1️⃣ Three Use Cases Where Asynchronous Programming is Needed

Asynchronous programming allows a program to perform long-running tasks without blocking the main thread. Typical use cases include:

1. **I/O-Bound Operations (File, Network, Database)**
    - **Example:** Reading a large file from disk, querying a database, or making HTTP requests.
    - **Reason:** These operations take time waiting for external resources. Asynchronous programming allows other tasks to run while waiting.
    - **Code Example (C#):**

```csharp
async Task<string> ReadFileAsync(string path)
{
  using StreamReader reader = new StreamReader(path);
  return await reader.ReadToEndAsync();
}
```

2. **User Interface Responsiveness**
    - **Example:** In GUI applications (WPF, WinForms, Xamarin), performing long tasks on the main thread will freeze the UI.
    - **Reason:** Asynchronous tasks prevent blocking the UI thread, keeping the app responsive.
    - **Code Example (C#):**

```csharp
async void Button_Click(object sender, EventArgs e)
{
  string data = await GetDataFromApiAsync();
  textBox.Text = data;
}
```

3. **High-Concurrency Server Applications**
    - **Example:** Web servers handling many simultaneous HTTP requests.
    - **Reason:** Asynchronous programming allows the server to handle multiple requests without creating a new thread per request (efficient resource use).

- **Code Example (ASP.NET Core C#):**

```
public async Task<IActionResult> GetUsers()
{
  var users = await _dbContext.Users.ToListAsync();
  return Ok(users);
}
```

Other examples: calling multiple APIs simultaneously, long-running computations that can be offloaded, etc.

2️⃣ Difference Between Thread and Task

Feature	Thread	Task
Definition	A thread is a low-level unit of execution in OS.	A task represents an asynchronous operation, higher-level abstraction.
Creation	Thread t = new Thread(Method); t.Start();	Task t = Task.Run(() => Method()); or async/await
Managed by	Operating System (OS).	.NET Task Scheduler.
Lightweight	Heavier: each thread consumes memory (stack ~1MB).	Lightweight: multiple tasks can share threads in thread pool.
Best For	CPU-bound operations that need dedicated threads.	I/O-bound and CPU-bound asynchronous operations.
Control	You manage lifecycle manually (Start, Abort).	Easier control with continuation, async/await, cancellation tokens.

💡 **Key Idea:**

- **Threads** = low-level OS execution unit.
- **Tasks** = higher-level abstraction to simplify async programming, often using threads under the hood.

1. Why do we need Architecture in any project?

Key Reasons / Benefits

- **Organize complexity & structure**
  As systems grow, you need a disciplined way to divide responsibilities, modules, and interactions. Architecture provides that high-level structure.
  Red Hat+2SEI+2

- **Enable non-functional qualities (quality attributes)**
  Architecture is the way to influence performance, scalability, maintainability, reliability, security, modifiability, etc.
  insights.sei.cmu.edu+2Net Solutions+2

- **Scalability & extensibility**
  As requirements evolve, you want the system to adapt with minimal pain. Good architecture plans for change.
  Apiumhub+2Designveloper+2

- **Separation of concerns / modularity / low coupling**
  You isolate parts (UI, business logic, data access, external systems) so changes in one don't ripple chaos everywhere.
  ITNEXT+2Designveloper+2

- **Team collaboration & parallel development**
  With clear boundaries and contracts between modules, different teams can work independently and integrate smoothly.

- **Risk mitigation & decision making early**
  Architectural decisions (e.g. choice of frameworks, data flow, deployment model) made early reduce costly refactorings later.
  vFunction+2The Knowledge Academy+2

- **Communication & documentation**
  Architecture gives stakeholders (developers, managers, clients) a shared blueprint and language to understand the system.
  The Knowledge Academy+2insights.sei.cmu.edu+2

In summary: Architecture is more than just code structure — it's the blueprint that ensures your system not only works now, but can evolve, scale, and be maintained.

---

2. What is N-Tier Architecture?

Definition & Concept

An **N-Tier architecture** (sometimes “multi-tier” or “layered”) splits an application into **logical (and often physical) tiers/layers**, each with a distinct responsibility.
Microsoft Learn+2Medium+2

“N” can be 3, 4, or more — you choose how many layers you need (presentation, business, data access, service layer, etc.).
Medium+2ITNEXT+2

Typical Layers / Tiers

- **Presentation (UI / Client) Layer**
  Handles user interaction (web UI, desktop, mobile, APIs).
- **Business Logic / Domain / Service Layer**
  Encapsulates business rules, workflows, validations.
- **Data Access / Repository Layer**
  Interfaces with the database (SQL, NoSQL, file storage, etc.).
- **Database / Storage Layer**
  The actual persistent storage (tables, documents, etc.).
- Optionally: **Service / Integration Layer** (for external APIs, messaging), **Infrastructure Layer**, etc.

Characteristics & Observations

- Layers typically depend in one direction (upper layer uses lower).
- Implementation details (e.g. EF, DB) are often tied to lower layers.
- Because dependencies go downward, business logic might depend on data access details (unless carefully abstracted).
- Physical separation possible: e.g. presentation layer deployed on different machines than data layer.
  Microsoft Learn

Pros & Cons

Pros	Cons
Clear separation of concerns	Tendency for tight coupling if abstractions not used
Easy to reason about the responsibilities	Lower flexibility in replacing infrastructure without affecting upper layers
Familiar, well-understood	Can lead to “anemic” domain (business logic scattered)
Easy to test individual layers (if well abstracted)	When models or dependencies cross layers, you get leakage

---

3. What is Onion Architecture?

Definition & Philosophy

Onion Architecture is an architectural pattern introduced by Jeffrey Palermo, emphasizing *dependency inversion* and the idea that the **core domain / business logic** should be at the center, with outer layers depending inward via abstractions.
Stack Overflow+3Code Maze+3Medium+3

The architecture is visualized as concentric rings (like an onion). Dependencies always point toward the center (inward), never outward.

Core Layers (from center outward)

1. **Domain / Core**
    - Entities, domain models, business rules, domain interfaces (abstractions).
    - No dependencies on outer layers.
2. **Application / Use Cases / Service Layer**
    - Coordinates operations, implements use cases, orchestrates domain.
    - Depends on core abstractions, not on infrastructure.
3. **Infrastructure**
    - Concrete implementations: data access, external services, file system, email, etc.
    - Implements interfaces defined in core/application layers.
4. **Presentation / UI / API**
    - Web UI, REST API, UI frameworks, controllers.
    - Calls into application layer, doesn’t depend on infrastructure specifics.

Key Principles & Advantages

- **Dependency Inversion**: Outer layers reference interfaces defined in inner layers; inner layers know nothing about outer ones.
  Medium+2Code Maze+2
- **Separation of concerns & loose coupling**: Core logic is decoupled from external systems (DB, networks, UI).
  progressdesire.com+3Medium+3Code Maze+3
- **Testability**: Because core doesn't depend on infrastructure, you can unit-test domain logic easily using mocks/stubs.
  Medium+1
- **Flexibility / Replaceability**: You can swap infrastructure layers (e.g. database implementation, message bus) without touching domain logic.
  Medium+3Medium+3Code Maze+3

Differences vs N-Tier / Layered

- In classic layering, the business logic layer often depends on data access layer (or its interfaces) — i.e., coupling downward.
- In Onion, the business logic is core and **doesn't depend on lower (infrastructure) layers** — infrastructure depends on domain.
- Thus, interfaces/abstractions usually live in inner layers (domain or application), not outer layers.
  ITNEXT+3Stack Overflow+3Stack Overflow+3
- **Better enforce the Dependency Inversion Principle (DIP).**
  Medium+2Code Maze+2

When to Use / Considerations

- Suited for medium-to-large systems where maintainability, testability, and flexibility are important.
- More initial setup and abstraction overhead.
- Not always needed for small/simple apps.

---

4. Interview Question: Is LINQ slow in execution?

(And what about deferred vs eager execution?)

Short, balanced answer

LINQ itself is **not inherently slow**, but its performance depends on how and when queries are executed. You must understand **deferred execution** vs. **eager execution**, and how multiple enumerations or misuse can lead to inefficiencies.

Deferred Execution vs Eager Execution

Deferred Execution

- With deferred execution, the LINQ query is **not executed immediately** when you define it; it's executed **later, when you enumerate or force it** (e.g. foreach, .ToList(), .ToArray(), etc.). Microsoft Learn+2ITNEXT+2
- It allows building a query pipeline, combining filters, projections, etc., without executing until needed.
- It can improve performance because it might avoid unnecessary work or merge operations before executing. blog.somewhatabstract.com+3Microsoft Learn+3Progress.com+3
- But if you enumerate the same query multiple times, the underlying data source will be queried multiple times (unless cached). DEV Community+2Progress.com+2

Example:

```
IEnumerable<int> query = numbers.Where(n => n % 2 == 0);

// no execution yet


foreach(var n in query)

   Console.WriteLine(n);  // actual execution happens here
```

Eager Execution

- With eager execution, the query is **executed immediately**, and the results are materialized (e.g. into a List<T>).
- Methods like .ToList(), .ToArray(), .Count(), .First(), etc. force execution.
- Useful when you want the result right away, or when you want to cache the results and prevent re-query.
- But can be less efficient if the data is large or if you don't need the full result set.

Example:

```
List<int> evens = numbers.Where(n => n % 2 == 0).ToList();

// execution happens here
```

When does LINQ *appear* slow / pitfalls to watch for

- **Multiple enumeration**: If you do foreach on the same deferred query multiple times, it re-executes each time.
- **Complex queries with large data sets**: If too many filters, projections, or joins without optimization, you might get performance overhead.
- **Using non-optimal operators**: E.g. repeatedly calling .OrderBy() or .Distinct() inefficiently.
- **Mixing LINQ to Objects and LINQ to SQL/EF poorly**: Sometimes writing queries that are executed in memory rather than translated to SQL efficiently.
- **Deferred execution hiding exceptions or delays**: Errors in query will surface at enumeration time, possibly surprising.