Requirements Analysis and Cost Estimation

Q. What is Requirement. Describe the Characteristics of requirement

A **requirement** is a **condition or capability** that a software system must have to satisfy the needs of users, stakeholders, or business objectives. It defines **what the system should do** and acts as a foundation for software development.

Formal Definition:

"A requirement is a documented representation of a condition or capability needed by a user to solve a problem or achieve an objective." — IEEE Standard

Example:

For an **online banking system**, requirements might include:
✓ Users must be able to **log in securely**.
✓ The system should allow **fund transfers** between accounts.
✓ It should display **transaction history** for the last 6 months.

Characteristics of a Good Requirement

A **well-defined requirement** must have the following characteristics:

1. Correct

✓ The requirement must be **accurate and error-free**.
✓ It should truly reflect the **customer's needs**.
Example: "The system must display account balance in real-time."

---

2. Complete

✓ The requirement should provide **all necessary details**.
✓ No missing information or ambiguity.
Bad Example: "The system should allow payments." (No details about payment methods)
Good Example: "The system should support **credit cards, debit cards, and UPI payments**."

---

3. Clear & Unambiguous

✓ Each requirement should have **only one possible interpretation**.
✓ Avoid **vague terms** like "fast," "efficient," or "user-friendly."

Bad Example: "The app should load quickly."
Good Example: "The app should load within **2 seconds** on a 4G connection."

---

4. Consistent

✓ No **contradictions** between different requirements.

Good Example: "The system should allow password reset via email."

Bad Example: "The system should not send emails for password reset." (Contradiction)

---

5. Verifiable (Testable)

✓ The requirement must be **measurable and testable**.

✓ It should define **clear success criteria**.
Example: "The login process should not take more than **5 seconds**."

---

6. Feasible (Realistic)

✓ The requirement should be **technically and economically possible**.

✓ Avoid demands that **exceed available resources**.
Bad Example: "The system should handle **1 billion users at launch**."
Good Example: "The system should scale up to **100,000 users in the first phase**."

---

7. Traceable

✓ Each requirement should be **linked to a business need or objective**.

✓ Helps in tracking **why** a requirement was included.

Example: "Requirement ID: **REQ-101** → Feature: **User Authentication** → Business Goal: **Ensure Security**."

---

8. Modifiable

✓ The requirement should be **easy to update** when business needs change.
Example: If **new regulations** require a **change in data privacy rules**, the requirement should be adaptable.

---

9. Prioritized

✓ Requirements should be **ranked based on importance**.
Example:

- **High Priority:** "System must have **multi-factor authentication**."
- **Low Priority:** "System should have a **dark mode option**."

Q. Explain the types of requirements

Software requirements are broadly classified into **two main types**:
1. Functional Requirements (FRs) – Define **what** the system should do.
2. Non-Functional Requirements (NFRs) – Define **how** the system should perform.

---

1. Functional Requirements (FRs)

Functional requirements describe the **specific functions, features, and behaviors** of a software system. They define **how the system interacts with users, external systems, and data**.

Examples:
✓ Users must be able to **log in using a username and password**.
✓ The system should **process online payments using credit cards and UPI**.
✓ A hospital management system must **store patient records securely**.

Key Characteristics:

- Defines **inputs, processes, and outputs**.
- Describes **system behavior** in different scenarios.
- Can be written as **use cases or user stories**.

---

2. Non-Functional Requirements (NFRs)

Non-functional requirements define the **quality attributes, constraints, and system behavior expectations**. They specify how well the system performs rather than what it does.

Example:
✓ The website should **load within 2 seconds** under normal conditions.
✓ The system should handle **1,000 concurrent users** without crashing.
✓ The database should be **backed up every 24 hours** for security.

Types of Non-Functional Requirements (NFRs)

A. Product Requirements (Performance & Quality Attributes)

These define **how the software should perform** in terms of speed, usability, security, etc.

Examples:
✓ **Performance:** The system should process **100 transactions per second**.

✓ **Usability:** The UI should be **user-friendly and accessible**.
✓ **Security:** Data should be **encrypted using AES-256 encryption**.

B. Organizational Requirements (Development & Business Constraints)

These define **company policies, tools, standards, and project constraints** that must be followed.

Examples:
✓ **Development Standards:** The software should follow **ISO 9001 quality standards**.
✓ **Programming Language:** The system must be developed using **Java and PostgreSQL**.
✓ **Deployment Environment:** The system should run on **AWS cloud servers**.

C. External Requirements (Legal & Regulatory Compliance)

These define the **legal, regulatory, and ethical** constraints imposed on the software.

Examples:
✓ **Legal Compliance:** The software must comply with **GDPR (General Data Protection Regulation)**.
✓ **Regulatory Standards:** Banking applications must follow **RBI security guidelines**.
✓ **Ethical Requirements:** The system should **not store sensitive user data without consent**.

Q. What are the different requirements engineering tasks/Explain feasibility studies in detail

Requirements Engineering (RE) is the **systematic process** of gathering, analyzing, documenting, and validating software requirements. It ensures that the final software **meets user needs and business objectives**.

Purpose: To develop a **clear, complete, and accurate** set of software requirements.
Key Activities: Requirements **elicitation, analysis, specification, validation, and management**.

Different Tasks in Requirements Engineering

1. Requirements Elicitation (Gathering Requirements)

✓ The process of **collecting requirements** from stakeholders.
✓ Involves understanding **what users need** from the system.

Techniques Used:

- **Interviews** → Direct discussions with users.
- **Surveys & Questionnaires** → Collecting structured feedback.
- **Observation** → Studying how users perform tasks.
- **Prototyping** → Creating a working model for feedback.

Example: Interviewing doctors and nurses to gather requirements for a **hospital management system**.

---

2. Requirements Analysis (Refining Requirements)

✓ Identifying **gaps, conflicts, or missing details** in requirements.
✓ Classifying requirements into **functional and non-functional**.
✓ Ensuring **feasibility** and **removing ambiguities**.

Techniques Used:

- **Data Flow Diagrams (DFDs)** → Understanding system workflows.
- **Use Case Diagrams** → Representing user interactions.
- **Prototyping** → Checking feasibility of complex features.

Example: Analyzing if an **e-commerce app** can handle **1 million transactions per day**.

---

3. Requirements Specification (Documenting Requirements)

✓ Writing requirements in a structured format.
✓ Creating a **Software Requirements Specification (SRS) document**.

Key Contents of an SRS:

- **Introduction** → Purpose & scope of the system.
- **Functional Requirements** → Features & capabilities.
- **Non-Functional Requirements** → Performance, security, scalability.
- **System Models** → Diagrams & architecture.

Example: Writing an SRS for an **online banking system**, detailing login, transaction, and security requirements.

---

4. Requirements Validation (Ensuring Correctness)

✓ Checking if the requirements **meet customer expectations**.
✓ Ensuring **consistency, completeness, and feasibility**.

Techniques Used:

- **Reviews & Inspections** → Team discussions to identify errors.
- **Prototyping** → Developing a basic version for validation.
- **Testing Scenarios** → Simulating real-world cases.

Example: A retail company **reviews the requirements** to ensure their **inventory system** meets business needs.

---

5. Requirements Management (Tracking & Handling Changes)

✓ Keeping track of **requirement changes** throughout development.
✓ Ensuring that **updates do not affect software stability**.

Key Activities:

- **Version Control** → Managing different versions of requirements.
- **Change Management** → Handling modifications systematically.
- **Traceability Matrix** → Mapping requirements to test cases & development stages.

Example: Updating **data privacy requirements** in a social media app due to new GDPR regulations.

Q. Explain is SRS document

A **Software Requirements Specification (SRS) Document** is a **detailed description** of a software system's **functional and non-functional requirements**. It serves as a **blueprint** for developers, testers, and stakeholders, ensuring that the software meets user needs and business goals.

Purpose: Defines **what** the software should do (not how).
Importance: Helps in **clear communication** between stakeholders and the development team.

Components of an SRS Document

An SRS document typically consists of the following sections:

1. Introduction

✓ Provides an **overview** of the software project.
✓ Defines the **scope, objectives, and intended users**.

Example: "This document specifies requirements for an online banking system, allowing users to manage their accounts, transfer funds, and view transaction history."

---

2. Overall Description

✓ Describes the **general functionality and purpose** of the system.
✓ Includes:

- **Product perspective** (How the system fits into the existing environment).
- **User characteristics** (Target audience, skill level).
- **Assumptions & constraints** (Hardware, software, legal requirements).

Example: "The system will be accessible via web and mobile, requiring an internet connection."

---

3. Functional Requirements

✓ Describes the **features and functions** of the system.
✓ Specifies **inputs, processing, and expected outputs**.

Example:
✓ Users must be able to **log in with a username and password**.
✓ The system must allow **fund transfers between accounts**.
✓ A search feature should **retrieve products based on keywords**.

---

4. Non-Functional Requirements

✓ Defines **quality attributes**, such as performance, security, usability, etc.

Example:
✓ **Performance:** The system must handle **1,000 concurrent users**.
✓ **Security:** All passwords must be **encrypted using AES-256**.
✓ **Usability:** The mobile app should load within **3 seconds**.

---

5. System Models (Diagrams & Representations)

✓ Includes **UML diagrams, flowcharts, and data models** to visually explain system behavior.

Example:

- **Use Case Diagram** for user interactions.
- **Data Flow Diagram (DFD)** for data movement within the system.

---

6. Assumptions and Dependencies

✓ Lists any **assumptions** made during requirement gathering.
✓ Identifies **external dependencies** (APIs, third-party services).

Example: "The system assumes that users have a stable internet connection and Java Runtime installed."

Structure of an SRS Document (IEEE Standard 830-1998)

1. Introduction

1.1 Purpose

1.2 Scope

1.3 Definitions, Acronyms, and Abbreviations

1.4 References

1.5 Overview

2. Overall Description

2.1 Product Perspective

2.2 Product Functions

2.3 User Characteristics

2.4 Constraints

2.5 Assumptions and Dependencies

3. Specific Requirements

3.1 Functional Requirements

3.2 Non-Functional Requirements

3.3 Interface Requirements

3.4 System Models

4. Appendices

Advantages of an SRS Document

✓ **Clear Communication** → Ensures all stakeholders agree on project goals.
✓ **Better Project Planning** → Helps in estimating **cost, time, and resources**.
✓ **Early Issue Detection** → Identifies conflicts or missing requirements.
✓ **Acts as a Legal Contract** → Serves as a reference for future disputes.

Q. What are the Characteristics of SRS

A **Software Requirements Specification (SRS)** document serves as a **blueprint** for software development. A well-written SRS ensures **clarity, accuracy, and completeness**, reducing misunderstandings between stakeholders and developers.

Characteristics of a Good SRS

1. Correctness

✓ The SRS must accurately define **all user and system requirements**.
✓ Every requirement must **match the actual needs** of stakeholders.

Example: If the system needs **OTP-based login**, the requirement should **clearly mention OTP verification via SMS or email**.

---

2. Completeness

✓ The SRS should cover **all functional and non-functional requirements**.
✓ It must include **system constraints, assumptions, and dependencies**.

Example: Instead of just stating "The system should generate reports," it should specify:
"The system must generate **monthly sales reports in PDF format** and allow **export to Excel**."

---

3. Unambiguity (Clarity)

✓ Each requirement must have **only one interpretation**.
✓ Avoid vague terms like "fast," "user-friendly," or "efficient."

Example: Instead of saying "The system should load quickly," it should state:
"The system should **load within 3 seconds** on a 4G network."

---

4. Consistency

✓ Requirements should **not conflict** with each other.
✓ All terminologies should be **uniform across the document**.

Bad Example:

- "The system should send **email notifications for password reset**."
- "The system should **not use email for password recovery**." (Contradiction!)

Good Example:

- "The system should allow password recovery via **email and SMS OTP**."

---

5. Verifiability (Testability)

✓ Every requirement must be **measurable and testable** through **inspection or testing**.
✓ If a requirement cannot be tested, it should be rewritten.

Example: Instead of "The UI should be user-friendly," state:
"The UI should follow **Material Design Guidelines** and support **dark mode**."

---

6. Modifiability

✓ The SRS should be **structured and modular**, making it **easy to update**.
✓ Each requirement should be **uniquely identified** (e.g., **REQ-101, REQ-102**).

Example: If a new **government regulation** affects data privacy, changes should be **easily incorporated** into the SRS.

---

7. Traceability

✓ Every requirement should be **linked to a specific function, business need, or test case**.
✓ Helps in **tracking changes** and verifying implementation.

Example:

Requirement ID	Feature	Linked to Test Case
REQ-101	User Login	TC-001 (Verify login works correctly)
REQ-202	Payment Gateway	TC-010 (Verify successful transactions)

---

8. Feasibility

✓ Requirements should be **realistic and achievable** within project constraints.
✓ Avoid **overly complex or impossible demands**.

Bad Example: "The system must handle **unlimited** simultaneous users."
Good Example: "The system must support **10,000 concurrent users** on AWS infrastructure."

---

9. Prioritization

✓ Important requirements should be **ranked based on priority**.
✓ Helps developers **focus on critical features first**.

Example:

Priority	**Requirement**	**Feature**
High	REQ-101	User Authentication
Medium	REQ-202	Dark Mode UI
Low	REQ-305	Custom Avatars

Q. Explain 3Ps

The **3Ps in Software Engineering** refer to:

1. People → The individuals involved in software development.
2. Process → The methods and frameworks used to develop software.
3. Product → The final software system being developed.

These three elements form the **foundation** of software development and project management, ensuring that high-quality software is delivered efficiently.

Explanation of 3Ps

1. People (Who is involved?)

✓ People are the **most important asset** in software development.
✓ Includes **developers, project managers, testers, customers, and stakeholders**.

Roles in Software Development:

Role	**Responsibility**
Project Manager	Plans, monitors, and controls the project.

Role	Responsibility
Software Developer	Writes and maintains the code.
Quality Analyst (Tester)	Ensures the software meets quality standards.
Business Analyst	Gathers and analyzes requirements.
Customer/Stakeholder	Defines project goals and requirements.

Example: In a **mobile banking app**, the **bank (customer)** provides requirements, **developers** build the app, and **testers** ensure security and reliability.

---

2. Process (How is the software developed?)

✓ Defines **methods, frameworks, and models** used to develop software.
✓ Ensures **efficiency, quality, and reliability**.

Common Software Development Processes:

Process Model	Description	Example Use Case
Waterfall Model	Linear, step-by-step development	Fixed-scope projects like Payroll Systems.
Agile Model	Iterative development with frequent updates	Startups, dynamic projects like e-commerce apps.
V-Model	Testing at every stage	Safety-critical software like medical devices.
Spiral Model	Risk-driven iterative model	High-risk projects like banking software.

Example: A company developing an **online food delivery app** may follow the **Agile process**, delivering updates in **small sprints** based on customer feedback.

---

3. Product (What is being developed?)

✓ The **final software system** that is delivered to users.
✓ Must meet **functional & non-functional requirements**.

Characteristics of a Good Software Product:

Characteristic	Description
Functionality	Performs the required tasks correctly.

Characteristic	Description
Usability	Easy for users to navigate and interact with.
Performance	Responds quickly and handles multiple users.
Security	Protects user data from threats.
Maintainability	Can be updated and improved over time.

Example:

- **Google Search Engine** is a product that provides **fast and accurate search results**.
- **Microsoft Word** is a product that helps users **create and edit documents**.

Importance of 3Ps in Software Engineering

✓ Ensures **efficient and structured** software development.
✓ Helps in **team coordination and project success**.
✓ Leads to **high-quality software** that meets user needs.

Q. Explain Software Project Estimation

Software Project Estimation is the process of predicting the **effort, time, cost, and resources** required to complete a software project. It helps in **planning, budgeting, and risk management**, ensuring that projects are delivered successfully.

Why is estimation important?

- Ensures **realistic deadlines** and **budget allocation**.
- Helps in **resource planning** (manpower, tools, infrastructure).
- Reduces **project failure risks** by setting achievable goals.

Key Factors in Software Project Estimation

Several factors influence project estimation:

✓ **Project Scope** → Clearly defines features & functionalities.
✓ **Size of the Software** → Measured in **LOC (Lines of Code) or Function Points**.
✓ **Complexity** → Affects development time & cost.
✓ **Resources Required** → Developers, testers, hardware, tools.
✓ **Experience of the Team** → Skilled teams complete tasks faster.

Software Project Estimation Techniques

There are several techniques for estimating software projects:

1. Lines of Code (LOC) Estimation

✓ Estimates project size by counting the **number of lines of code**.
✓ Used to calculate **effort, time, and cost**.

Example:

- A simple login system may require **500 LOC**, while a banking system may need **100,000 LOC**.

Formula:
Effort (Person-Months) = (LOC / Productivity Rate)
(Where Productivity Rate is the number of LOC a developer can write in a month.)

Limitations: Difficult to estimate for Agile projects where requirements change frequently.

2. Function Point (FP) Estimation

✓ Measures the **functionality of the system** rather than LOC.
✓ Counts **inputs, outputs, and interactions** to estimate effort.

Example:

Function Type	Weight (Complexity)	Count	Total Points
User Inputs	4	5	20
User Outputs	5	3	15
User Queries	4	2	8
Internal Files	7	4	28
External Interfaces	6	3	18
Total FP			**89 FP**

Formula:
Effort = FP × Productivity Factor

Advantages:

- Independent of programming language.
- More **accurate than LOC estimation**.

Limitations: Requires detailed documentation and experience.

3. COCOMO (Constructive Cost Model)

✓ A mathematical model that calculates **effort, time, and cost** based on project size.
✓ Developed by **Barry Boehm**.

Types of COCOMO Models:

Model	Best For	Example
Basic COCOMO	Small projects	Payroll system
Intermediate COCOMO	Medium projects	Inventory management system
Advanced COCOMO	Large, complex projects	ERP system, Banking Software

Formula:
Effort (PM) = a × (KLOC)^b
(Where a and b are constants based on project type.)

Limitations: Not suitable for Agile projects.

4. Expert Judgment Estimation

✓ Uses **past experience** and **expert opinions** to estimate effort.
✓ Works well for **small projects** with similar past projects.

Example:

- A senior developer estimates that a **hotel booking system** will take **6 months** based on past experience.

Limitations: Can be **subjective** and depend on expert bias.

5. Delphi Estimation Technique

✓ **Group-based estimation method** where multiple experts **give independent estimates**.
✓ Estimates are **discussed, refined, and averaged** to reach a consensus.

Example:

- Three developers estimate a feature's effort:
  - Developer 1: **10 days**
  - Developer 2: **15 days**
  - Developer 3: **12 days**
- The final estimate is the **average (12 days).**

Advantages:

- Reduces **individual bias**.
- Works well for **complex projects**.

Limitations: Time-consuming and requires **multiple expert discussions**.

Importance of Software Project Estimation

✓ Helps in **realistic project planning**.
✓ Ensures **cost-effective** software development.
✓ Reduces **project failure risks**.
✓ Helps in **allocating resources efficiently**.

Q. Explain Lines of Code (LOC)

Lines of Code (LOC) is a **software size estimation technique** that measures the size of a software project by counting the **number of lines in the source code**. It is used for estimating **effort, cost, time, and productivity** in software development.

Purpose: Helps in **project planning, resource allocation, and productivity measurement**.
Used In: Effort estimation, productivity measurement, and defect analysis.

How LOC is Measured?

The total number of **Lines of Code (LOC)** includes:
✓ **Executable Statements** → Actual code written in functions and modules.
✓ **Declarations** → Variable and class declarations.
✓ **Control Statements** → Loops, conditions (if, else, for, while).

Types of LOC

1. Physical LOC (PLOC)

✓ Counts **all lines**, including comments and blank lines.
✓ Easier to calculate but **not very accurate**.

---

2. Logical LOC (LLOC)

✓ Counts **only executable statements**, ignoring comments and blank lines.
✓ More **accurate for effort estimation**.

LOC-Based Effort Estimation

Effort Estimation Formula:
Effort (Person-Months) = LOC / Productivity Rate

Where:

- **LOC** = Total lines of code.
- **Productivity Rate** = Lines of code a developer can write in a month (Varies from 500 to 1000 LOC per month).

Example Calculation:

- Estimated LOC = **10,000**
- Productivity Rate = **1000 LOC per month**
- **Effort = 10,000 / 1000 = 10 Person-Months**

Development Cost = Effort × Cost per Developer per Month

Advantages of LOC Estimation

✓ **Simple & Easy to Measure** → Requires only a code count.
✓ **Useful for Productivity Measurement** → Helps in comparing developers' efficiency.
✓ **Widely Used in COCOMO Model** → Used for cost estimation in project planning.

Disadvantages of LOC Estimation

✓ **Language Dependent** → Different languages require different LOC for the same task (Python vs. Java).
✓ **Encourages Inefficient Coding** → Developers may write **longer code** instead of optimized code.
✓ **Not Suitable for Modern Development** → Agile and Object-Oriented projects focus on reusability rather than LOC.

Q. Explain Function Point (FP)

Function Point (FP) is a **software size estimation technique** that measures the **functionality of a system** instead of counting **lines of code (LOC)**. It is **independent of programming language** and focuses on the **user's perspective** rather than technical details.

Developed by Allan J. Albrecht at IBM in 1979.
Used for: Effort estimation, cost estimation, and productivity measurement.

Why Use Function Points Instead of LOC?

LOC measures code length, but different languages require different LOC for the same functionality.
FP measures system functionality, making it **language-independent and more accurate** for estimation.

Example:

- A login feature in **Python** may take **50 LOC**, but in **Java**, it may take **100 LOC**.
- **Function Points remain the same** regardless of the language.

Function Point Components

FP estimation is based on **five major components:**

Component	Description	Example
External Inputs (EI)	User inputs that modify system data	Login form, Registration form
External Outputs (EO)	Processed information provided to the user	Sales report, Invoice generation
External Inquiries (EQ)	Requests that retrieve data without modifying it	Search queries, User profile lookup
Internal Logical Files (ILF)	Data stored and used within the system	Customer database, Employee records
External Interface Files (EIF)	Data accessed from external systems	Payment gateway API, Third-party inventory

---

Function Point Estimation Steps

Step 1: Count Function Types & Assign Weights

Each function type is classified as **Simple, Average, or Complex**, and assigned a weight.

Weight Table (Standard IFPUG Weights):

Function Type	Simple	Average	Complex
External Inputs (EI)	3	4	6
External Outputs (EO)	4	5	7

Function Type	Simple	Average	Complex
External Inquiries (EQ)	3	4	6
Internal Logical Files (ILF)	7	10	15
External Interface Files (EIF)	5	7	10

Step 2: Calculate Unadjusted Function Points (UFP)

Formula:

UFP = ∑ (Function Count × Weight)

Example Calculation:

Function Type	Count	Complexity	Weight	Total
External Inputs (EI)	5	Average	4	20
External Outputs (EO)	3	Complex	7	21
External Inquiries (EQ)	2	Simple	3	6
Internal Logical Files (ILF)	4	Average	10	40
External Interface Files (EIF)	3	Simple	5	15
Total UFP	-	-	-	**102**

Step 3: Apply Complexity Adjustment Factor (CAF)

✓ The **Complexity Adjustment Factor (CAF)** is calculated based on **14 general system characteristics** (such as reliability, performance, security, etc.).
✓ Each characteristic is rated from **0 (No impact) to 5 (Strong influence)**.

Formula:

CAF = 0.65 + (0.01 × Sum of All Characteristic Scores)

Example: If the total characteristic score is **30**, then:

CAF = 0.65 + (0.01 × 30) = 0.95

Step 4: Calculate Adjusted Function Points (AFP)

Formula:

AFP=UFP×CAF

Example:

AFP = 102 × 0.95 = 96.9 ≈ 97 Function Points

Effort Estimation Using Function Points

Once function points are determined, **effort estimation** can be done using:

Effort (Person-Months) = Function Points × Productivity Factor

Example Calculation:

- **Function Points = 97**
- **Productivity Factor = 2.5 (Industry standard varies between 2-5 FP per month)**

Effort = 97 × 2.5 = 242.5 Person-Months

Cost Estimation:

Cost = Effort × Developer Salary Per Month

Advantages of Function Point Analysis

✓ **Language-Independent** → Works for **Java, Python, C++, etc.**.
✓ **More Accurate than LOC** → Measures **functionality instead of code size**.
✓ **Better for Agile Development** → Can estimate effort for evolving projects.
✓ **Improves Productivity Measurement** → Helps in **team performance evaluation**.

Disadvantages of Function Point Analysis

✓ **Complex to Calculate** → Requires experience to classify functions correctly.
✓ **Subjective Estimation** → Different analysts may assign different weights.
✓ **Not Suitable for Small Projects** → Overhead of calculation for small applications.

Q. Explain COCOMO II

COCOMO II (Constructive Cost Model II) is an advanced **software cost estimation model** that predicts the **effort, time, and cost** required for software development. It is an **improved version of COCOMO** developed by **Barry Boehm in 1997**, designed to handle **modern software development practices like Agile and Object-Oriented Programming (OOP).**

Why COCOMO II?

- Supports **modern development methodologies** (Agile, Component-Based, etc.).

- Accounts for **reusability, risk assessment, and adaptability**.
- More **accurate than traditional COCOMO**.

COCOMO II Models

COCOMO II consists of **three submodels**, each used at different **stages of software development**:

COCOMO II Model	Purpose	When Used?
Application Composition Model	Estimates effort for projects using **existing components & rapid prototyping**	Early stage of development
Early Design Model	Estimates effort when **requirements are not fully defined**	After initial requirement gathering
Post-Architecture Model	Provides the **most accurate cost estimate**	After detailed system architecture is designed

---

Effort Estimation Formula in COCOMO II

COCOMO II uses the following general formula to estimate effort:

Effort (Person-Months) = A × $(\text{Size})^n$ × EAF

Where:

- **A** = Constant (Varies based on project type).
- **Size** = Measured in **KSLOC (Kilo Source Lines of Code)** or **Function Points (FP)**.
- **n** = Exponent (Depends on project complexity).
- **EAF (Effort Adjustment Factor)** = Adjusts effort based on 17 cost drivers (e.g., team experience, reliability).

Example Calculation:

- If **A = 2.94**, **Size = 50 KSLOC**, **n = 1.10**, and **EAF = 1.2**, then:

Effort = 2.94 × (50) ^ 1.10 × 1.2 = 2.94 × 62.09 × 1.2 = 218.8 Person-Months

---

COCOMO II Cost Drivers

COCOMO II adjusts effort based on **17 cost drivers**, categorized into **four main groups**:

Category	Cost Drivers	Example Impact
Product Factors	Required Reliability, Complexity, Reusability	High security systems need more effort
Platform Factors	Execution Time, Memory Constraints	Real-time applications require more optimization
Personnel Factors	Analyst Capability, Programmer Experience	Experienced teams reduce effort
Project Factors	Development Flexibility, Tools Usage	Using DevOps tools speeds up development

Example: If a banking system needs **high security**, the **reliability cost driver increases**, increasing total effort.

Advantages of COCOMO II

✓ **Works for modern development models** (Agile, Component-Based, Object-Oriented).
✓ **More accurate than traditional COCOMO**.
✓ **Adjusts based on project complexity & team skills**.
✓ **Helps in risk management & resource allocation**.

Disadvantages of COCOMO II

✓ **Complex calculations** require historical data and expert analysis.
✓ **Difficult to use for small projects** due to multiple cost drivers.
✓ **Accuracy depends on input quality** (wrong assumptions lead to incorrect estimates).

Q. Explain COCOMO

COCOMO (**Constructive Cost Model**) is a **software cost estimation model** developed by **Barry Boehm in 1981**. It predicts **effort, time, and cost** required for software development based on the **size of the project (Lines of Code - LOC)**.

Why use COCOMO?

- Helps in **budget estimation and project planning**.
- Provides a **mathematical formula** for effort estimation.
- Classifies projects into **three categories** based on complexity.

Types of COCOMO Models

COCOMO has **three types of models**, depending on project complexity:

COCOMO Model	Best For	Example
Basic COCOMO	Simple, small projects with well-defined requirements	Payroll system, small inventory system
Intermediate COCOMO	Medium-sized projects with some complexity and risk	ERP system, E-commerce websites
Advanced (Detailed) COCOMO	Large, complex projects with multiple teams	Banking software, Aerospace systems

---

Basic COCOMO Model

The **Basic COCOMO Model** calculates **effort, development time, and number of developers** based on **Lines of Code (LOC)**.

Basic COCOMO Effort Estimation Formula

✓ **Effort (Person-Months) = a × (KLOC)^b**
✓ **Time (Months) = c × (Effort)^d**
✓ **Number of Developers = Effort / Time**

Where:

- **KLOC** = Thousands of Lines of Code (LOC / 1000).
- **a, b, c, d** = Constants based on project type.

COCOMO Constants for Different Project Types:

Project Type	a	b	c	d
Organic (Simple)	2.4	1.05	2.5	0.38
Semi-Detached (Medium)	3.0	1.12	2.5	0.35
Embedded (Complex)	3.6	1.20	2.5	0.32

Example Calculation for Basic COCOMO

Problem Statement:
Estimate the effort and development time for a **Semi-Detached project** with **50,000 LOC (50 KLOC)**.

Step 1: Calculate Effort

Effort = 3.0 × (50) ^ 1.12 = 3.0 × 65.8 = 197.4 Person-Months

Step 2: Calculate Development Time

Time = 2.5 × (197.4) ^ 0.35 = 2.5 × 6.36 = 15.9

Step 3: Calculate Number of Developers

Developers = 197.4 / 15.9 = 12.4 ≈ 12 developers

Final Answer:

- **Effort:** 197.4 person-months
- **Development Time:** 15.9 months
- **Team Size:** 12 developers

Intermediate COCOMO Model

✓ The **Intermediate COCOMO Model** improves Basic COCOMO by considering **cost drivers** (e.g., team experience, project complexity).
✓ Uses an **Effort Adjustment Factor (EAF)** to refine effort estimation.

Formula:

Effort = a × (KLOC) ^ b × EAF

Where **EAF** is calculated using **15 cost drivers**, such as:

- **Product Complexity** (Simple, Moderate, High).
- **Team Experience** (Low, Medium, High).
- **Development Tools** (Basic, Advanced).

Example:

- If **EAF = 1.2**, then **Effort = 197.4 × 1.2 = 236.9 Person-Months**.

Detailed (Advanced) COCOMO Model

✓ **Most accurate** COCOMO model.
✓ Divides the project into **multiple modules** and estimates effort for each one separately.
✓ Includes **all cost drivers** for more precise calculations.

Best suited for:

- Large-scale **enterprise applications**.
- **Aerospace & defense projects**.
- **Real-time and embedded systems**.

Advantages of COCOMO

✓ **Provides structured cost estimation**.
✓ **Considers project complexity** (Intermediate & Advanced models).
✓ **Useful for large-scale projects**.
✓ **Helps in project planning and resource allocation**.

Disadvantages of COCOMO

✓ **Dependent on accurate LOC estimation**.
✓ **Not suitable for Agile projects** where requirements change frequently.
✓ **Doesn't consider modern software development methodologies** (like DevOps, Cloud Computing).

Q. Explain General format of SRS

Q. Explain Multiple Viewpoints in requirement engineering

In **Requirements Engineering (RE)**, different stakeholders have different **perspectives and needs** for the software system. **Multiple viewpoints** help in gathering **comprehensive and conflict-free** requirements by considering **all stakeholders' perspectives**.

Why are multiple viewpoints important?

- Ensures **no requirement is missed**.
- Helps in **resolving conflicts** between different stakeholders.
- Leads to a **more complete and user-friendly system**.

Types of Viewpoints in Requirement Engineering

1. User Viewpoint (End-Users Perspective)

✓ Focuses on how **users will interact** with the system.
✓ Defines **functional requirements** related to UI, usability, and user experience.

Example:

- In an **e-commerce website**, customers want:
  - **Easy navigation** to browse products.
  - **Quick checkout** with minimal steps.

  - **Order tracking and notifications**.

---

2. Business Viewpoint (Organizational Goals & Policies)

✓ Focuses on the **business objectives and constraints**.
✓ Defines **non-functional requirements** like performance, security, and cost.

Example:

- For an **online banking system**, the bank’s business goals may include:
  - **Increase customer retention** through a user-friendly app.
  - **Ensure compliance** with government regulations (e.g., RBI guidelines).
  - **Minimize operational costs** by reducing manual transactions.

---

3. Developer Viewpoint (Technical Perspective)

✓ Focuses on **technical implementation** and feasibility.
✓ Defines **system architecture, programming languages, and integrations**.

Example:

- A software development team might require:
  - Use of **Java & MySQL** for backend development.
  - **Cloud-based infrastructure (AWS, Azure)** for scalability.
  - API integration with **third-party payment gateways** (PayPal, Stripe).

---

4. Regulatory & Legal Viewpoint (Compliance & Security)

✓ Focuses on **laws, standards, and industry regulations**.
✓ Defines **external non-functional requirements** such as data privacy, security, and accessibility.

Example:

- A **healthcare management system** must:
  - Comply with **HIPAA** (Health Insurance Portability and Accountability Act) for patient data security.
  - Ensure **GDPR compliance** for user data protection in Europe.
  - Follow **ISO 27001 security standards**.

---

5. Operational & Maintenance Viewpoint (System Administration)

✓ Focuses on **system deployment, monitoring, and long-term support**.
✓ Defines **requirements for maintainability, backups, and upgrades**.

Example:

- An **ERP (Enterprise Resource Planning) system** should:
  - Have **automatic backup** every 24 hours.
  - Allow **remote server monitoring** for performance issues.
  - Support **regular software updates** without downtime.

Importance of Multiple Viewpoints in Requirements Engineering

✓ **Ensures all perspectives are covered**, leading to **complete requirements**.
✓ **Reduces conflicts** between stakeholders by identifying and resolving **contradictory needs** early.
✓ **Improves software quality** by considering **usability, security, performance, and business goals**.