# oops/cn/OS

Here is a **complete interview question bank for OOPS, Operating Systems (OS), and Computer Networks (CN)** — exactly what is asked for **1–3 years full-stack developer roles**.

---

# 🔥 OOPS (Object-Oriented Programming)

## 📌 Fundamentals

---

# ✅ 1) What is OOP? Why do we need it?

### ■ Explanation (simple, conceptual)

OOP (Object-Oriented Programming) is a programming approach where software is organized around **objects**—real-world entities that contain **data (attributes)** and **behavior (methods)**.

It allows us to build applications as a collection of interacting objects rather than writing logic in a procedural manner.

### ■ When/Why is it used

- To build **modular, maintainable, scalable** applications
- To reduce code duplication and improve reusability
- To represent real-world problems naturally (User, Product, Payment...)
- For clean architecture in large projects (layers, services, models)
- To support principles like abstraction, encapsulation, inheritance, and polymorphism

### ■ Example

A Banking system:

- Customer → object
- Account → object

- Loan → object

  Each object has its own properties and operations.

■ **Short summary line**

**OOP models software as interacting objects and helps create maintainable, reusable, and scalable applications.**

---

# ✅ 2) Explain the 4 pillars of OOP.

■ **Explanation (simple, conceptual)**

The 4 pillars of OOP define the fundamental building blocks for object-oriented design.

---

## 🔵 1. Encapsulation (Data Hiding)

Binding data + methods inside a class and controlling access using access modifiers (private, public).

Prevents unwanted modification.

**Example:**

```
public class BankAccount
{
    private double balance;   // hidden
    public void Deposit(double amount) => balance += amount;
}
```

---

## 🔵 2. Abstraction (Hiding Complexity)

Showing only essential features and hiding internal logic.

**Example:**

```
account.Transfer(500);
```

We don't see validation logic, DB calls, etc.

---

## 🔵 3. Inheritance (Reusability)

Creating a new class from an existing class.

**Example:**

```
class Employee {}
class Manager : Employee {}
```

---

## 🔵 4. Polymorphism (Many Forms)

Same method behaves differently based on the object.

**Example:**

Overriding:

```
public override void CalculateSalary() { }
```

### ■ When/Why used

- Encapsulation → secure data
- Abstraction → simplify usage
- Inheritance → reuse base logic
- Polymorphism → flexible behavior

### ■ Short summary line

**The 4 pillars—encapsulation, abstraction, inheritance, and polymorphism—form the foundation of object-oriented design.**

---

# ✅ 3) What is Class and Object? Example.

### ■ Explanation (simple, conceptual)

**Class** → Blueprint or template

**Object** → Actual instance created from the class

Class defines properties + behavior; object uses them.

### ■ When/Why used

- Class helps structure code

- Objects hold real data and perform actions
- Essential for organizing layers (Models, DTOs, Services in .NET/Java)

■ **Example**

```
public class Car
{
    public string Brand;
    public void Drive() { }
}

// Object
Car c1 = new Car();
c1.Brand = "BMW";
c1.Drive();
```

■ **Short summary line**

**Class is a blueprint; object is a real instance created from that blueprint.**

---

# ✅ 4) Difference Between Abstraction vs Encapsulation

■ **Explanation (simple, conceptual)**

These two principles are often confused but are fundamentally different:

| Feature | Abstraction | Encapsulation |
|---|---|---|
| Purpose | Hide complexity | Hide data |
| Focus | What object does | How object stores data |
| Achieved using | Abstract classes, interfaces | Access modifiers, properties |
| Example | `List.Add()` hides internal logic | Private fields with getters/setters |

■ **When/Why used**

## 🔵 Abstraction

Used when we want to expose only required functionality.

Example: Service Interfaces (`IUserService`, `IOrderService`).

### 🔵 Encapsulation

Used to protect data from outside modification.

Example: Private fields in DTOs or Models.

#### ⬛ Example

```
// Encapsulation
private int salary;

// Abstraction
public abstract class Payment
{
    public abstract void Process();
}
```

#### ⬛ Short summary line

**Abstraction hides implementation; encapsulation hides data and bundles it with behavior.**

---

# ✅ 5) Inheritance — Types and Use Cases

#### ⬛ Explanation (simple, conceptual)

Inheritance allows one class to **reuse** and **extend** another class's properties/methods.

It reduces duplication and supports logical hierarchy.

#### ⬛ Types of Inheritance

## 1. **Single Inheritance**

One parent → one child

```
class Employee {}
class Manager : Employee {}
```

## 2. **Multilevel Inheritance**

Parent → Child → Subchild

```
class A {}
class B : A {}
class C : B {}
```

### 3. **Hierarchical Inheritance**

One parent → many children

```
class Shape {}
class Circle : Shape {}
class Square : Shape {}
```

### 4. **Multiple Inheritance**

One child inherits from multiple parents

❌ Not allowed in C# / Java for classes

✔ Supported using interfaces

### 5. **Hybrid Inheritance**

Combination of multiple + multilevel

✔ Possible using interfaces

■ **When/Why used**

### ✔ Reuse common logic

Example: BaseEntity with `CreatedAt`, `UpdatedAt`.

### ✔ Build clean hierarchies

Example:

`Vehicle → Car, Truck, Bike`.

### ✔ Override behavior using polymorphism

Example:

Different tax calculation for different employee roles.

■ **Example**

```
public class Animal
{
    public virtual void Sound() {}
}

public class Dog : Animal
{
    public override void Sound() => Console.WriteLine("Bark");
}
```

■ **Short summary line**

**Inheritance promotes code reusability, hierarchy creation, and polymorphic behavior.**

---

# ✅ 6) Compile-time vs Runtime Polymorphism

■ **Explanation (simple, conceptual)**

Polymorphism means **one function, many forms**.

It occurs in two ways:

### 🔵 Compile-Time Polymorphism (Early Binding)

Decision is made at **compile time**.

Achieved using **method overloading**.

### 🔵 Runtime Polymorphism (Late Binding)

Decision is made at **runtime**.

Achieved using **method overriding** via inheritance.

---

■ **When/Why used**

### ✔ Compile-Time Polymorphism

- Improve readability
- Same method name for different parameters

- Helpful in utility/helper classes

### ✔️ Runtime Polymorphism

- Implement dynamic behavior
- Override methods in child classes
- Used in service architecture (e.g., PaymentService → UPI/Card/COD)

#### ■ Example

```
// Compile-time polymorphism
public int Add(int a, int b) => a + b;
public int Add(int a, int b, int c) => a + b + c;

// Runtime polymorphism
public class Animal { public virtual void Sound() {} }
public class Dog : Animal { public override void Sound() => Console.WriteLi
ne("Bark"); }
```

#### ■ Short summary line

**Overloading = compile-time polymorphism; overriding = runtime polymorphism.**

# ✅ 7) Method Overloading vs Overriding

#### ■ Explanation (simple, conceptual)

| Feature | Overloading | Overriding |
|---|---|---|
| When decided | Compile time | Runtime |
| Where used | Same class | Parent–child classes |
| Purpose | Multiple versions of same method | Change base class behavior |
| Parameters | Must differ | Must be same |
| Keywords | No keyword | `override` (C#) |

#### ■ When/Why used

### ✔️ Overloading

- For flexible APIs
- Same method name with different inputs
- Example: `Print(int)` and `Print(string)`

### ✔️ Overriding

- To provide custom behavior
- Useful in polymorphism
- Example: `CalculateTax()` different for Employee and Manager

---

#### ■ Example

```
// Overloading
void Log(string message)
void Log(string message, int level)

// Overriding
class Base { public virtual void Show() {} }
class Derived : Base { public override void Show() {} }
```

#### ■ Short summary line

**Overloading changes parameters; overriding changes behavior.**

---

# ✅ 8) Can we overload constructor? Why use it?

#### ■ Explanation (simple, conceptual)

Yes — constructor overloading means having **multiple constructors with different parameters** in the same class.

#### ■ When/Why used

- To provide **different ways to initialize an object**
- To set default values
- To handle optional parameters cleanly
- To improve usability and flexibility

---

### ⬛ Example

```
public class User
{
    public string Name;
    public int Age;

    public User()              { Name = "Guest"; Age = 0; }
    public User(string name)    { Name = name; }
    public User(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

### ⬛ Short summary line

**Yes, constructors can be overloaded to create objects in multiple flexible ways.**

---

# ✅ 9) What is `this` keyword?

### ⬛ Explanation (simple, conceptual)

`this` is a keyword that refers to the **current object** of the class.

Used to access instance fields, methods, and constructors.

### ⬛ When/Why used

- To differentiate between class fields and method parameters
- To call one constructor from another ( `this()` )
- To pass current object as a parameter

---

### ⬛ Example

```
public class Employee
{
    private string name;
```

```
    public Employee(string name)
    {
        this.name = name; // using this to refer to field
    }
}
```

Constructor chaining:

```
public Employee() : this("Unknown") {}
```

■ **Short summary line**

**`this` refers to the current object and is used for field access and constructor chaining.**

---

# ✅ 10) What is `super` keyword?

■ **Explanation (simple, conceptual)**

`super` (Java) / `base` (C#) refers to the **parent class**.

It is used to access parent class fields, methods, or constructors.

■ **When/Why used**

- To call parent constructor
- To reuse parent logic
- To avoid code duplication
- To override methods but still call parent implementation

---

■ **Example (C# using `base`)**

```
public class Parent
{
    public Parent() { Console.WriteLine("Parent constructor"); }
    public virtual void Display() { Console.WriteLine("Parent display"); }
}

public class Child : Parent
{
```

```
    public Child() : base() {} // calling Parent constructor
    public override void Display()
    {
        base.Display(); // calling Parent display
        Console.WriteLine("Child display");
    }
}
```

### ■ Short summary line

**`super` / `base` refers to the parent class and is used to call parent constructors and methods.**

---

## 📌 Advanced & Interview-Focused

---

# ✅ 1) Abstract Class vs Interface

### ■ Explanation (simple, conceptual)

Both are used to achieve **abstraction**, but they differ in purpose:

### 🔵 Abstract Class

- Can contain **abstract + non-abstract** methods
- Can have **fields, constructors, and default implementations**
- Supports **single inheritance**

### 🟣 Interface

- Defines **only behavior/contract**
- Contains only **abstract methods** (C# also allows default/static methods)
- Supports **multiple inheritance**

### ■ When/Why used

### ✔ Use **Abstract Class** when:

- You want a common base class with shared code
- You want to enforce partial abstraction

- Child classes share common state or fields

  Example: `BaseController`, `BaseEntity`

### ✔️ Use **Interface** when:

- You want to define a contract
- Multiple unrelated classes must follow the same behavior

  Example: `ILogger`, `IRepository`, `IDisposable`

#### ■ Example

```
public abstract class Animal
{
    public abstract void Sound();
    public void Sleep() { }
}

public interface IWalk
{
    void Walk();
}
```

#### ■ Short summary line

**Abstract class = shared base with partial implementation, Interface = contract for multiple unrelated classes.**

---

# ✅ 2) Multiple Inheritance — Supported or Not?

#### ■ Explanation (simple, conceptual)

Multiple inheritance means a class can inherit from **more than one parent class**.

In C# and Java:

❌ **Multiple inheritance of classes is NOT allowed.**

But:

✔️ **Multiple inheritance using interfaces IS allowed.**

■ **When/Why used**

Avoid ambiguity and complexity from multiple parents (diamond problem).

Use interfaces for behavior composition.

■ **Example**

```
public interface IRun { void Run(); }
public interface IFly { void Fly(); }

public class Bird : IRun, IFly { }
```

■ **Short summary line**

**Classes cannot have multiple parents, but they can implement multiple interfaces.**

---

# ✅ 3) Diamond Problem — How Solved?

■ **Explanation (simple, conceptual)**

Diamond Problem occurs when:

Two parent classes inherit from the same base class →

A child class inherits both parents →

Compiler becomes confused about which parent’s method to use.

Since **multiple class inheritance is not allowed**, C# and Java avoid this problem entirely.

In languages that allow it (C++), virtual inheritance solves it.

■ **When/Why used**

- To prevent ambiguity in method resolution
- To avoid complexity in class hierarchies

■ **Example (Why C# avoids it)**

```
// Not allowed in C#
class A {}
class B : A {}
```

```
class C : A {}
class D : B, C {} // error
```

■ **Short summary line**

**Diamond problem happens due to multiple class inheritance—C# solves it by not allowing multiple class inheritance.**

---

# ✅ 4) Access Modifiers — public / private / protected / internal

■ **Explanation (simple, conceptual)**

Access modifiers control **visibility** of classes and members.

### 🔵 public

Accessible everywhere.

### 🔵 private

Accessible only within the same class.

### 🔵 protected

Accessible in the same class + derived classes.

### 🔵 internal

Accessible only within the **same assembly/project**.

### 🔵 protected internal

Accessible within same assembly **OR** derived classes.

### 🔵 private protected

Accessible within same class **OR** derived classes in same assembly.

■ **When/Why used**

- private → to hide data
- protected → allow child classes to use base members
- internal → expose APIs inside project only

- public → expose behavior to entire application

■ **Example**

```
public class BankAccount
{
    private double balance;          // hidden
    protected void Calculate() { }    // child classes can access
    internal int id;                 // same project
}
```

■ **Short summary line**

**Access modifiers decide where a class/member is visible, ensuring security and good design.**

---

# ✅ 5) Static vs Non-static Methods

■ **Explanation (simple, conceptual)**

## 🔵 Static Methods

- Belong to the class, NOT objects
- Called using class name
- Cannot access instance members
- Memory allocated once

## 🟣 Non-static Methods

- Belong to object instances
- Can access both static & instance members

■ **When/Why used**

## ✔ Use static when:

- Method does not depend on object data
- Utility, helper, factory methods
- Example: `Math.Max()`, Logger utilities

### ✔️ Use non-static when:

- Behavior depends on object-specific data
- Need to work with instance fields
- Example: `account.Deposit(100);`

■ **Example**

```
public class Calculator
{
    public static int Add(int a, int b) => a + b; // no instance needed
    public int Multiply(int a, int b) => a * b;   // object-specific behavior
}
```

■ **Short summary line**

**Static methods belong to class; non-static methods belong to objects and access instance data.**

---

# ✅ 6) What is a Pure Virtual Function / Abstract Method?

■ **Explanation (simple, conceptual)**

A **pure virtual function** (C++) or **abstract method** (C#, Java) is a method that has **no implementation** in the base class and **must be implemented** by derived classes.

It forces child classes to provide their own behavior.

■ **When/Why is it used**

- To achieve full abstraction
- To enforce a contract among child classes
- To define a common method name but allow different implementations
- Used in frameworks, services, interfaces, and base classes

■ **Example**

```
public abstract class Shape
{
    public abstract void Draw();  // abstract method
}

public class Circle : Shape
{
    public override void Draw() => Console.WriteLine("Drawing Circle");
}
```

■ **Short summary line**

**Abstract methods define a contract with no implementation—child classes must override them.**

---

# ✅ 7) IS-A vs HAS-A Relationship

■ **Explanation (simple, conceptual)**

## 🔵 IS-A (Inheritance Relationship)

Represents **"A is a type of B"**

Implemented using **inheritance**.

Example:

`Manager IS-A Employee`

## 🔵 HAS-A (Composition/Aggregation)

Represents **"A has a B"**

Implemented using **object references inside classes**.

Example:

`Car HAS-A Engine`

■ **When/Why used**

## ✔ IS-A

- When child class is a **specialized version** of parent
- For reuse and polymorphism

### ✔️ HAS-A

- For building complex objects from smaller objects
- Preferred over inheritance (more flexible)

■ **Example**

```
// IS-A
class Dog : Animal { }

// HAS-A
class Car
{
    Engine engine = new Engine();
}
```

■ **Short summary line**

**IS-A means inheritance; HAS-A means composition—representing ownership or usage.**

---

# ✅ 8) What is Object Composition?

■ **Explanation (simple, conceptual)**

Object Composition means creating complex classes by **combining objects of other classes** rather than using inheritance.

It is also known as **"HAS-A" relationship**.

■ **When/Why used**

- Preferred over inheritance because it avoids tight coupling
- Promotes reusability and flexibility
- Allows behavior change at runtime by replacing components
- Used in modern architectures (DI containers, service classes)

■ **Example**

```
public class Car
{
```

```
    private Engine engine;   // Car HAS-A Engine
    public Car(Engine e) ⇒ engine = e;
}
```

■ **Short summary line**

**Composition builds complex objects by using other objects, providing flexibility and loose coupling.**

---

# ✅ 9) SOLID Principle — Explain Each Rule

■ **Explanation (simple, conceptual)**

SOLID is a set of 5 design principles that improve object-oriented design and make software maintainable.

---

## 🔵 S — Single Responsibility Principle (SRP)

A class should have **one reason to change** (one responsibility).

**Why:** Avoids bloated classes.

**Example:**

`InvoiceService` (calculations) should not send emails → put email logic in `EmailService`.

---

## 🔵 O — Open/Closed Principle (OCP)

Classes should be **open for extension, closed for modification**.

**Why:** Extend behavior without touching old code.

**Example:**

Use interfaces or strategy pattern to add new payment methods without modifying existing logic.

---

## 🔵 L — Liskov Substitution Principle (LSP)

Child class should **replace** parent class without breaking functionality.

**Why:** Avoid runtime exceptions when using polymorphism.

**Example:**

If `Bird` has `Fly()`, `Penguin` should NOT inherit from `Bird` (because it cannot fly).

---

## 🔵 I — Interface Segregation Principle (ISP)

No class should be forced to implement **unnecessary methods**.

**Why:** Create smaller, focused interfaces.

**Example:**

Instead of `IWorker` with Work + Eat, create `IWorkable` and `IEatable`.

---

## 🔵 D — Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules; both should depend on **abstractions**.

**Why:** Decoupling, flexibility, unit testing.

**Example:**

Use interfaces + DI container:

```
public class PaymentService
{
    private readonly IPaymentGateway _gateway;
    public PaymentService(IPaymentGateway gateway)
    {
        _gateway = gateway;
    }
}
```

### ⬛ Short summary line

**SOLID improves code flexibility, maintainability, testability, and reduces coupling.**

---

# ✅ 10) Cohesion vs Coupling — Which Is Preferred?

### ⬛ Explanation (simple, conceptual)

## 🔵 Cohesion

- How **focused** a class/module is
- High cohesion = class has **one purpose**
- Example: `EmailService` only sends emails

## 🟣 Coupling

- How much one class **depends on** another
- Low coupling = loose dependency
- Example: Using interfaces instead of concrete classes

### ■ When/Why used

## ✔ Prefer:

- **High cohesion**
- **Low coupling**

## Why?

- Easier to maintain
- Better reuse
- Easier testing
- Better architecture

### ■ Example

```
// High cohesion, low coupling
public interface INotification { void Send(); }

public class EmailNotification : INotification
{
    public void Send() { }
}
```

### ■ Short summary line

**High cohesion and low coupling produce clean, maintainable, and scalable software.**

---

## 📌 Real-time OOP usage

---

# ✅ 1) How OOPS fits in real project architecture?

### ◼️ Explanation (simple, conceptual)

OOPS is the foundation for designing **layered, modular, and scalable** enterprise applications.

Modern backend applications (.NET/Java) use OOPS to break the system into components like:

- **Models (Entities, DTOs)**
- **Services (Business logic)**
- **Repositories (DB operations)**
- **Controllers (API layer)**

Each component is represented as **classes and objects**, and OOPS principles ensure separation, reusability, and testability.

---

### ◼️ When/Why is it used

- Controllers should NOT contain business logic → handled via abstraction
- Services should NOT talk directly to DB → handled via encapsulation + repositories
- Common rules should be reusable → inheritance
- Different implementations for same behavior → polymorphism
- Complex systems should hide internal details → abstraction in service/repository layers

Essential for:

✔️ Enterprise APIs

✔️ Microservices

✔️ Domain-driven design

✔️ Clean architecture

---

## ⬛ Example / Real scenario (Your .NET Expense Tracker + BofA Project)

### 🔷 Controller Layer (OOPS → abstraction + clean boundary)

```
public class ExpenseController : ControllerBase
{
    private readonly IExpenseService _service;
    public ExpenseController(IExpenseService service) { _service = service; 
}

    public async Task<IActionResult> AddExpense(ExpenseDto dto)
    {
        await _service.AddExpense(dto);
        return Ok();
    }
}
```

Controller does NOT know how expense is stored → OOPS abstraction.

---

### 🔷 Service Layer (OOPS → Abstraction + Polymorphism)

```
public class ExpenseService : IExpenseService
{
    private readonly IExpenseRepository _repo;

    public async Task AddExpense(ExpenseDto dto)
    {
        // business rules + validations
        await _repo.Add(dto);
    }
}
```

---

### 🔷 Repository Layer (OOPS → Encapsulation + Composition)

```
public class ExpenseRepository : IExpenseRepository
{
    private readonly DbContext _ctx;

    public Task Add(ExpenseDto dto)
```

```
    {
        // all SQL or EF code is encapsulated here
    }
}
```

---

⬛ **Short summary line**

**OOPS organizes a real project into controllers, services, repositories, and models to keep code modular, reusable, and maintainable.**

---

# ✅ 2) Where did you use abstraction in your project?

⬛ **Explanation (simple, conceptual)**

Abstraction hides unnecessary implementation details and exposes only required operations.

You work with **interfaces instead of concrete logic**, so high-level layers don't need to know how low-level code works.

---

⬛ **When/Why is it used**

- Hide database implementation from controllers
- Prevent exposing business rules
- Secure internal logic
- Allow easy testing + swapping implementations
- Reduce coupling
- Support dependency injection

---

⬛ **Your project-specific examples (VERY GOOD FOR INTERVIEW)**

### 🔷 Abstraction in Service Layer

You exposed only *what* the service does, not *how* it does it.

```
public interface ITransactionService
{
```

```
    Task<bool> ProcessTransaction(TransactionDto dto);
}
```

Controller uses this abstract contract → does NOT know payment validation logic or DB queries.

---

### 🔷 Abstraction in Repository Layer

```
public interface IExpenseRepository
{
    Task AddExpense(Expense expense);
}
```

Concrete class hides SQL/EF logic, so service layer interacts abstractly.

---

### 🔷 Abstraction in Validation and Utilities

Example:

You created an abstraction for logging/error handling, while implementation changed between local → cloud.

---

#### ■ Short summary line

**You used abstraction by defining interfaces in service/repository layers, hiding internal logic from controllers and ensuring loose coupling.**

---

# ✅ 3) Why do we use interface for service layer in .NET/Java?

#### ■ Explanation (simple, conceptual)

Interfaces define *what* a class should do, not *how* it should do it.

Service layers use interfaces to allow flexible, loosely coupled architectures.

---

#### ■ When/Why is it used

### ✔ 1. Loose Coupling

Controller depends on `IUserService` instead of `UserService`.

Easier to change implementation without breaking code.

## ✔ 2. Dependency Injection

DI containers (like .NET Core) work best with interfaces.

```
services.AddScoped<IUserService, UserService>();
```

## ✔ 3. Testability

Interfaces can be mocked:

```
var mock = new Mock<IUserService>();
```

## ✔ 4. Multiple Implementations

E.g., PaymentService could have:

- UPI implementation
- CreditCard implementation
- NetBanking implementation

## ✔ 5. Clean Architecture

Interfaces help maintain clear boundaries between layers.

---

### ■ Example

```
public interface IUserService
{
    Task<UserDto> GetUser(int id);
}

public class UserService : IUserService
{
    public Task<UserDto> GetUser(int id) { ... }
}
```

Controller depends only on the interface—not the implementation.

---

### ■ Short summary line

**Interfaces give loose coupling, DI support, multiple implementations, and high testability in the service layer.**

---

# ✅ 4) Factory Pattern vs Builder — In Which Scenario?

### ■ Explanation (simple, conceptual)

## 🔵 Factory Pattern

Used when **your program needs to decide which object to create** based on input/logic.

Factory = *object creation decision*.

---

## 🟣 Builder Pattern

Used when **object construction is complex**, involving many steps or optional parameters.

Builder = *step-by-step object creation*.

---

### ■ When/Why used

## ✔ Use Factory when:

- You need to return different concrete classes

  Example: `SMSNotification`, `EmailNotification`, `PushNotification`
- Object choice depends on input
- Want to avoid "newing" objects everywhere

**Your real project example**

Choosing fraud scoring algorithm:

```
IFraudChecker checker = FraudCheckerFactory.Create("RuleBased");
```

---

## ✔ Use Builder when:

- Object has **too many fields**

- Object needs validation between steps
- Many optional fields
- You want readable construction code

**Your real project example**

Building a complex SQL query report or request payload:

```
var request = new PaymentRequestBuilder()
              .WithAmount(1000)
              .WithCustomer("John")
              .WithReference("XYZ")
              .Build();
```

---

### ⬛ Short summary line

**Factory chooses *which* object to create; Builder constructs *how* the object is built step-by-step.**

---

---

# 🔥 OPERATING SYSTEM (OS)

## 📌 Basics

---

# ✅ 1) What is an Operating System?

### ⬛ Explanation (simple, conceptual)

An Operating System (OS) is a system software that acts as an **interface between the user and the hardware**.

It manages CPU, memory, storage, processes, and all input/output devices.

It ensures programs run smoothly by allocating resources and preventing conflicts.

### ⬛ When/Why is it used

- To manage hardware resources efficiently
- To run multiple applications safely
- To provide security and access control

- To handle process scheduling, memory allocation, file storage
- To provide UI/CLI for user interaction

■ **Example**

Windows, Linux, macOS, Android, iOS.

OS handles tasks like:

- Running your browser, VS Code, SQL Server
- Allocating RAM to apps
- Performing disk operations

■ **Short summary line**

**OS manages hardware, provides an interface for users, and controls how programs run.**

---

# ✅ 2) Types of OS — Batch, Multitasking, Real-time

■ **Explanation (simple, conceptual)**

Different OS types are designed for different workloads and environments.

---

## 🔵 1. Batch Operating System

Executes jobs in **batches without user interaction**.

■ **When/Why**

- Used for data processing, payroll, billing
- Efficient for large, repetitive tasks
- No immediate user involvement

■ **Example**

Older IBM mainframe systems, nightly banking batch jobs.

---

## 🔵 2. Multitasking Operating System

Allows multiple tasks/processes to run **seemingly at the same time**.

■ **When/Why**

- For desktops/laptops
- Allows running Chrome + VS Code + Teams simultaneously
- Improves productivity

◼️ **Example**

Windows, Linux, macOS.

---

## 🔵 3. Real-Time Operating System (RTOS)

Responds to events **within strict time constraints**.

Guarantees “predictable response time.”

◼️ **When/Why**

- Used where delays cannot be tolerated
- Mission-critical systems
- Embedded systems

◼️ **Example**

Pacemakers, automotive systems, robots, flight control systems.

◼️ **Short summary line**

**Batch → No interaction, Multitasking → Many tasks, RTOS → Time-critical systems.**

---

# ✅ 3) What is a Kernel?

◼️ **Explanation (simple, conceptual)**

Kernel is the **core component** of an Operating System.

It is responsible for interacting with hardware and managing system resources.

Think of the kernel as the **brain of the OS**.

◼️ **When/Why is it used**

- Manages processes (scheduling, switching)
- Allocates memory
- Controls device drivers
- Handles system calls

- Ensures security and protection

⬛ **Types**

- **Monolithic Kernel** (Linux): Large, high performance
- **Microkernel** (Minix): Smaller, modular, more secure
- **Hybrid Kernel** (Windows): Mix of both

⬛ **Example**

When a program tries to read a file:

Program → System Call → Kernel → Storage Device.

⬛ **Short summary line**

**Kernel is the core of OS that manages CPU, memory, processes, and hardware interactions.**

---

# ✅ 4) System Calls vs Library Calls

⬛ **Explanation (simple, conceptual)**

## 🔵 System Calls

Requests made by programs to the OS kernel.

They switch execution from **user mode to kernel mode**.

Examples:

- `read()`
- `write()`
- `open()`
- `fork()`
- `exec()`

## 🔵 Library Calls

Functions provided by programming libraries (C#, Java, Python).

They run completely in **user mode**.

Examples:

- `printf()`
- `fopen()`
- Math functions
- String functions

---

### ⬛ When/Why used

## ✔️ System Calls

- Needed when interacting with hardware
- Require OS privileges
- Slower due to mode switching

## ✔️ Library Calls

- Faster
- Just code inside a library
- Sometimes internally they call system calls

---

### ⬛ Example

```
printf("hello");
```

This is a **library call**, but internally it uses a **system call** like `write()`.

Another example:

In .NET:

`File.ReadAllText()` → library call

Internally uses OS system calls to read from disk.

---

### ⬛ Short summary line

**System calls interact with kernel; library calls are user-level functions built on top of system calls.**

---

# 📌 Processes & Threads

---

# ✅ 1) Process vs Thread

### ■ Explanation (simple, conceptual)

A **process** is an independent program in execution with its own memory.

A **thread** is the smallest unit of execution inside a process and shares the process's memory.

**Process = big container**

**Thread = lightweight unit inside the process**

---

### ■ When/Why used

## ✔ Process

- Heavy tasks
- Applications that must be isolated
- If one process crashes → others not affected

## ✔ Thread

- For parallelism inside the same application
- Web servers use threads to handle multiple requests
- Faster context switching

---

### ■ Example

- Opening Chrome = **Process**
- Each Chrome tab = **Thread**
- In .NET/Java: Each API request → thread from thread pool

---

### ■ Short summary line

**Process is independent with separate memory; threads are lightweight units inside a process sharing memory.**

---

# ✅ 2) PCB (Process Control Block) — What it contains?

### ■ Explanation (simple, conceptual)

PCB (Process Control Block) is a **data structure in OS** that stores **all information about a process**.

OS uses PCB to manage & switch processes.

---

### ■ When/Why used

- To track the state of every process
- Required for context switching
- Helps OS resume execution from exact point

---

### ■ PCB Contains:

1. **Process ID (PID)**
2. **Process state** (Ready/Running/Waiting...)
3. **Program counter** (next instruction address)
4. **CPU registers**
5. **Memory info** (base/limit, page tables)
6. **I/O status info**
7. **CPU scheduling info** (priority, time used)
8. **Open files list**

---

### ■ Example

When switching between VS Code and Chrome, OS saves each process's PCB so it can resume later without losing state.

---

### ■ Short summary line

**PCB stores all necessary information about a process so OS can manage and resume it.**

---

# ✅ 3) Context Switching — Why Needed?

### ■ Explanation (simple, conceptual)

Context switching is the process of **saving the state of currently running process** and **loading the state of another process**.

This allows multitasking.

---

### ⬛ When/Why used

- When CPU switches between processes/threads
- For multitasking OS
- To provide fair CPU time to all tasks
- During interrupts (I/O event, timer interrupt)

Without context switching, only one program could run at a time.

---

### ⬛ Example

- You type in Chrome → interrupt → OS switches to Slack notification → context switch → returns to Chrome
- In servers: OS switches between multiple API requests (threads)

---

### ⬛ Short summary line

**Context switching lets CPU save one process's state and load another, enabling multitasking.**

---

# ✅ 4) Process States → New, Ready, Running, Waiting, Terminated

### ⬛ Explanation (simple, conceptual)

A process moves through different states during its lifetime.

---

### ⬛ Process Lifecycle

## 🔵 1. New

Process is created but not ready for execution.

## 🔵 2. Ready

Process is in memory, waiting for CPU.

## 🔵 3. Running

CPU is executing the process instructions.

## 🔵 4. Waiting (Blocked)

Process is waiting for some I/O event:

(ex: disk read, network response, user input)

## 🔵 5. Terminated

Process finished or killed.

---

### ■ When/Why used

- OS uses states to schedule processes efficiently
- Helps decide which process to give CPU time
- Required for round-robin, priority scheduling

---

### ■ Example

When your VS Code loads a file:

- Running → Waiting for disk → Running again → Terminated (operation completed)

---

### ■ Short summary line

**A process moves through new → ready → running → waiting → terminated during its lifetime.**

---

---

# ✅ 5) Multithreading vs Multiprocessing

### ■ Explanation (simple, conceptual)

## 🔵 Multithreading

A single process contains **multiple threads** that share the same memory space.

## 🔵 Multiprocessing

Multiple **independent processes**, each with its own memory, run in parallel.

---

### ■ When/Why used

## ✔ Multithreading

- Lightweight

- Faster communication (shared memory)
- Best for I/O-bound tasks
- Common in servers (API requests handled via thread pool)

### ✔️ Multiprocessing

- Heavy but more stable
- Processes do NOT affect each other
- Best for CPU-heavy work (ML, video processing)
- Avoids thread-locking issues because each process has its own memory

---

### ■ Example

- Chrome tabs → Threads inside Chrome process (multithreading)
- Running Chrome + VS Code + Postman → Different processes (multiprocessing)
- .NET Kestrel handling multiple API requests → Multithreading

---

### ■ Short summary line

**Multithreading = many threads in one process; Multiprocessing = many independent processes running in parallel.**

---

# ✅ 6) Thread Synchronization — Why Required?

### ■ Explanation (simple, conceptual)

Thread synchronization ensures that **shared data** is accessed safely when multiple threads try to read/write at the same time.

Without synchronization → **race condition** occurs.

---

### ■ When/Why used

- When multiple threads access the same variable
- To prevent inconsistent data and corruption
- To enforce order of execution

- To avoid race conditions

### ■ Example (Race condition)

Two threads updating bank balance simultaneously:

Without lock:

```
Thread1: read balance = 100
Thread2: read balance = 100
Thread1: writes 80
Thread2: writes 90
Final balance = 90 (incorrect)
```

Using lock:

```
lock(_lockObj)
{
    balance -= amount;
}
```

### ■ Short summary line

**Thread synchronization prevents race conditions when multiple threads access shared data.**

# ✅ 7) What is a Deadlock? 4 Conditions for Deadlock

### ■ Explanation (simple, conceptual)

A **deadlock** occurs when two or more processes/threads are **waiting forever** for each other, and none can proceed.

### ■ When/Why occurs

Mainly due to poor locking design or resources requested in wrong order.

### ■ Four Conditions of Deadlock (VERY IMPORTANT)

## ✔ 1. Mutual Exclusion

Resource cannot be shared.

Example: Printer, database row lock.

### ✔️ 2. Hold and Wait

A thread holds one resource and waits for another.

### ✔️ 3. No Preemption

Resource cannot be forcibly taken away.

### ✔️ 4. Circular Wait

Thread A waits for Thread B → Thread B waits for Thread A.

---

#### ■ Example

```
Thread1 locks Resource A → waiting for Resource B
Thread2 locks Resource B → waiting for Resource A
```

Both wait forever → deadlock.

---

#### ■ Short summary line

**Deadlock happens when mutual exclusion, hold-and-wait, no preemption, and circular wait occur simultaneously.**

---

# ✅ 8) How Do You Prevent Deadlocks?

#### ■ Explanation (simple, conceptual)

Deadlocks can be avoided by breaking at least **one** of the four deadlock conditions.

---

#### ■ When/Why used

- In multithreaded systems
- Database systems
- Distributed systems
- API servers (thread pools)

---

#### ■ Deadlock Prevention Techniques

## ✔️ 1. **Lock Ordering** (most practical)

Acquire locks in **same global order**.

Example:

```
lock(resourceA)
{
    lock(resourceB)
    {
        // safe
    }
}
```

## ✔️ 2. Use Timeouts for Locks

If a lock is not acquired → give up.

```
Monitor.TryEnter(lockObj, 1000)
```

## ✔️ 3. Avoid Nested Locks

Reduce holding multiple locks.

## ✔️ 4. Release Resources Quickly

Keep critical sections small.

## ✔️ 5. Use Semaphores / Mutex properly

Prevent multiple threads from blocking each other.

## ✔️ 6. Use concurrency libraries

Like `ConcurrentDictionary`, `ConcurrentQueue`.

---

### ■ Example (Avoid circular wait)

If every thread acquires locks in alphabetical order (A → B → C), circular wait cannot happen.

---

### ■ Short summary line

**Avoid deadlocks by using lock ordering, timeouts, reducing nested locks, and careful resource management.**

---

## 📌 Memory Management

---

# ✅ 1) Paging vs Segmentation

■ **Explanation (simple, conceptual)**

### 🔵 Paging

- Memory is divided into **fixed-size blocks**
  - Physical memory → **frames**
  - Logical memory → **pages**
- Eliminates external fragmentation
- Simple memory management
- Every page = same size
- Page table is used for mapping

### 🔵 Segmentation

- Memory is divided into **variable-size segments** based on program structure

  Example: Code segment, Data segment, Stack segment
- Eliminates internal fragmentation
- More logical (resembles actual program structure)

---

■ **When/Why used**

### ✔ Paging

- When OS wants efficient & simple memory allocation
- Best for large systems
- Avoids external fragmentation

### ✔ Segmentation

- Used when a program has logical divisions
- Supports protection (each segment can have separate permissions)

---

### ■ Example

A program has:

- Code: 8 KB
- Data: 5 KB
- Stack: 2 KB

Segmentation → stored as 3 different segments

Paging → broken into pages like: page1, page2, page3… (all same size, e.g., 4KB)

---

### ■ Short summary line

**Paging = fixed-size blocks; Segmentation = variable-size logical divisions.**

---

# ✅ 2) Virtual Memory — Why Used?

### ■ Explanation (simple, conceptual)

Virtual memory allows a system to run programs **larger than physical RAM** by temporarily storing data on disk (swap space).

It gives an **illusion of large memory**.

---

### ■ When/Why used

- To allow multiple large applications to run simultaneously
- To avoid "Out of Memory" errors
- To isolate each process memory (security)
- To increase multitasking capability

---

### ■ Example

If your physical RAM = 8GB

But applications need 12GB

OS moves less-used pages to disk → continues running normally.

---

■ **Short summary line**

**Virtual memory allows programs to run even when RAM is insufficient by using disk space as memory.**

---

# ✅ 3) Thrashing — What and Why?

■ **Explanation (simple, conceptual)**

Thrashing occurs when the OS spends **more time swapping pages** between RAM and disk than executing the actual program.

System becomes extremely slow.

---

■ **When/Why it happens**

- Too many processes running
- Not enough RAM
- High page-fault rate
- Wrong paging algorithms

---

■ **Example**

If every few milliseconds OS swaps pages in/out → CPU becomes idle waiting for memory → system freezes.

Typical Windows example:

RAM full → Hard disk usage at 100% → system very slow.

---

■ **Short summary line**

**Thrashing is excessive paging that slows down the system when RAM is overloaded.**

---

# ✅ 4) Page Replacement Algorithms — FIFO, LRU

■ **Explanation (simple, conceptual)**

When a new page must be loaded but memory is full, OS chooses a page to remove using replacement algorithms.

---

### 🔵 1. FIFO (First In First Out)

Evicts the **oldest loaded page**.

■ **When/Why used**

- Simple and easy
- But may remove frequently used pages

■ **Example**

Pages loaded in order: A, B, C

Next page = D → remove A (oldest)

---

### 🔵 2. LRU (Least Recently Used)

Evicts the page that has not been used for the **longest time**.

■ **When/Why used**

- More accurate than FIFO
- Prevents removing frequently used pages
- More costly (need timestamps)

■ **Example**

Recent usage: C (just used), B, A

Least recently used = A → remove A.

---

■ **Short summary line**

**FIFO removes earliest page; LRU removes least recently used page for better performance.**

---

## ✅ 5) Stack vs Heap Memory

■ **Explanation (simple, conceptual)**

### 🔵 Stack

- Stores local variables, function calls, parameters
- Memory managed automatically (LIFO)
- Very fast

- Limited size
- Allocated at compile/run time

## 🟣 Heap

- Stores objects, dynamic memory (`new` in .NET/Java)
- Memory managed manually (GC)
- Slower than stack
- Large memory area
- Allocated at runtime

---

### ⬛ When/Why used

## ✔ Stack

- For small, short-lived data
- Function calls, parameters, return addresses

## ✔ Heap

- For large objects
- Objects whose lifetime is unknown
- Shared objects

---

### ⬛ Example (C# / Java)

```
int x = 10;   // stack
Person p = new Person(); // heap
```

---

### ⬛ Short summary line

**Stack holds local variables & call frames; heap holds objects & dynamic memory.**

---

## 📌 Scheduling Algorithms

---

# ✅ 1) FCFS, SJF, Round Robin — Differences

**■ Explanation (simple, conceptual)**

These are CPU scheduling algorithms used by OS to decide **which process gets CPU next**.

---

## 🔵 FCFS (First Come First Serve)

- Processes served in **arrival order**
- Simple and non-preemptive
- No starvation
- Long processes delay short ones ("convoy effect")

**When/Why used:**

- Simple batch systems
- When fairness (order) matters
- Low overhead

---

## 🔵 SJF (Shortest Job First)

- Process with **shortest execution time** is chosen
- Minimizes average waiting time
- Can cause starvation (short jobs always picked)

**When/Why used:**

- CPU-bound processes
- When job lengths are predictable
- Improves throughput

---

## 🔵 Round Robin (RR)

- Each process gets fixed time slice (quantum)
- Preemptive

- Best for time-sharing & interactive systems

**When/Why used:**

- Operating systems for users
- Ensures fairness
- No starvation

---

### ■ Example

Processes (burst time): P1=10, P2=4, P3=2

| Algorithm | First Process | Behavior |
|---|---|---|
| FCFS | P1 | Long wait for others |
| SJF | P3 | Shortest first |
| RR (quantum=2) | Time sliced | Fair turn-taking |

---

### ■ Short summary line

**FCFS = arrival order, SJF = shortest first, RR = equal time slices (fair for all).**

---

# ✅ 2) Priority Scheduling + Starvation Problem

### ■ Explanation (simple, conceptual)

Priority scheduling selects the process with the **highest priority** (or lowest number) to run first.

---

### ■ When/Why used

- Used when some processes must run earlier (system, real-time tasks)
- Useful for time-critical workloads
- Allows OS to schedule important tasks before regular ones

---

## 🔵 Starvation Problem

Occurs when **low-priority processes wait indefinitely** because higher-priority processes keep arriving.

### Example

- P1 (High priority)
- P2 (High priority)
- P3 (High priority)
- P4 (Low priority)

P4 may **never** get CPU → starvation.

---

## 🔵 Solution → Aging Technique

Gradually increase the priority of waiting processes.

Example:

Every 2 seconds in ready queue → increase priority by 1.

Eventually low-priority processes get executed.

---

#### ⬛ Short summary line

**Priority scheduling picks highest priority; starvation happens when low-priority tasks never run—solved by aging.**

---

# ✅ 3) Pre-emptive vs Non-preemptive Scheduling

#### ⬛ Explanation (simple, conceptual)

### 🔵 Pre-emptive Scheduling

OS can **interrupt** a running process and give CPU to another process with:

- Higher priority
- Time slice expiration
- System event

### 🔵 Non-preemptive Scheduling

Once a process gets CPU, it **runs until completion** or voluntarily releases CPU.

---

#### ⬛ When/Why used

### ✔️ Pre-emptive

- Used in modern OS (Windows, Linux)
- Best for real-time and interactive systems
- Ensures responsiveness
- But more overhead due to context switching

### ✔️ Non-preemptive

- Simple, predictable
- Less overhead
- Used in embedded systems
- Risk of long waiting time

---

**■ Example**

| Scenario | Result |
| --- | --- |
| Running process P1, higher priority P2 arrives | Pre-emptive → switch to P2 |
| Running process P1, even if P2 arrives | Non-preemptive → P2 must wait |

---

**■ Short summary line**

**Pre-emptive scheduling interrupts running processes; non-preemptive lets a process run until it finishes or yields.**

---

## 📌 File System & OS Internals

---

# ✅ 1) Inode Structure

**■ Explanation (simple, conceptual)**

An **inode** (index node) is a data structure used in Unix/Linux file systems to **store metadata of a file**.

It does *not* store the file name or content — only metadata.

---

**■ When/Why is it used**

- To keep track of file properties

- To locate the actual data blocks of a file on disk
- Faster file access and directory traversal
- Enables linking (hard links use same inode)

---

### ⬛ Inode Contains:

1. File size
2. File type (directory, regular file)
3. Ownership (UID, GID)
4. Permissions (rwx flags)
5. Timestamps (created, modified, accessed)
6. Number of links
7. File location pointers:
    - Direct pointers
    - Indirect pointers
    - Double indirect pointers
    - Triple indirect pointers

---

### ⬛ Example

When you run `ls -i` in Linux, you see inode numbers.

The OS uses inode pointer metadata to locate file data blocks.

---

### ⬛ Short summary line

**Inode stores file metadata and pointers to data blocks, not file content or name.**

---

# ✅ 2) What is a Semaphore?

### ⬛ Explanation (simple, conceptual)

A **semaphore** is a synchronization mechanism used to control access to shared resources in concurrent systems.

It uses an integer value to allow or block threads.

---

■ **When/Why used**

- To limit how many threads access a critical section
- To avoid race conditions
- To coordinate tasks in multithreading
- Works well when multiple threads can access a resource (not just one)

■ **Types of Semaphore:**

1. **Binary Semaphore**
    - Value = 0 or 1
    - Works like a mutex
2. **Counting Semaphore**
    - Value ≥ 0
    - Allows multiple threads at once (e.g., max 3 threads)

■ **Example**

```
sem_wait(&s);   // lock
// critical section
sem_post(&s);   // unlock
```

■ **Short summary line**

**Semaphore is a signaling mechanism used to control access to shared resources using a counter.**

# ✅ 3) Mutex vs Semaphore

■ **Explanation (simple, conceptual)**

| Feature | Mutex | Semaphore |
| --- | --- | --- |
| Access | Only 1 thread | Multiple threads allowed |
| Type | Locking mechanism | Signaling mechanism |
| Ownership | Owned by a thread | Not owned |
| Value | Always 1 | Can be >1 |

| Feature | Mutex | Semaphore |
|---|---|---|
| Use case | Critical section with single access | Limited resource pool |

---

### ■ When/Why used

## ✔ Use **Mutex** when:

- Only **one** thread can access a resource at a time
- Resource is exclusive (DB write, file write)

## ✔ Use **Semaphore** when:

- A resource allows **multiple** threads
- Example: Thread pool, connection pool

---

### ■ Example

A shared printer → only ONE process prints → **mutex**

DB connection pool with 10 connections → **counting semaphore with value 10**

---

### ■ Short summary line

**Mutex allows only one thread; semaphore allows many threads based on counter value.**

---

# ✅ 4) Race Condition — Example

### ■ Explanation (simple, conceptual)

A race condition occurs when **two or more threads modify shared data simultaneously**, leading to inconsistent results.

---

### ■ When/Why happens

- No proper locking or synchronization
- Shared variables accessed concurrently
- Multi-threaded applications, servers, OS scheduling

---

### ■ Example

Bank account withdrawal — two threads access same balance:

Balance = 100

Thread1 withdraw(40)

Thread2 withdraw(30)

Both read 100 →

Thread1 writes 60

Thread2 writes 70

Final balance becomes **70** or **60** → incorrect.

With lock:

```
lock(lockObj)
{
    balance -= amount;
}
```

---

■ **Short summary line**

**Race condition is inconsistent data caused by unsynchronized access to shared variables.**

---

# ✅ 5) What Happens When You Open an Application in OS?

■ **Explanation (simple, conceptual)**

When you open an app (e.g., VS Code, Chrome), the OS performs multiple steps to load and execute it.

---

■ **When/Why is it used**

This flow explains **process creation**, **loading**, **scheduling**, and **execution**.

---

■ **Detailed Steps (Interview-Ready)**

### 1. User launches the program

Clicking icon or running from terminal triggers OS launcher.

---

### 2. OS creates a new process

- Assigns **Process ID (PID)**
- Allocates PCB (Process Control Block)

---

## 3. Program binary loaded into memory

- Loads **code**, **data**, **stack**, **heap**
- Allocates memory using paging mechanism

---

## 4. OS loads required libraries

- DLLs in Windows
- .so shared objects in Linux
- Loads runtime (CLR for .NET, JVM for Java)

---

## 5. Scheduler assigns CPU

Process goes to **ready queue**, then to **running** state.

---

## 6. OS sets up file handles & I/O

- Opens necessary files
- Initializes UI/graphics
- Loads configuration

---

## 7. Code execution begins

Instruction by instruction, OS manages:

- CPU scheduling
- Memory allocation
- I/O interrupts

---

## 8. App runs until user closes it

OS cleans resources, closes files, destructs process.

---

■ **Short summary line**

**Opening an application involves process creation, loading program into memory, linking libraries, CPU scheduling, and execution.**

---

# 🔥 COMPUTER NETWORKS (CN)

## 📌 Basics & Networking Models

---

# ✅ 1) What is a Computer Network?

### ⬛ Explanation (simple, conceptual)

A Computer Network is a system where **multiple devices (computers, servers, mobiles, routers)** are connected to each other to **share data, resources, and services**.

It enables communication over wired or wireless mediums.

### ⬛ When/Why is it used

- To share files, applications, printers
- To communicate (email, chat, video calls)
- To access internet and cloud services
- For distributed systems and APIs

### ⬛ Example

- Your office LAN
- Internet
- Mobile hotspot network
- Cloud-based systems communicating via REST APIs

### ⬛ Short summary line

**A computer network connects devices to communicate and share data/resources.**

---

# ✅ 2) OSI Model — 7 Layers & Functions

### ⬛ Explanation (simple, conceptual)

OSI (Open Systems Interconnection) model is a **7-layer framework** used to understand how data travels in a network.

Each layer has a specific responsibility.

---

■ **7 Layers (Top → Bottom)**

## 7. Application Layer

- User-facing protocols (HTTP, SMTP, FTP)

## 6. Presentation Layer

- Data formatting (encryption, compression)

## 5. Session Layer

- Establish/maintain/terminate communication sessions

## 4. Transport Layer

- End-to-end delivery
- TCP/UDP
- Segmentation, error control

## 3. Network Layer

- Logical addressing (IP)
- Routing packets through routers

## 2. Data Link Layer

- MAC addressing
- Frames
- Error detection

## 1. Physical Layer

- Electrical signals
- Cables, NIC, radio waves

---

■ **When/Why is it used?**

- Helps standardize communication
- Helps troubleshoot network issues
- Explains routing, encryption, sessions

■ **Example**

Sending an email:

Application (SMTP) → Transport (TCP) → Internet (IP) → Data link → Physical.

■ **Short summary line**

**OSI model explains how data flows through 7 layers from physical hardware to user applications.**

---

# ✅ 3) TCP/IP Model — Layer Comparison with OSI

■ **Explanation (simple, conceptual)**

TCP/IP model is the real-world model used on the internet.

It has **4 layers**, not 7.

---

■ **TCP/IP Layers vs OSI Layers**

| TCP/IP Layer | OSI Equivalent | Function |
|---|---|---|
| **Application** | Application + Presentation + Session | Protocols like HTTP, DNS |
| **Transport** | Transport | TCP/UDP, port numbers |
| **Internet** | Network | IP addressing, routing |
| **Network Access** | Data Link + Physical | MAC, frames, bits |

---

■ **When/Why used**

- Used by all internet devices
- Defines how packets travel from source → destination
- Forms basis of network communication in modern systems

■ **Example**

When you send an HTTP request:

Browser → TCP → IP → Ethernet → Internet → Server

⬛ **Short summary line**

**TCP/IP has 4 layers and is the practical version of OSI used in real internet communication.**

---

# ✅ 4) What is Encapsulation & Decapsulation?

⬛ **Explanation (simple, conceptual)**

## 🔵 Encapsulation (Sender Side)

Adding headers (and sometimes trailers) to data as it moves **down the layers**.

Example: Application data → TCP header → IP header → Ethernet header

## 🔵 Decapsulation (Receiver Side)

Removing headers as data moves **up the layers**.

---

⬛ **When/Why used**

- Required to send data across networks
- Each layer adds its own control information
- Ensures correct delivery, routing, error control

---

⬛ **Example**

```
You send: "GET /index.html"
Encapsulation:
HTTP → TCP → IP → MAC → Bits → Sent via network

Receiver:
Bits → MAC → IP → TCP → HTTP → Decapsulation
```

---

⬛ **Short summary line**

**Encapsulation adds protocol headers; decapsulation removes them during data transmission.**

---

# ✅ 5) What are Protocols? Examples.

**⬛ Explanation (simple, conceptual)**

A protocol is a set of **rules and formats** that defines how devices communicate in a network.

It ensures that sender and receiver understand each other.

---

**⬛ When/Why used**

- To standardize communication
- To ensure reliability (TCP), addressing (IP), security (HTTPS), etc.
- Without protocols, communication would be impossible

---

**⬛ Examples**

### Application Layer Protocols

- HTTP / HTTPS → Web browsing
- FTP → File transfer
- SMTP / IMAP → Email
- DNS → Domain name lookup

### Transport Layer

- TCP → Reliable communication
- UDP → Fast, no confirmation

### Network Layer

- IP → Addressing and routing
- ICMP → Ping
- ARP → Resolve IP → MAC

---

**⬛ Short summary line**

**Protocols are rules for communication—like HTTP, TCP, IP, DNS—that allow devices to exchange data reliably.**

---

## 📌 IP, Routing & Switching

# ✅ 1) IPv4 vs IPv6

### ⬛ Explanation (simple, conceptual)

IPv4 and IPv6 are versions of Internet Protocol used for addressing devices.

## 🔵 IPv4

- 32-bit address
- ~4.3 billion unique addresses
- Written as: `192.168.1.10`
- Supports NAT due to limited addresses

## 🟣 IPv6

- 128-bit address
- Almost unlimited addresses
- Written as: `2001:0db8:85a3::8a2e:0370:7334`
- Built-in security (IPSec)

---

### ⬛ When/Why used

- IPv4 still widely used (legacy systems)
- IPv6 used to avoid address exhaustion and support more devices (IoT)

---

### ⬛ Example

- Laptop using IPv4 in home network
- Cloud servers / modern devices using IPv6

---

### ⬛ Short summary line

**IPv4 is 32-bit and limited; IPv6 is 128-bit and provides huge address space with better security.**

---

# ✅ 2) Static IP vs Dynamic IP

### ⬛ Explanation (simple, conceptual)

### 🔵 Static IP

- Assigned manually
- Never changes
- Used for servers (DNS, email, APIs)

### 🟣 Dynamic IP

- Assigned automatically by DHCP
- Changes when device reconnects
- Used for normal users (home networks)

---

#### ■ When/Why used

- Static → when permanent connectivity needed
- Dynamic → reduces management overhead

---

#### ■ Example

Your mobile → Dynamic IP

Your Azure VM or API endpoint → Static IP

---

#### ■ Short summary line

**Static IP stays fixed; dynamic IP changes and is assigned by DHCP.**

---

# ✅ 3) What is Subnetting? Default Subnet Sizes

#### ■ Explanation (simple, conceptual)

Subnetting divides a large network into **smaller logical networks**.

It improves IP management, security, and reduces congestion.

---

#### ■ When/Why used

- To isolate departments (HR, IT, Finance)
- To improve routing efficiency
- To avoid broadcast storms

- To manage IP pools better

### ⬛ Default Subnet Sizes / Classes

| Class | Range | Default Subnet Mask | Hosts |
|---|---|---|---|
| **A** | 0–127 | 255.0.0.0 | ~16 million |
| **B** | 128–191 | 255.255.0.0 | ~65,000 |
| **C** | 192–223 | 255.255.255.0 | 254 |

### ⬛ Example

A company divides a `192.168.1.0/24` network into:

- `192.168.1.0/26`
- `192.168.1.64/26`
- `192.168.1.128/26`

Each subnet gets 62 hosts.

### ⬛ Short summary line

**Subnetting divides a network into smaller parts to improve management and security.**

# ✅ 4) Public IP vs Private IP

### ⬛ Explanation (simple, conceptual)

## 🔵 Public IP

- Accessible over the internet
- Unique globally
- Assigned by ISP / Cloud providers

## 🟣 Private IP

- Used inside LAN
- Not reachable from internet
- Requires NAT to communicate outside

### ■ When/Why used

- Public IP → servers, websites, cloud VMs
- Private IP → office/home devices

---

### ■ Examples

## ✔ Private IP Ranges:

- 10.0.0.0 – 10.255.255.255
- 172.16.0.0 – 172.31.255.255
- 192.168.0.0 – 192.168.255.255

## ✔ Public IP example:

52.12.145.11 (AWS VM)

---

### ■ Short summary line

**Public IP is globally accessible; private IP is used inside local networks.**

---

# ✅ 5) Router vs Switch vs Hub

### ■ Explanation (simple, conceptual)

## 🔵 Hub

- Broadcasts data to all devices
- No intelligence
- Works on Physical layer
- Very outdated

## 🟣 Switch

- Forwards data using MAC addresses
- Works on Data Link layer
- Faster, secure
- Most commonly used in LANs

## 🔴 Router

- Routes data between networks
- Uses IP addresses
- Connects LAN to internet
- Performs NAT, firewall, DHCP

---

### ⬛ When/Why used

- Hub → small/simple networks (rare today)
- Switch → connect devices inside same network
- Router → connect different networks (LAN ↔ Internet)

---

### ⬛ Example

- Home Wi-Fi router contains router + switch + DHCP
- Office networks use switches everywhere

---

### ⬛ Short summary line

**Hub broadcasts, switch filters using MAC, router routes using IP across networks.**

---

# ✅ 6) ARP — What and Why?

### ⬛ Explanation (simple, conceptual)

ARP (Address Resolution Protocol) maps an **IP address → MAC address** within a local network.

Network needs MAC to deliver frames inside LAN.

---

### ⬛ When/Why used

- Required when a device knows target IP but not MAC
- Essential for LAN communication
- Every time a packet enters LAN, ARP resolves it

---

### ⬛ Example

Your PC wants to send to 192.168.1.5 →

It broadcasts ARP request:

**"Who has 192.168.1.5? Send me your MAC."**

Target replies with its MAC.

---

■ **Short summary line**

**ARP resolves IP addresses to MAC addresses inside a local network.**

---

# ✅ 7) DHCP Working — Step by Step

■ **Explanation (simple, conceptual)**

DHCP (Dynamic Host Configuration Protocol) automatically assigns IP, gateway, DNS to devices.

---

■ **When/Why used**

- Avoid manual IP configuration
- For scalable networks (office, ISP networks)
- Ensures unique IP assignment
- Supports IP reuse

---

■ **DHCP 4-Step Process (DORA)**

### 1. Discover

Client → Broadcast request:

"Is there any DHCP server?"

### 2. Offer

Server → Offers an IP address.

### 3. Request

Client → Requests the offered IP.

### 4. Acknowledge

Server → Confirms assignment and sends:

- IP

- Subnet mask
- Gateway
- DNS

---

### ⬛ Example

Your phone connecting to Wi-Fi → automatically gets:

- 192.168.1.x
- gateway 192.168.1.1
- DNS 8.8.8.8

---

### ⬛ Short summary line

**DHCP assigns IP automatically using DORA: Discover → Offer → Request → Acknowledge.**

---

## 📌 TCP/UDP (Must Know)

---

# ✅ 1) TCP vs UDP Differences

### ⬛ Explanation (simple, conceptual)

TCP and UDP are **Transport Layer protocols** that handle data delivery between systems.

---

## 🔵 TCP — Transmission Control Protocol

- Connection-oriented
- Reliable delivery (acknowledgements, retransmissions)
- Ensures ordered data delivery
- Slower but accurate
- Heavyweight (more overhead)

## 🟣 UDP — User Datagram Protocol

- Connectionless
- No reliability, no acknowledgements

- No ordering guarantee
- Faster and lightweight
- Best for real-time apps

---

### ⬛ When/Why used

## ✔️ TCP

- When accuracy > speed
- For financial, banking, login, file transfer

## ✔️ UDP

- When speed > accuracy
- Dropping some packets is acceptable
- For real-time systems

---

### ⬛ Example Table

| Feature | TCP | UDP |
| --- | --- | --- |
| Reliability | Yes | No |
| Connection | Required | Not needed |
| Speed | Slower | Faster |
| Use cases | Web, Email | Video calls, Gaming |

---

### ⬛ Short summary line

**TCP is reliable and connection-based; UDP is fast and connectionless.**

---

# ✅ 2) Why TCP is Reliable? Explain 3-Way Handshake (SYN–SYN/ACK–ACK)

### ⬛ Explanation (simple, conceptual)

TCP provides **reliable, ordered, and error-checked** data delivery.

It uses acknowledgements, retransmissions, and sequencing.

---

### ⬛ Why TCP is reliable?

1. **3-way handshake** (connection setup)
2. **Sequence numbers**
3. **Acknowledgements (ACKs)**
4. **Retransmission of lost packets**
5. **Flow control (Window size)**
6. **Congestion control (Slow start)**
7. **Error detection**

---

**■ 3-Way Handshake (Connection Establishment)**

### Step 1: SYN (synchronize)

Client → Server: "I want to connect, here's my sequence number."

### Step 2: SYN + ACK

Server → Client: "OK, I acknowledge; here's my sequence."

### Step 3: ACK

Client → Server: "I acknowledge your sequence."

After this → connection is established.

---

**■ Short summary line**

**TCP is reliable due to handshake, sequencing, ACKs, retransmission, flow & congestion control.**

---

# ✅ 3) What is Flow Control & Congestion Control in TCP?

**■ Explanation (simple, conceptual)**

### 🔵 Flow Control

Controls **sender → receiver** speed.

Prevents sender from overwhelming the receiver.

TCP uses **Sliding Window Protocol** for this.

---

## 🔵 Congestion Control

Controls traffic in the **network** to avoid congestion (overloaded routers).

TCP uses:

- Slow Start
- Additive Increase
- Multiplicative Decrease
- Congestion Window (cwnd)

---

#### ⬛ When/Why used

### ✔ Flow Control

- Needed when sender sends data faster than receiver can process
- Protects slow devices

### ✔ Congestion Control

- Needed to avoid packet loss
- Ensures network stability
- Prevents router overload

---

#### ⬛ Example

### Flow Control example:

Receiver says:

"My window size = 4 packets"

Sender sends only 4 packets at a time.

---

### Congestion Control example:

If packet loss occurs →

TCP reduces sending rate by half (Multiplicative Decrease).

---

#### ⬛ Short summary line

**Flow control protects the receiver; congestion control protects the network.**

---

# ✅ 4) Use Cases — When to Use TCP vs UDP

### ■ Explanation (simple, conceptual)

Use TCP when data must be correct.

Use UDP when speed matters more than accuracy.

---

### ■ When/Why used

## ✔ Use **TCP** for:

- Web requests (HTTP/HTTPS)
- Banking transactions
- File uploads/downloads
- Emails
- Database connections
- REST APIs
- Login/authentication systems

## ✔ Use **UDP** for:

- Video calls (Zoom, Teams)
- Live streaming
- Online gaming
- DNS queries
- IoT real-time sensors
- Voice over IP (VoIP)

---

### ■ Example

In a video call, dropping a few frames is OK → UDP

In an online payment API, accuracy is critical → TCP

---

### ■ Short summary line

**TCP is used for accurate delivery; UDP is used for fast, real-time communication.**

---

## 📌 HTTP/HTTPS + Web Communication

---

# ✅ 1) HTTP vs HTTPS — Difference

### ■ Explanation (simple, conceptual)

**HTTP** is an insecure protocol used to transfer data between client and server.

**HTTPS** is secure HTTP that uses **SSL/TLS encryption** to protect data during transmission.

---

### ■ When/Why is it used

## ✔️ HTTP

- Used when data is non-sensitive
- Faster but not secure

## ✔️ HTTPS

- Encrypts data
- Protects from hacking (MITM attacks)
- Mandatory for login pages, payments, banking
- Used almost everywhere today

---

### ■ Example

```
http://example.com       // Data in plain text
https://example.com      // Data encrypted with TLS
```

---

### ■ Short summary line

**HTTPS = HTTP + SSL/TLS encryption, providing secure communication.**

---

# ✅ 2) What is SSL/TLS Handshake?

■ **Explanation (simple, conceptual)**

SSL/TLS handshake is the process by which **browser and server establish a secure, encrypted connection**.

---

■ **When/Why is it used**

- To exchange keys securely
- To verify server identity
- To start encrypted communication

---

■ **Steps (Interview-Ready)**

### 1. Client Hello

Client sends supported encryption algorithms.

### 2. Server Hello

Server responds with chosen algorithm + **server certificate**.

### 3. Certificate Verification

Client verifies certificate authenticity.

### 4. Key Exchange

Client generates a **session key** and encrypts it with server's public key.

### 5. Secure Connection Established

Both sides now share same secret session key.

All further communication is **encrypted**.

---

■ **Short summary line**

**SSL/TLS handshake securely exchanges keys between client and server to begin encrypted communication.**

---

# ✅ 3) What is a Cookie? What is a Session?

■ **Explanation (simple, conceptual)**

### 🔵 Cookie

- Small piece of data stored on **client/browser**
- Used for remembering users, preferences, tracking

### 🟣 Session

- Data stored on **server**
- Identifies logged-in users
- More secure than cookies

---

#### ⬛ When/Why used

- Authentication
- Shopping cart
- Remembering user preferences
- Tracking user activity

---

#### ⬛ Example

### Cookie example:

```
isLoggedIn = true
theme = dark
```

### Session example (server memory):

```
sessionId → userId 101
```

---

#### ⬛ Short summary line

**Cookies store data in browser; sessions store data on server for user identification.**

---

# ✅ 4) What is JWT Token? Where Stored?

### ⬛ Explanation (simple, conceptual)

JWT (JSON Web Token) is a compact, signed token used for **stateless authentication**.

It contains:

- Header
- Payload (user info, roles)
- Signature

Server does NOT store session data; token itself contains trustable info.

---

### ⬛ When/Why used

- Modern REST APIs
- Single Page Applications (Angular, React)
- Mobile apps
- Microservices authentication
- Stateless systems

---

### ⬛ Where is JWT Stored?

- **LocalStorage** (most common)
- **SessionStorage**
- **HTTP-only cookies** (more secure)

---

### ⬛ Example (JWT Structure)

```
xxxxx.yyyyy.zzzzz
```

---

### ⬛ Short summary line

**JWT is a stateless authentication token usually stored in browser storage or secure cookies.**

---

# ✅ 5) CORS — What & Why?

### ⬛ Explanation (simple, conceptual)

CORS (Cross-Origin Resource Sharing) is a browser security mechanism that blocks requests from **different domains** unless the server explicitly allows them.

Browser only allows cross-domain API calls if the server replies with **Access-Control-Allow-Origin** header.

---

### ■ When/Why used

- Angular/React frontend calling .NET backend
- Prevents unauthorized websites from calling APIs
- Avoids cross-site attacks

---

### ■ Example

Frontend:

```
http://localhost:4200
```

Backend API:

```
http://localhost:5000
```

Browser blocks request → unless API sends:

```
Access-Control-Allow-Origin: http://localhost:4200
```

---

### ■ Short summary line

**CORS protects APIs by controlling which domains can access them.**

---

# ✅ 6) DNS — How Domain Resolves to IP?

### ■ Explanation (simple, conceptual)

DNS (Domain Name System) converts **domain names to IP addresses** so browsers can find servers.

---

### ■ When/Why used

- To avoid remembering numeric IPs (e.g., 142.250.182.46)

- To allow friendly URLs
- Essential for internet communication

---

■ **Steps of DNS Resolution (Interview-Ready)**

### 1. Browser Cache

Check if IP already stored.

### 2. OS Cache

OS checks local DNS cache.

### 3. Router Cache

Home router stores recently resolved domains.

### 4. ISP DNS Server

If not found above, ISP DNS resolves it.

### 5. Root DNS Server

Points to TLD servers (.com, .org, .net).

### 6. TLD DNS Server

Points to authoritative DNS of domain.

### 7. Authoritative DNS Server

Returns the real IP (e.g., `142.250.182.46`).

Browser then connects to that IP via TCP/HTTPS.

---

■ **Short summary line**

**DNS converts domain names into IP addresses using a hierarchical lookup from cache → ISP → root → authoritative servers.**

---

## 📌 Network Security

---

# ✅ 1) Firewall — Purpose?

■ **Explanation (simple, conceptual)**

A **firewall** is a network security device/software that **filters incoming and outgoing traffic** based on defined rules.

It allows safe traffic and blocks malicious/unauthorized traffic.

### ■ When/Why is it used

- To protect servers from external attacks
- To block suspicious IPs/ports
- To restrict access to internal systems
- To enforce network security policies
- Used at organization gateways, cloud VMs, APIs

### ■ Example

A firewall can block:

- Port 23 (Telnet, insecure)
- Unknown IP trying to hit API
- Unauthorized traffic from public internet

Azure Example: *NSG (Network Security Group) acts as a firewall for VNets.*

### ■ Short summary line

**Firewall filters traffic to protect networks by allowing only trusted communication.**

---

# ✅ 2) Proxy vs VPN

### ■ Explanation (simple, conceptual)

## 🔵 Proxy

Acts as an **intermediate server** between client and internet.

Hides client identity and controls internet access.

## 🟣 VPN

Creates an **encrypted tunnel** between user and network, hiding data and IP.

---

### ■ When/Why used

### ✔️ Proxy is used for:

- Content filtering (block sites)
- Caching pages
- Hiding client identity
- Monitoring employee internet usage

### ✔️ VPN is used for:

- Securely accessing office network remotely
- Encrypting all internet traffic
- Hiding real IP and location

---

**■ Example**

- Office uses a proxy to block Facebook.
- Employee uses VPN to connect to office server securely from home.

**■ Short summary line**

**Proxy controls and filters traffic; VPN encrypts and hides all traffic for secure remote access.**

---

# ✅ 3) Man-in-the-Middle (MITM) Attack

**■ Explanation (simple, conceptual)**

MITM attack occurs when an attacker secretly intercepts and possibly modifies communication between two parties **without their knowledge**.

**■ When/Why it happens**

- Using public Wi-Fi
- Weak/no HTTPS
- Fake access points
- ARP spoofing
- Unsecured networks

**■ Example**

You send login credentials to a website using HTTP.

Attacker in between captures username/password.

HTTPS prevents MITM because data is encrypted end-to-end.

⬛ **Short summary line**

**MITM attack intercepts communication between client and server to steal or modify data.**

---

# ✅ 4) CSRF vs XSS Attack

⬛ **Explanation (simple, conceptual)**

## 🔵 CSRF (Cross-Site Request Forgery)

Tricks a **logged-in user** into performing unwanted actions on a website.

- Attacker forces your browser to send a request using your **session cookies**
- Targets **state-changing actions**

### Example

While logged into bank site, you click malicious link → money transferred without your intent.

---

## 🟣 XSS (Cross-Site Scripting)

Attacker injects **malicious JavaScript** into a website that runs on users' browsers.

- Steals cookies
- Redirects, phishing
- Defaces UI

### Example

Comment box allows script:

```
<script>alert('Hacked')</script>
```

---

■ **When/Why used**

### CSRF:

- Attacker uses victim's **authentication**
- Requires victim to be logged in

### XSS:

- Attacker runs **script inside victim browser**
- No login required

---

■ **Short summary line**

**CSRF forces a user to perform unwanted actions; XSS injects malicious scripts into a website.**

---

# ✅ 5) IDS vs IPS

■ **Explanation (simple, conceptual)**

## 🔵 IDS (Intrusion Detection System)

Monitors traffic, detects malicious activity, and **alerts**.

## 🟣 IPS (Intrusion Prevention System)

Monitors traffic and **blocks** malicious activity in real time.

---

■ **When/Why used**

### IDS is used when:

- Organization wants monitoring
- Security team wants visibility
- No automatic blocking needed

### IPS is used when:

- Immediate prevention required
- Protecting critical servers

- Automated security policy enforcement

---

■ **Example**

### IDS:

Detects unusual traffic pattern → sends alert

*"Possible SQL Injection attack detected."*

### IPS:

Detects pattern → **blocks request immediately**

---

■ **Short summary line**

**IDS detects and alerts; IPS detects and blocks threats automatically.**

---

## 📌 Real-time Networking Questions

---

# ✅ 1) What happens when you type a URL in the browser?

■ **Explanation (simple, conceptual)**

Typing a URL triggers a full chain of browser → OS → network → server interactions to fetch the webpage.

■ **When/Why is it used**

This explains the entire request lifecycle and is a COMMON interview question that checks networking + DNS + HTTP basics.

---

■ **Step-by-step Process (Interview-Perfect)**

### 1. URL Parsing

Browser analyzes the URL:

- Protocol (HTTP/HTTPS)
- Domain (google.com)
- Path (/search)

---

## 2. Browser Cache Check

Browser checks if it already knows IP from DNS cache.

---

## 3. DNS Lookup

If not in cache → OS → Router → ISP → Root DNS → Authoritative DNS

Returns server IP (e.g., `142.250.182.46`)

---

## 4. TCP Handshake

For HTTPS:

- SYN → SYN/ACK → ACK

  Connection established.

---

## 5. TLS/SSL Handshake (if HTTPS)

- Certificate exchange
- Key exchange
- Secure encrypted tunnel created

---

## 6. HTTP Request Sent

Browser sends: GET /index.html

Headers include cookies, tokens, user-agent.

---

## 7. Server Processes Request

Server generates response using:

- Controllers
- Services
- Database calls
- Caching layer
- Load balancer routing

---

## 8. HTTP Response Returned

Server returns HTML/JSON + status code.

---

### 9. Browser Renders

- HTML parsed
- CSS applied
- JS executed
- Images loaded

---

■ **Short summary line**

**Typing a URL triggers DNS lookup → TCP/TLS handshake → HTTP request → server response → browser rendering.**

---

# ✅ 2) Why latency increases in remote API requests?

■ **Explanation (simple, conceptual)**

Latency = **time delay** between sending request and receiving response.

It increases because the request must travel longer distances and pass through multiple network layers.

---

■ **When/Why latency increases**

### ✔ 1. **Physical distance**

Longer distance → more hops → more delay.

Example: Client in India calling API hosted in US.

### ✔ 2. **Network congestion**

High traffic on routers → queue delays.

### ✔ 3. **Slow server or overloaded backend**

Server takes longer to process the request.

### ✔ 4. **DNS lookup time**

Slow DNS increases initial request delay.

### ✔️ 5. **TLS/SSL handshake**

HTTPS handshake adds extra round-trips.

### ✔️ 6. **Serialization/Deserialization overhead**

JSON/XML encoding/decoding takes time.

### ✔️ 7. **Multiple microservices calls**

Internal network calls add latency.

---

#### ■ Example

Your Angular app calls .NET API deployed in a different region → server takes time → network hops → increased latency.

---

#### ■ Short summary line

**Latency increases due to distance, network congestion, slow servers, TLS handshakes, and multiple internal hops.**

---

# ✅ 3) How load balancers help in scaling?

#### ■ Explanation (simple, conceptual)

Load balancer distributes incoming traffic across multiple servers so no single server gets overloaded.

#### ■ When/Why used

- To prevent server overload
- To achieve high availability
- To scale horizontally (add more servers)
- To reduce downtime
- To balance requests among multiple instances

---

#### ■ How it works (simple)

User → Load Balancer → Server 1/2/3/...

LB routes based on algorithms:

- Round robin
- Least connections
- IP hash
- Weighted distribution

---

■ **Benefits**

- Handles traffic spikes
- Improves performance
- Enables auto-scaling in cloud
- Provides health checks (remove unhealthy instances)

---

■ **Example**

Azure Application Gateway / AWS ALB:

If API traffic spikes, LB routes requests to 10 instances instead of 2.

---

■ **Short summary line**

**Load balancers distribute traffic across servers to achieve performance and horizontal scaling.**

---

# ✅ 4) How distributed systems communicate?

■ **Explanation (simple, conceptual)**

Distributed systems consist of multiple services running on different machines.

They communicate over the network using well-defined protocols.

---

■ **Common Communication Methods**

## 🔵 1. REST APIs (HTTP/HTTPS)

Most common

- Language independent
- Simple
- JSON-based

## 🔵 2. gRPC

- High performance
- Binary protocol
- Used between microservices

## 🔵 3. Message Queues

For async communication

- RabbitMQ
- Azure Service Bus
- Kafka

## 🔵 4. Event-driven Communication

Publish → Subscribe model

- Useful for decoupled systems
- Example: OrderPlaced event triggers StockService, PaymentService

## 🔵 5. TCP/UDP Socket communication

Used in gaming, streaming, or real-time apps.

---

### ■ When/Why used

- For microservices
- For distributed backend architectures
- To ensure fault tolerance
- To allow independent scaling

---

### ■ Example (Real-time for your projects)

## Fraud Detection System

- RiskScoreService calls UserProfileService via REST
- TransactionService sends events to Kafka
- AI model service communicates via gRPC

## Expense Tracker

- Angular → .NET API (REST)
- API → SQL Server
- API → EmailService via internal REST API

---

■ **Short summary line**

**Distributed systems communicate via REST, gRPC, message queues, or event-driven messaging to stay scalable and decoupled.**

---