3. Azure DevOps & Azure Cloud

 Basic Level

- What is Azure DevOps and its main components?

Answer:
Azure DevOps is a **DevOps lifecycle toolset** by Microsoft for **planning, developing, testing, and delivering** applications.

Main components:

- **Azure Boards**: Work item tracking (Agile, Scrum, Kanban).
- **Azure Repos**: Git repositories with branch policies.
- **Azure Pipelines**: CI/CD automation.
- **Azure Test Plans**: Manual and exploratory testing.
- **Azure Artifacts**: Package management (NuGet, npm, etc.).

I've used Boards and Pipelines daily in my projects for agile tracking and CI/CD automation.

- What are pipelines in Azure DevOps?

Answer:
Pipelines are workflows that **automate the build, test, and deploy** process.

- **CI (Continuous Integration):** Compiles code, runs tests.
- **CD (Continuous Deployment):** Pushes artifacts to environments (Dev, QA, Prod).
- Pipelines can be built using:
    - **Classic editor (GUI-based)** or
    - **YAML (code-based)**.

I've used both formats for .NET Core build and deploy processes.

- What is a service connection in Azure DevOps?

Answer:
A service connection securely stores credentials to connect Azure DevOps to **external services**, like:

- Azure subscriptions (for deployments)
- GitHub, DockerHub, etc.

Types:

- Azure Resource Manager
- GitHub personal access token
- Docker registry credentials

I typically use **Azure Resource Manager** connections to deploy apps to **App Service** or **Key Vault**.

- Explain the concept of a YAML-based pipeline.

Answer:
YAML pipelines define CI/CD logic using .yml files stored in the repository, offering **version control**, **reusability**, and **flexibility**.

Example:

```
yaml
CopyEdit
trigger:
  branches: [main]

jobs:
- job: Build
  pool:
    vmImage: 'windows-latest'
  steps:
  - task: DotNetCoreCLI@2
    inputs:
      command: 'build'
      projects: '**/*.csproj'
```

I've used YAML pipelines to **build, test, and deploy** .NET Core apps with **Dev, QA, Prod environments**.

 Intermediate Level

- How do you integrate your .NET Core app with an Azure DevOps pipeline?

Answer:
Steps:

1. Push code to Azure Repos/GitHub.
2. Create a YAML pipeline or use the Classic editor.
3. Use built-in tasks:
    - DotNetCoreCLI@2 for build, restore, publish, test.
    - PublishBuildArtifacts@1 to generate artifacts.
4. Add a release pipeline to deploy to App Service.

I've used this setup for **CI/CD of APIs and front-end apps**, deploying to **Azure App Services** or containers.

- What is the difference between Classic and YAML pipelines?

Feature	Classic Pipeline	YAML Pipeline
UI/UX	GUI-based (drag-and-drop)	Code-based (YAML file)
Version Control	Not stored in repo	Stored and versioned with source code
Flexibility	Less flexible for branching	Easy multi-branch or condition setup

I've used Classic for quick proof-of-concepts and YAML for long-term projects requiring **DevOps as Code**.

- How do you use Azure Artifacts and NuGet packages?

Answer:
Azure Artifacts lets you host and consume NuGet, npm, Maven, etc.

Use cases:

- Host internal libraries/packages.
- Share versioned modules across teams.

Steps:

- Create a feed in Azure Artifacts.
- Push NuGet packages using dotnet nuget push.
- Add the feed URL in nuget.config.

I've used Artifacts to **share custom libraries** across microservices.

 Advanced Level

- How do you secure secrets and credentials in Azure DevOps?

Answer:

1. Use **Azure Key Vault** and reference secrets in YAML.
2. Use **Pipeline Secrets** (Library → Variable Groups → Mark as secret).
3. Avoid hardcoding secrets in YAML; instead, use environment variables.

Example:

yaml
CopyEdit
env:
 ConnectionString: $(MySecret)

I integrate Azure Key Vault for **connection strings**, **API keys**, and use **RBAC** to limit access.

- How do you set up CD to Azure App Services?

Answer:

1. Create a **service connection** to Azure.
2. Use **release pipeline** or YAML job with AzureWebApp@1 task:

yaml
CopyEdit
- task: AzureWebApp@1
 inputs:
  azureSubscription: 'MyConnection'

```
  appType: 'webApp'
  appName: 'my-dotnet-app'
  package: '$(System.DefaultWorkingDirectory)/**/*.zip'
```

I configure **slot-based deployments** for production with **approval gates** in Dev → QA → Prod flow.

- What is infrastructure as code? Have you used ARM templates or Bicep?

Answer:
Infrastructure as Code (IaC) automates provisioning of cloud infrastructure using code.

Options in Azure:

- **ARM Templates** (JSON-based)
- **Bicep** (simplified syntax for ARM)
- **Terraform** (cross-platform)

I've used **ARM templates and Bicep** to deploy:

- App Service plans
- Key Vaults
- SQL databases
- Storage accounts

Benefits include **repeatable infrastructure**, **version control**, and **consistency** across environments.

- How would you rollback a failed deployment?

  Answer:

  Enable **deployment slots** and use **swap on success** strategy.

  Use **Release Pipeline gates** for approval and **manual intervention**.

  Implement **automatic rollback** via health check failure or **custom scripts**.

  Maintain **previous deployment artifacts** for rollback.

  I configure pipelines to deploy to **staging slots first**, validate, then swap to production.

✅4. Programming Languages & OOPs (C#, Python)

 C# & OOPs

- What is the difference between abstract class and interface?

Feature	Abstract Class	Interface
Inheritance	Single inheritance	Multiple inheritance
Members with logic	Can have implemented methods	Cannot have implementations (until C# 8 default methods)
Fields/Constructors	Can have fields and constructors	Cannot contain fields or constructors
Use Case	When base class has **shared code**	When **contracts** are needed without implementation

Example:

csharp
CopyEdit
```
abstract class Animal {
   public abstract void Speak();
   public void Eat() => Console.WriteLine("Eating...");
}

interface IWalkable {
   void Walk();
}
```

Usage:
I use **abstract classes** for base business logic and **interfaces** for defining behavior (like IDisposable, IRepository<T>).

- What are value types and reference types?
- **Value Types** (e.g., int, float, bool, struct):
  - Stored on the **stack**
  - Copied by value
  - Fast but limited in size
- **Reference Types** (e.g., class, string, array, object):
  - Stored on the **heap**
  - Copied by reference
  - Allow shared state

Example:

csharp
CopyEdit
```
int a = 5;
int b = a; // b is a copy

Person p1 = new Person();
Person p2 = p1; // both refer to the same object
```

- What is polymorphism and where have you used it?

Polymorphism allows different classes to be treated through a common interface or base class.

- **Compile-time (Overloading):** Same method name, different signatures
- **Runtime (Overriding):** Base class method overridden in derived class using virtual and override

Example:

csharp
CopyEdit

```
public class Animal {
   public virtual void Speak() => Console.WriteLine("Animal speaks");
}
public class Dog : Animal {
   public override void Speak() => Console.WriteLine("Dog barks");
}
```

Usage:
I used polymorphism for implementing **business rules** via interfaces (e.g., IValidator) and strategy patterns.

- Explain the use of async and await.

Answer:
async and await enable **asynchronous programming** using **Task-based** APIs.

- Improves responsiveness
- Avoids thread blocking
- Common in web/API calls and I/O operations

Example:

csharp
CopyEdit
```
public async Task<string> GetDataAsync() {
   var response = await httpClient.GetStringAsync("https://api.example.com/data");
   return response;
}
```

Usage:
In .NET Core APIs, I use async for:

- **Database access** via EF Core
- **API integrations**
- **File I/O operations**

- What is boxing and unboxing?

Answer:

- **Boxing:** Converting a **value type to object** type (stored in heap)
- **Unboxing:** Converting back to **value type** from object

Example:

csharp
CopyEdit
```
int num = 123;
object obj = num; // Boxing
int result = (int)obj; // Unboxing
```

Impact:
Boxing can affect performance; hence I avoid it in **loops and collections** (use generics like List<int>).

✅6. Software Design Principles (SOLID, CQRS)

□ SOLID Principles

- What is the Single Responsibility Principle? Provide a real example.

Answer:
Single Responsibility Principle (SRP) states that a class should **have only one reason to change**, i.e., it should only do one thing.

Bad Example (violates SRP):

```
csharp
CopyEdit
public class Invoice {
   public void CalculateTotal() { ... }
   public void SaveToDatabase() { ... }
   public void GeneratePDF() { ... }
}
```

Good Example (SRP):

```
csharp
CopyEdit
public class InvoiceCalculator { public void CalculateTotal(Invoice invoice) { ... } }
public class InvoiceRepository { public void Save(Invoice invoice) { ... } }
public class InvoicePDFGenerator { public void Generate(Invoice invoice) { ... } }
```

Where I used it:
In my e-commerce project, I split invoice logic into **calculation**, **persistence**, and **report generation** to maintain **modularity and testability**.

- Explain Open/Closed Principle.

Answer:
Open/Closed Principle: Classes should be **open for extension but closed for modification**.

Example:

```
csharp
CopyEdit
public interface IDiscount {
   decimal ApplyDiscount(decimal amount);
}

public class StudentDiscount : IDiscount { ... }
public class SeniorDiscount : IDiscount { ... }

public class BillingService {
   public decimal GetFinalAmount(IDiscount discount, decimal amount) {
      return discount.ApplyDiscount(amount);
   }
}
```

You can add new discount strategies without modifying BillingService.

- What is the Interface Segregation Principle? When do you use it?

Answer:
A class should **not be forced to implement interfaces it doesn't use**. Break down large interfaces into **smaller, specific ones**.

Bad Example:

```
csharp
CopyEdit
public interface IWorker {
   void Work();
   void Eat();
}
```

Good Example:

```
csharp
CopyEdit
public interface IWorkable { void Work(); }
public interface IFeedable { void Eat(); }
```

Where I used it:
In a background job system, I separated IProcessJob, IScheduleJob, and ILogJob to keep responsibilities clean.

- How does the Dependency Inversion Principle work in a layered architecture?

Answer:
DIP states:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Use **interfaces** and **dependency injection** to decouple layers.

Example:

```
csharp
CopyEdit
// Application Layer
public class OrderService {
   private readonly IOrderRepository _repo;
   public OrderService(IOrderRepository repo) { _repo = repo; }
}

// Infrastructure Layer
public class OrderRepository : IOrderRepository { ... }
```

In Startup.cs:

```
csharp
CopyEdit
services.AddScoped<IOrderRepository, OrderRepository>();
```

This enables **unit testing** and makes swapping implementations easy (e.g., switch from SQL to MongoDB).

 CQRS & Patterns

- What is the main benefit of CQRS over CRUD?

Answer:
CQRS (Command Query Responsibility Segregation) separates **read and write logic** into different models.

Benefits:

- Better scalability (reads and writes can scale independently)
- Clearer domain logic separation
- Optimized data models per operation
- Enables **event sourcing** and **audit logging**

I've used CQRS in modules with **high read traffic**, like dashboard reporting.

- Where would you apply CQRS in a .NET Core app?

Answer:

- In **API layers**: separate Command and Query handlers for each use case.
- Commands update DB, Queries fetch DTOs.
- Use MediatR for dispatching.

Example:

plaintext
CopyEdit
➢POST /CreateOrder → CommandHandler → DbContext.Save()
➢GET /GetOrderById → QueryHandler → DTO response

Used in my inventory project for **order creation and lookup** separation.

- Explain Mediator Pattern and how it relates to CQRS.

Answer:
The **Mediator pattern** defines a central object to **encapsulate communication** between components.

In .NET Core:

- **MediatR** is commonly used to implement CQRS.
- It allows sending commands and queries via IMediator.Send().

Example:

csharp
CopyEdit
```
await _mediator.Send(new CreateUserCommand { Name = "Saurabh" });
```

Benefits:

- Decouples controllers from business logic
- Supports clean architecture and testability

- Difference between Command Handler and Query Handler.

Aspect	Command Handler	Query Handler
Purpose	Performs write operations	Fetches and returns data
Side Effects	Yes (e.g., DB updates, events)	No side effects (read-only)
Returns	Void or identifier/result	DTO or read model
Examples	CreateOrderHandler, UpdateUser	GetUserByIdHandler, GetOrdersList

Where I used it:
In user management, I separated **update profile (command)** from **get user details (query)**.

- What is the Unit of Work Pattern?

Answer:
Unit of Work (UoW) ensures that multiple **related DB operations** are executed as a **single transaction**.

In EF Core:
DbContext acts as Unit of Work.

Benefits:

- Maintains data consistency
- Reduces the number of database calls
- Coordinates multiple repositories

Example:

csharp
CopyEdit
```
using (var context = new AppDbContext()) {
   var user = new User { Name = "Akanksha" };
   var order = new Order { Amount = 500 };

   context.Users.Add(user);
   context.Orders.Add(order);
   context.SaveChanges(); // one atomic transaction
}
```

✅7. Web Deployment (IIS, GitHub)

Deployment via IIS

- What are the steps to deploy a .NET Core web app to IIS?

Answer:
To deploy a .NET Core web application to IIS:

1. **Publish the app**
   Use Visual Studio or CLI:

   bash
   CopyEdit
   dotnet publish -c Release -o ./publish

2. **Install .NET Core Hosting Bundle** on the IIS server
   - It configures the reverse proxy and adds the ASP.NET Core Module.
3. **Create a site in IIS:**
   - Go to IIS Manager → Sites → Add Website
   - Point **Physical Path** to publish folder
4. **Set application pool:**
   - Use **No Managed Code**
   - Ensure app pool is using **Integrated pipeline**
5. **Set permissions:**
   - Grant read + execute permission to **IIS_IUSRS** or **Network Service**
6. **Configure bindings:**
   Add port or domain name as required (e.g., localhost:5001, or myapp.local).

Usage:
I've deployed REST APIs and admin portals using IIS with **custom domains**, **HTTPS**, and **logs enabled**.

- How do you configure a hosting environment in IIS?

Answer:

You configure environment-specific settings using:

- **ASPNETCORE_ENVIRONMENT** variable
  Set in:
  - System Environment Variables
  - Or in web.config:

    xml
    CopyEdit
    <environmentVariables>
      <add name="ASPNETCORE_ENVIRONMENT" value="Production" />
    </environmentVariables>

- The environment name must match values like Development, Staging, Production in your code (e.g., Startup.cs).

Where I used this:
I configure **different appsettings for staging and production** using this approach.

- How do you troubleshoot a 500 Internal Server Error after deployment?

Answer:

1. **Enable stdout logging** in web.config:

   xml
   CopyEdit
   <aspNetCore stdoutLogEnabled="true" stdoutLogFile=".\logs\stdout" />

2. Check logs at logs\stdout*
3. Ensure:
   - Hosting bundle is installed
   - Application pool has permissions
   - appsettings.Production.json is not missing/invalid
   - Required files in /publish folder exist (like web.config, .dll)
4. Use **Event Viewer** for detailed exceptions.

Pro Tip:
Set launchSettings.json to match IIS port to replicate issues locally.

 Git & GitHub

- What is the difference between git pull and git fetch?

Command	Description
git fetch	Gets latest changes from remote, doesn't merge
git pull	Fetch + Merge into current branch automatically

Example:

bash
CopyEdit
git fetch origin
git merge origin/main
vs
git pull origin main

Best Practice:
I use git fetch before rebasing to **avoid unintentional merges** in shared branches.

- How do you resolve merge conflicts?

Answer:

1. When you run:

   bash
   CopyEdit
   git pull origin main

   and conflict occurs, Git marks it with <<<<<<<, =======, >>>>>>>.

2. Open file → resolve manually or use a GUI tool (e.g., VSCode Source Control)
3. Add and commit:

   bash
   CopyEdit
   git add .
   git commit -m "Resolved conflict"

4. Continue rebase or push.

Where I used this:
During team sprints, resolving conflicts in shared services like Startup.cs or model files is common.

- What is the purpose of .gitignore?

Answer:

.gitignore prevents certain files/folders from being tracked by Git.

Common exclusions:

pgsql
CopyEdit
bin/
obj/
*.user
*.log
*.suo
.vscode/
appsettings.Development.json

Why it matters:
Prevents clutter, protects sensitive info, and keeps repo clean.

- How do you revert a commit?

Answer:

- To **undo a local commit** (but keep changes):

  bash
  CopyEdit
  git reset --soft HEAD~1

- To **undo and discard changes:**

  bash
  CopyEdit
  git reset --hard HEAD~1

- To **revert a pushed commit** (safe for shared branches):

bash
CopyEdit
git revert <commit-hash>

Where I used this:
If a build breaks after push, I revert the last commit using revert for safe rollback.
