Docker containers, Kubernetes clusters, or cloud platforms like AWS**. Deployment pipelines ensure the same steps are followed every time, increasing reliability and minimizing human errors.

Build Triggers in Jenkins (Automation Entry Point)
Build triggers define **when Jenkins should start a job or pipeline automatically**. They are the foundation of CI automation.

---

1. GitHub Webhook Trigger (Most Important)
This trigger starts a Jenkins build immediately when code is pushed to GitHub. GitHub sends an event notification to Jenkins using a webhook. This is the **most efficient and recommended trigger** for real-time Continuous Integration.

---

2. Poll SCM Trigger
Poll SCM allows Jenkins to periodically check the Git repository for changes using **CRON syntax**. If changes are detected, a build is triggered.
Example: H/5 * * * * → Poll every 5 minutes.
This trigger is useful when webhooks are not available but is less efficient.

---

3. Build Periodically (Scheduled Builds)
This trigger runs Jenkins jobs at fixed times regardless of code changes. It also uses **CRON syntax**.
Commonly used for **nightly builds, regression testing, and performance testing**.

---

4. Trigger Builds Remotely
This trigger allows Jenkins jobs to be started via a **URL and token**. External systems or scripts can invoke Jenkins builds programmatically. It is useful for integrations with monitoring tools and external automation systems.

---

5. Build After Other Projects Are Built (Job Chaining)
This trigger starts a job only after another job completes successfully. It is used to create **multi-step pipelines**, such as build → test → deploy, ensuring proper execution order.

CRON Syntax (Very Important for Exams)
CRON format:
MIN HOUR DAY MONTH DAY-OF-WEEK
Example:

- 0 0 * * * → Every day at midnight
- 0 2 * * 1-5 → Weekdays at 2 AM

Pipeline as Code Concept
Jenkins supports **Pipeline as Code**, where the entire CI/CD workflow is written in a **Jenkinsfile** and stored in the Git repository. This makes pipelines version-controlled, auditable, and reusable. Pipelines can be **Declarative** or **Scripted**, with Scripted Pipelines offering more flexibility using Groovy.

Groovy is a JVM-based scripting language that is fully compatible with Java and is mainly used for **automation and configuration**. It provides a **simple, concise syntax** compared to Java and supports dynamic typing, closures, and scripting features. In real-world projects, Groovy is widely used in **Jenkins pipelines** and **Gradle build

---

Scripted Pipeline Using Groovy (CI/CD Flow)
A Scripted Pipeline is written in **Groovy** and executed inside a node block. Each stage represents a step in the CI/CD lifecycle:
Below is a **clean, corrected Declarative Jenkins Pipeline** that includes:

1. **Git checkout**
2. **Maven build (clean → install → test → package)**
3. **Docker image build**
4. **Push image to Docker Hub**
5. **Email notification on success/failure**

prerequisites for Email Notification and Docker Hub integration.

✅ **Prerequisites (Very Important)**
🔹 **1. Jenkins Plugins Required**
Make sure the following plugins are installed:

- **Pipeline**
- **Git Plugin**
- **Maven Integration Plugin**
- **Docker Pipeline Plugin**
- **Email Extension Plugin (Email-ext)**

🔹 **2. Tool Configuration in Jenkins**
Go to **Manage Jenkins → Global Tool Configuration**

- Configure **JDK**
- Configure **Maven** (Name it exactly as MAVEN-HOME)
- Configure **Git**
- Install **Docker** on Jenkins machine and ensure Jenkins user has Docker access

---

🔹 **3. Docker Hub Credentials**
Go to **Manage Jenkins → Credentials → Global**

- Add credentials:
    - Kind: Username with password

- ID: docker-hub-creds
- Username: your Docker Hub username
- Password: Docker Hub password or access token

🔷 **4. Email Notification Configuration**

Go to **Manage Jenkins → Configure System**

- Configure **SMTP server** (example: Gmail)
- Enable **Extended E-mail Notification**
- Add:
  - SMTP server: smtp.gmail.com
  - Port: 587
  - Use SSL/TLS
- Add Jenkins email credentials (App Password recommended)

✅ **Rewritten Jenkins Declarative Pipeline**

```
pipeline {
  agent any

  tools {
    maven 'MAVEN-HOME'
  }

  environment {
    DOCKER_IMAGE = "dockerhubusername/mavenjava"
    DOCKER_CREDS = "docker-hub-creds"
    EMAIL_TO     = "team@example.com"
  }

  stages {

    stage('Checkout & Clean') {
      steps {
        git url: 'https://github.com/your-org/mavenjava.git', branch: 'main'
        bat "mvn clean"
      }
    }

    stage('Build') {
      steps {
        bat "mvn install"
      }
    }

    stage('Test') {
      steps {
        bat "mvn test"
      }
    }
```

```
    stage('Package') {
      steps {
        bat "mvn package"
      }
    }

    stage('Docker Build') {
      steps {
        bat "docker build -t %DOCKER_IMAGE%:%BUILD_NUMBER% ."
      }
    }

    stage('Docker Push') {
      steps {
        withCredentials([usernamePassword(
          credentialsId: "${DOCKER_CREDS}",
          usernameVariable: 'DOCKER_USER',
          passwordVariable: 'DOCKER_PASS'
        )]) {
          bat """
            docker login -u %DOCKER_USER% -p %DOCKER_PASS%
            docker push %DOCKER_IMAGE%:%BUILD_NUMBER%
          """
        }
      }
    }
  }

  post {

    success {
      emailext(
        to: "${EMAIL_TO}",
        subject: "SUCCESS: Jenkins Build #${BUILD_NUMBER}",
        body: """
        Build Successful!

        Job: ${JOB_NAME}
        Build Number: ${BUILD_NUMBER}
        Docker Image: ${DOCKER_IMAGE}:${BUILD_NUMBER}
        """
      )
    }

    failure {
      emailext(
        to: "${EMAIL_TO}",
        subject: "FAILED: Jenkins Build #${BUILD_NUMBER}",
        body: """
```

```
            Build Failed!

            Job: ${JOB_NAME}
            Build Number: ${BUILD_NUMBER}
            Please check Jenkins console logs.
            """
        )
      }
    }
}
```

End-to-End Jenkins CI/CD Flow with Triggers
Developers push code to GitHub. A **GitHub webhook trigger** immediately starts the Jenkins pipeline. Jenkins checks out the code, builds it using Maven, runs tests, packages the application, and deploys it to the target environment. Scheduled triggers may run nightly tests, while job chaining ensures deployments occur only after successful builds. Notifications are sent at every stage, ensuring transparency and fast feedback.

What is Software Testing
Software testing is
Below are **detailed, structured, exam- and interview-ready notes** on **Software Testing Strategies**, written in **clear flow**, **paragraph-wise**, and aligned with **software engineering syllabi and real-world practice**.

Software Testing Strategies – Detailed Notes

---

1. Strategic Approach to Software Testing
A **strategic approach to software testing** is a planned and systematic method used to verify that software meets its requirements and works correctly under different conditions. Testing is not performed randomly; instead, it follows a **defined strategy** that integrates testing activities throughout the software development lifecycle (SDLC).
The main goal of a testing strategy is to **reduce defects, improve quality, and ensure reliability**. A well-defined strategy helps identify what to test, how to test, when to test, and who should perform the testing. It also ensures efficient use of resources, early detection of defects, and risk minimization.
A strategic testing approach emphasizes:

- Early testing (shift-left testing)
- Risk-based testing
- Incremental and iterative testing
- Continuous feedback to developers

---

2. Verification and Validation (V&V)
Verification and Validation are two complementary processes used to ensure software quality.
Verification answers the question:
👉 *"Are we building the product right?"*
It focuses on checking whether the software is developed according to specifications and design documents. Verification activities are mostly **static**, meaning the code is not executed. Examples include:

- Reviews
- Inspections
- Walkthroughs
- Static code analysis

Validation answers the question:

👉 *"Are we building the right product?"*

It ensures the final product meets user requirements and expectations. Validation involves **dynamic testing**, where the software is executed with test cases. Examples include:

- System testing
- User acceptance testing
- Functional testing

Together, verification and validation ensure both **correct construction** and **correct functionality**.

---

3. Software Test Strategy

A **software test strategy** is a high-level document that defines the **overall approach to testing** within a project. It acts as a roadmap for test planning and execution.

A test strategy typically includes:

- Testing objectives
- Types of testing to be performed
- Testing levels (unit, integration, system, acceptance)
- Testing tools and environments
- Entry and exit criteria
- Risk analysis and mitigation
- Roles and responsibilities

A good test strategy ensures consistency across testing activities and aligns testing goals with business objectives.

---

4. Test Strategies for Conventional Software

Conventional software refers to traditional, procedural, or object-oriented applications (non-AI, non-real-time systems).

Testing strategies for conventional software follow a **layered approach**:

1. **Unit Testing** – Tests individual functions or modules.
2. **Integration Testing** – Tests interactions between modules.
3. **System Testing** – Tests the complete system as a whole.
4. **Acceptance Testing** – Confirms readiness for delivery.

This layered approach ensures defects are detected early and progressively as the system grows in complexity.

---

5. Black-Box Testing

Black-box testing focuses on testing the **functional behavior** of the software without knowledge of internal code structure. The tester only considers inputs and expected outputs.

Key characteristics:

- Based on requirements and specifications
- No knowledge of internal logic required
- Suitable for system and acceptance testing

Common black-box techniques:

- Equivalence Partitioning
- Boundary Value Analysis
- Decision Table Testing
- State Transition Testing

Black-box testing ensures that the software behaves correctly from the **user's perspective**.

---

6. White-Box Testing

White-box testing (also called structural testing) involves testing the **internal logic and code structure** of the software. The tester has full knowledge of the source code.
Key characteristics:

- Focuses on code paths, conditions, and loops
- Typically performed during unit testing
- Ensures complete code coverage

Common white-box techniques:

- Statement coverage
- Branch coverage
- Path coverage
- Condition coverage

White-box testing helps detect logic errors, dead code, and security vulnerabilities.

---

7. Validation Testing
Validation testing ensures that the software meets user requirements and is fit for real-world use. It confirms that the developed system fulfills its intended purpose.
Types of validation testing include:

- Functional testing
- Performance testing
- Security testing
- Compatibility testing
- User Acceptance Testing (UAT)

Validation testing is usually performed after system testing and before release.

---

8. System Testing
System testing is a high-level testing phase where the **entire integrated system** is tested as a complete unit. It evaluates both functional and non-functional requirements.
Key objectives:

- Verify end-to-end functionality
- Validate system behavior under real-world conditions
- Identify issues related to performance, security, and usability

System testing is performed in an environment that closely resembles production.

---

9. The Art of Debugging
Debugging is the process of **locating, analyzing, and fixing defects** found during testing or runtime. It is both a technical and analytical skill.
Steps in debugging:

1. Reproduce the problem
2. Isolate the root cause
3. Analyze the code logic
4. Fix the defect
5. Retest to ensure correctness
6. Perform regression testing

Effective debugging requires logical thinking, deep understanding of code, and systematic problem-solving.

---

10. Relationship Between Testing and Debugging

Testing	Debugging
Identifies defects	Fixes defects

Testing	Debugging
Performed by testers	Performed by developers
Prevents failures	Corrects failures

Both are essential for delivering high-quality software.

What is Docker

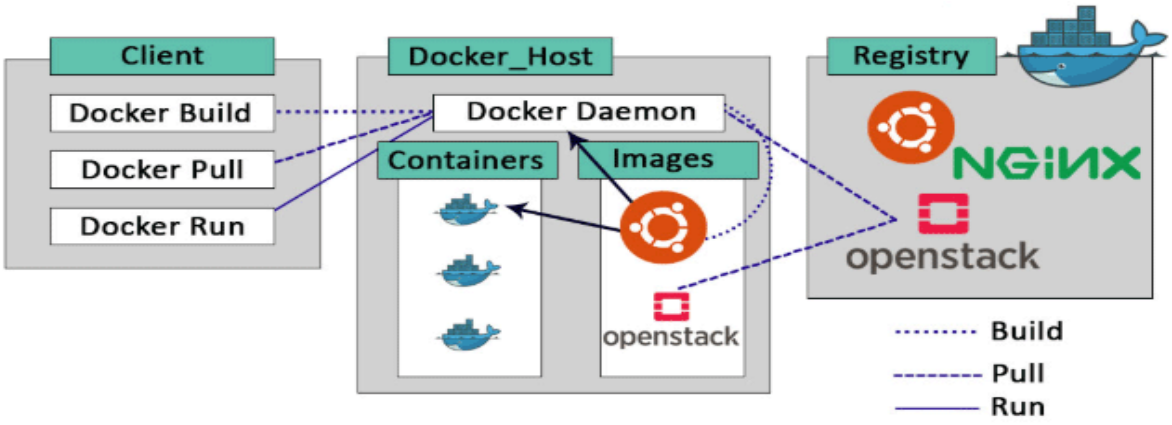
Docker is a containerization platform that allows developers to package an application along with its dependencies—libraries, binaries, and configuration files—into a single lightweight unit called a container. Docker ensures applications run consistently across different environments such as development, testing, and production. By isolating applications from the host operating system, Docker eliminates environment-related issues and improves portability and deployment reliability.

Docker Architecture

Docker follows a client–server architecture. **Docker Engine**

- **Docker Daemon (dockerd)**
  - Runs on the host OS and manages containers, images, volumes, and networks.
  - Listens for API requests from CLI or other tools.
- **Docker CLI (docker)**
  - Command-line interface used to communicate with the Docker Daemon.
  - Example: docker run nginx
- **REST API**

Exposes endpoints for programmatic communication with the Docker Daemon



Docker Components

1. **Images:**
   - Immutable, read-only templates used to create containers.
   - Built from a **Dockerfile** or pulled from a registry like Docker Hub.
   - Example: docker pull ubuntu:22.04
2. **Containers:**
   - Runtime instances of images.
   - Can be started, stopped, deleted, and restarted.
   - Example: docker run -it ubuntu:22.04 bash
3. **Volumes:**
   - Persistent storage to retain data beyond container lifecycle.

- Example: docker volume create my_data

4. **Networks:**
   - Enable container-to-container communication.
   - Types:
     - **Bridge:** Default, isolated network on a single host
     - **Host:** Shares host network stack
     - **Overlay:** Multi-host communication
     - **None:** No networking
5. **Registries:**
   - Stores Docker images.
   - Public: Docker Hub
   - Private: Self-hosted or cloud registries

Docker Daemon Workflow

When a user runs a Docker command, the Docker CLI forwards the request to the Docker Daemon. The daemon checks whether the required image exists locally. If not, it pulls the image from a registry. The daemon then creates a container by allocating filesystem layers, network interfaces, and resource isolation, and finally starts the containerized application.

Docker Image vs Docker Container

A Docker image is a static blueprint containing application code and dependencies, whereas a Docker container is a dynamic runtime instance of that image. One image can be used to create multiple containers. In simple terms, an image is like a class, and a container is an object created from that class.

Dockerfile (Image Build Recipe)

- **Docker Image:**
  - Read-only template
  - Blueprint of the application
  - Created using Dockerfile
- **Docker Container:**
  - Running instance of an image
  - Actual application execution
  - Created using docker run

☞ **Simple analogy:**

Image = Class / Blueprint

Container = Object / Running App

Example Dockerfile (Java / Python style – Exam Ready)

```
# Base image
FROM python:3.11-slim

# Set working directory inside container
WORKDIR /app                          # Layer 1

# Copy application files
COPY . .                              # Layer 2

# Install dependencies
RUN pip install -r requirements.txt   # Layer 3
```

```
# Expose application port
EXPOSE 5000                    # Layer 4

# Start the application
CMD ["python", "app.py"]       # Layer 5
```

Build the Image

```
docker build -t myapp:1.0 .
```

This Dockerfile builds **one Docker image**, which can be reused to create multiple containers.

Docker Image Layers

Each Dockerfile instruction creates a separate layer. These layers are cached and reused across builds, making Docker images efficient and lightweight. The final image is a stack of all layers combined.

Accessing and Modifying Containers

Containers can be accessed interactively using terminal mode. Users can inspect files, execute commands, or debug applications inside the container. Any changes made inside a running container are temporary unless committed to a new image. However, modifying the **Dockerfile and rebuilding the image is the recommended** and reproducible approach.

One Dockerfile – One Image Principle

A Dockerfile is designed to build **one image**, and Docker follows the principle of **one main process per container**. While a single image can create multiple containers, a Dockerfile itself does not define multiple containers. Multi-container setups are handled using Docker Compose.

Docker Compose (Multi-Container Orchestration)

Docker Compose is used to define and manage **multi-container applications** using a YAML configuration file called docker-compose.yml. It allows developers to define services, networks, volumes, and dependencies in a single file, making application orchestration simple and consistent.

Example docker-compose.yml (Web + Database)

```
version: "3.9"

services:
  web:
    build: .
    ports:
      - "8080:5000"
    depends_on:
      - db

  db:
    image: postgres:15
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secret
    volumes:
      - db_data:/var/lib/postgresql/data

volumes:
```

db_data:

Explanation

The web service is built using the Dockerfile in the current directory and exposes port 5000 internally, mapped to port 8080 on the host. The db service uses a PostgreSQL image and stores data persistently using a volume. The depends_on option ensures the database starts before the web service.

Docker Compose Commands and Workflow

Docker Compose simplifies application lifecycle management. The docker-compose build command builds images only. The docker-compose up -d command builds (if needed) and starts containers in detached mode. The --build flag forces image rebuilds. Docker Compose also supports scaling services horizontally using the --scale option.

Docker Networking

Docker networking allows containers to communicate securely. The bridge network is the default for single-host setups, while overlay networks enable communication across multiple hosts. Proper networking is essential for microservices-based applications.

Volumes and Persistent Storage

Docker volumes provide persistent storage independent of container lifecycle. Volumes are commonly used for databases, logs, and configuration files. They ensure data safety even when containers are recreated or removed.

Docker Container Lifecycle

The container lifecycle includes creation, start, execution, stop, and removal. Containers can be attached to for debugging, gracefully stopped, or permanently deleted. Understanding this lifecycle helps manage resources efficiently.

Container Orchestration Using Kubernetes

Understanding Container Orchestration

Container orchestration is the automated management of containerized applications across multiple machines. In real-world environments, applications consist of many containers that must be **deployed, scaled, monitored, restarted, and networked** properly. Manually managing containers becomes complex and error-prone as applications grow.

◈ **Why orchestration is required**

- Automates container deployment and management
- Handles failures and restarts automatically
- Scales applications up or down based on load
- Manages networking and service discovery
- Ensures high availability

☆ **Kubernetes** is the most popular container orchestration platform used in production systems.

What is Kubernetes (K8s)?

Kubernetes is an **open-source container orchestration platform** originally developed by Google. It is used to **deploy, manage, scale, and maintain containerized applications** automatically.

Key Purpose of Kubernetes

- Manage containers across a **cluster of machines**
- Maintain desired application state
- Automate scaling and self-healing

- Enable rolling updates with zero downtime

📌 Kubernetes works with **Docker and other container runtimes**.

Kubernetes Architecture (High-Level View)

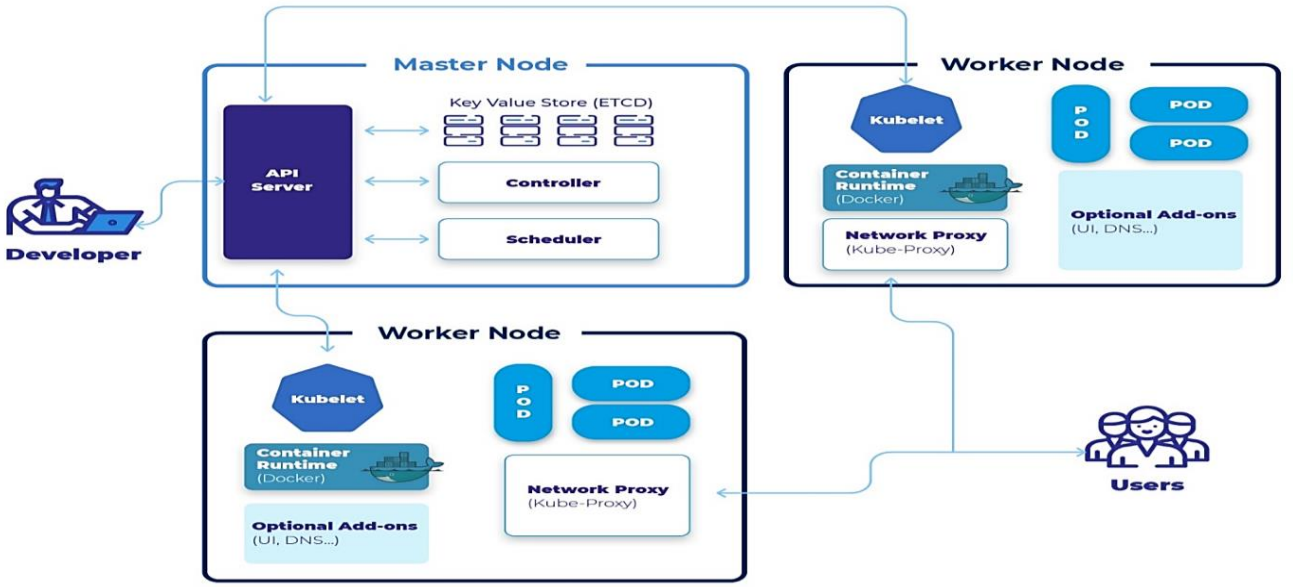
Kubernetes follows a **master–worker (control plane–node)** architecture.

Control Plane (Master Node)

- **API Server** – Entry point for all commands (kubectl)
- **Scheduler** – Assigns Pods to worker nodes
- **Controller Manager** – Maintains desired state
- **etcd** – Key-value store for cluster data

Worker Nodes

- **Kubelet** – Communicates with control plane
- **Container Runtime** – Runs containers (Docker, containerd)
- **Kube-proxy** – Handles networking and service routing



Kubernetes Core Concepts

Pod

A **Pod** is the **smallest deployable unit** in Kubernetes. It represents **one or more containers** that:

- Share the same network (IP, ports)
- Share storage volumes
- Are scheduled together on the same node

📌 **Key Points**

- One Pod usually runs **one main container**
- Pods are **ephemeral** (temporary)
- If a Pod dies, Kubernetes creates a new one

Node

A **Node** is a worker machine (VM or physical server) that runs Pods. Each node can host multiple Pods and is managed by the Kubernetes control plane.

Cluster

A **Kubernetes Cluster** consists of:

- One or more **control plane nodes**
- Multiple **worker nodes**
- Shared networking and storage

Deploying Pods in Kubernetes

Pods can be deployed using **YAML manifest files**.

Example: Pod Deployment YAML

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
  - name: myapp-container
    image: nginx
    ports:
    - containerPort: 80
```

Deploy Pod Command

```
kubectl apply -f pod.yaml
```

📌 **Important Notes**

- Pods are not self-healing by default
- Manual Pod management is not recommended for production
- Pods do not scale automatically

Kubernetes Deployments (Managing Pods)

A **Deployment** is a higher-level Kubernetes object used to **manage Pods automatically**. It ensures:

- Desired number of Pods are always running
- Automatic Pod replacement on failure
- Rolling updates without downtime
- Easy scaling

☆ **Deployments are production-ready objects**

Why Deployments are Preferred Over Pods

- Self-healing
- Scalability
- Version control (rollbacks)
- Zero-downtime updates

Creating a Deployment

Example: Deployment YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
```

```
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp-container
        image: nginx
        ports:
        - containerPort: 80
```

Apply Deployment

```
kubectl apply -f deployment.yaml
```

📌 **Explanation**

- replicas: 3 → Runs 3 Pods
- Kubernetes ensures **always 3 Pods running**
- If one Pod crashes, a new Pod is created automatically

How Deployment Manages Pods

Deployments use **ReplicaSets** internally to manage Pods.

Deployment Workflow

1. User creates Deployment
2. Deployment creates ReplicaSet
3. ReplicaSet creates Pods
4. Kubernetes monitors Pods continuously

If a Pod fails → **new Pod is automatically created**

Scaling Pods Using Deployment

Kubernetes allows **horizontal scaling**.

Manual Scaling

```
kubectl scale deployment myapp-deployment --replicas=5
```

Automatic Scaling (HPA)

- Based on CPU or memory usage
- Kubernetes increases or decreases Pods automatically

Rolling Updates and Rollbacks

Rolling Update

- Pods are updated **one by one**
- No downtime
- Old Pods replaced gradually

Rollback

```
kubectl rollout undo deployment myapp-deployment
```

📌 Ensures safe application upgrades

Kubernetes Self-Healing Features

Kubernetes continuously monitors application health and:

- Restarts failed containers
- Replaces failed Pods
- Reschedules Pods to healthy nodes
- Maintains desired state automatically

Software Risk Management

Risk Management is a systematic process of **identifying, analyzing, prioritizing, and controlling risks** that may negatively impact a software project's **cost, schedule, quality, or performance**. Its key objective is to **minimize uncertainty and potential losses** while maximizing project success.

Risk strategies can be **reactive** or **proactive**.
In a **reactive approach**, risks are addressed **only after problems occur**, leading to delays, cost overruns, and panic-driven decisions. In contrast, a
proactive strategy identifies risks early, prepares mitigation plans, and improves predictability and control. Proactive risk management is the recommended approach in modern software engineering.

Software risks are potential events affecting a project and can be classified into:

- **Project Risks:** Unrealistic deadlines, poor planning, lack of resources, team inexperience
- **Technical Risks:** New or unproven technology, integration issues, poor architecture, complex algorithms
- **Business Risks:** Market changes, budget cuts, customer loss, product cancellation
-

Risk Identification involves systematically discovering potential risks by **brainstorming, checklists, past project experience, expert judgment, or questionnaires**. Typical risks include **requirement changes, skill shortages, schedule pressure, poor communication, and tool limitations**.

Risk Projection (Estimation) evaluates risks based on **probability and impact**, calculating **Risk Exposure (RE = Probability × Impact)** to prioritize high-risk items. **Risk Refinement** breaks high-level risks into detailed, actionable sub-risks. For example, a high-level risk like *"Poor product quality"* can be refined into **inadequate testing, poor design reviews, and lack of coding standards**.

RMMM (Risk Mitigation, Monitoring, and Management) is used to handle risks throughout the project lifecycle.

- **Risk Mitigation:** Actions to reduce risk probability or impact (e.g., training, prototyping, using proven technologies)
- **Risk Monitoring:** Continuous tracking of risk indicators, status, and triggers
- **Risk Management:** Actions when a risk occurs (e.g., backup resources, schedule adjustment, alternative solutions)

An **RMMM Plan** documents all identified risks, their **probability and impact**, mitigation strategies, monitoring mechanisms, contingency plans, and responsible persons. It ensures **preparedness, reduces surprises, and enhances project control**.

FURPS (Functionality, Usability, Reliability, Performance, Supportability) is integrated into risk management to **identify and manage quality-related risks**. For example, performance risks like slow database queries and reliability risks like poor exception handling can be tracked and mitigated using FURPS.

Why EcoConnect Moved to AWS

As EcoConnect grew rapidly, on-premise servers became difficult to scale, maintain, and manage. AWS provides **scalability, reliability, cost efficiency, and global availability** using a pay-as-you-go model. This makes it ideal for startups like InnovateNow that need flexible infrastructure without heavy upfront investment.

Amazon EC2 (Elastic Compute Cloud)

Amazon EC2 is an AWS service that provides **resizable virtual servers (instances)** in the cloud. These instances act as the **foundation compute layer** where applications, Docker containers, and Kubernetes clusters run.

📌 **Key Idea**:

Even when using **Docker and Kubernetes**, you still need machines to run them—EC2 provides those machines.

Role of EC2 in Containerized Applications

EC2 instances host:

- Docker containers
- Kubernetes worker nodes
- Backend APIs and microservices

Instead of physical servers, EC2 offers **on-demand cloud servers** that can be scaled up or down easily based on application load.

Key EC2 Components

Amazon Machine Image (AMI)

An **AMI** is a template used to launch EC2 instances. It includes:

- Operating system (Ubuntu, Amazon Linux, etc.)
- Preinstalled software
  Using AMIs saves time and ensures consistency.

Instance Types

EC2 provides different instance types based on workload needs:

- **Compute-optimized** – CPU-intensive tasks
- **Memory-optimized** – Databases, caching
- **Storage-optimized** – Large data processing

This allows EcoConnect to **optimize performance and cost**.

Networking & Security in EC2

VPC (Virtual Private Cloud)

A **VPC** is a logically isolated virtual network in AWS where EC2 instances run. It provides full control over IP addressing, subnets, and routing.

Security Groups

Security Groups act as **virtual firewalls**, controlling:

- Inbound traffic (who can access the instance)
- Outbound traffic (what the instance can access)

Key Pair

A **Key Pair** is used for **secure SSH access** to EC2 instances.

Scalability & Availability Features

Auto Scaling

Automatically increases or decreases the number of EC2 instances based on traffic demand.

Elastic Load Balancing

Distributes incoming traffic across multiple EC2 instances to ensure:

- High availability

- Fault tolerance
- Better performance

EC2 in EcoConnect Architecture

For EcoConnect:

- EC2 instances host backend APIs
- Docker containers run on EC2
- Kubernetes clusters use EC2 as worker nodes
- Auto Scaling and Load Balancers handle traffic spikes

Key Benefits of EC2

- No hardware maintenance
- Easy scalability
- Flexible instance selection
- Strong security controls
- Cost optimization

Software Engineering – SDLC, Agile, and DevOps

1. What is a Software Process Model, and why is choosing the right model important?
2. Compare the Waterfall, Incremental, Prototyping, Spiral, and Agile models in terms of advantages, limitations, and best use cases.
3. Explain the Scrum framework and its key components, including Product Backlog, Sprint Planning, Daily Scrum, Sprint Review, and Sprint Retrospective.
4. What is DevOps, and how does it differ from Traditional Waterfall and Agile approaches?

Requirements & SRS

5. What are the main differences between functional and non-functional requirements?
6. Why is a Software Requirements Specification (SRS) important in a software project?
7. Explain the difference between verification and validation in software testing.

Source Code Management & Git

5. What is Source Code Management (SCM) and why is it important in software development?
6. Differentiate between Centralized Version Control Systems (CVCS) and Distributed Version Control Systems (DVCS) with examples.
7. What is the purpose of the .gitignore file? Give an example.
8. Explain the Git 3-step workflow: Working Directory → Staging Area → Local Repository.
9. How do you create a new branch and merge it into the main branch in Git?
10. What is a merge conflict? Explain how it can be resolved.
11. Explain cherry-picking in Git. When is it used?

Docker & Docker Compose

13. What is the difference between a Docker image and a Docker container?
14. Write a simple Dockerfile and explain its main instructions.
15. What is Docker Compose and when would you use it?

Kubernetes (Container Orchestration)

16. What is container orchestration and why is it needed?
17. Name and explain the primary Kubernetes objects used for deployment (e.g., Pods, Deployments, Services).
18. How does Kubernetes manage scaling and self-healing of containerized applications?

Software Risk Management & EC2

19. What is Software Risk Management, and what are the key steps involved in identifying, analyzing, and handling software project risks?

20. What are the key components of EC2, such as AMI, instance types, VPC, security groups, and key pairs, and how do they contribute to performance, security, and cost optimization?
