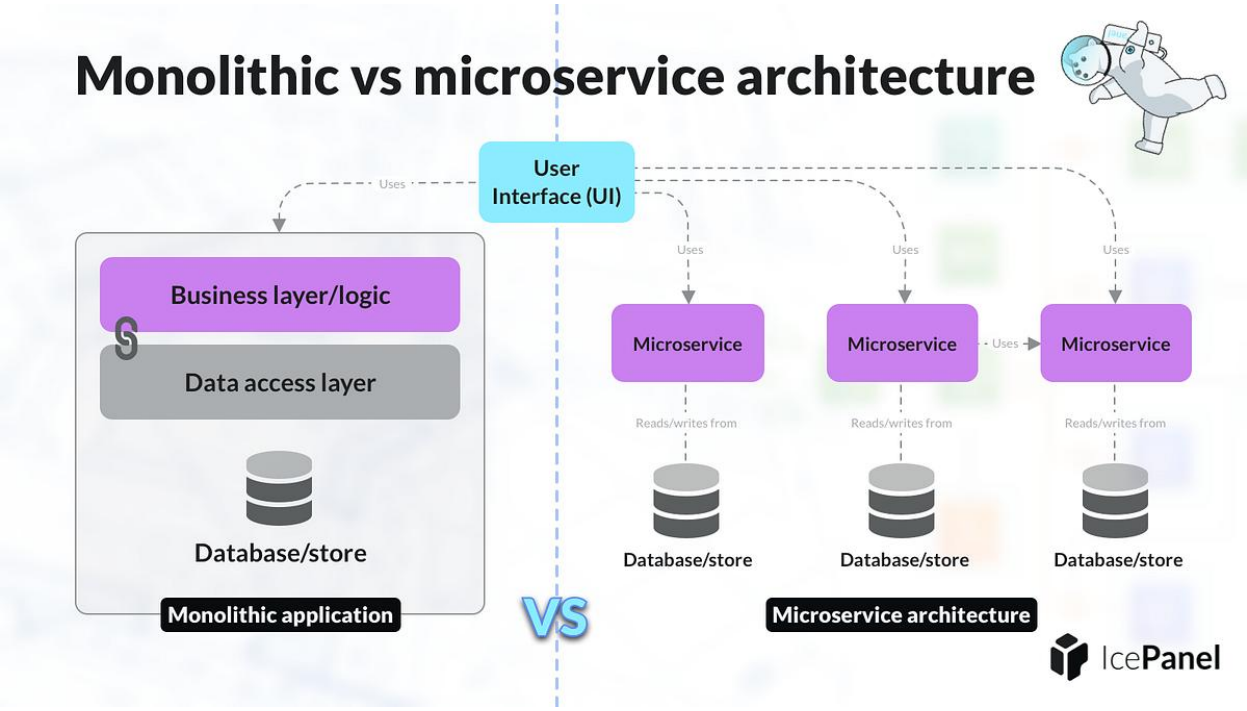
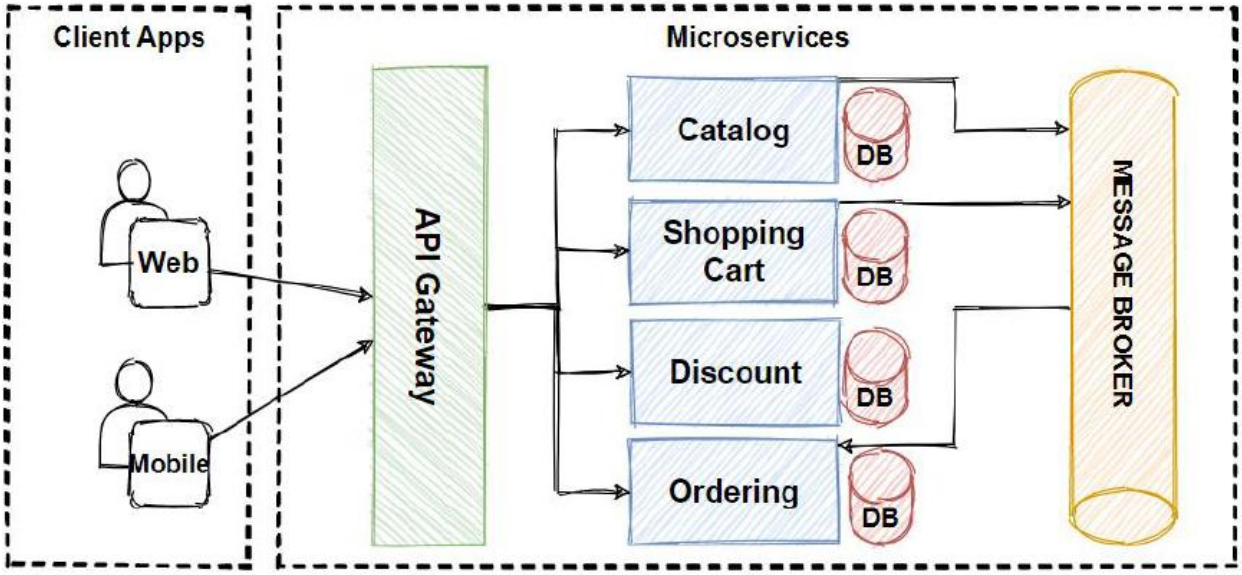
✅ Types of Architectures Used in .NET Project Development





1️⃣ Monolithic Architecture

A single, large application where all modules (UI, business logic, data access) are packaged and deployed as one unit.

✓ Structure

[UI]

[Business Logic]

[Data Access]

[Database]

(All inside one project/application)

✓ When Used

- Small applications
- Startups / quick MVPs
- When deployment speed is important

✓ Pros

- Simple to develop & deploy
- Easy debugging
- Less DevOps complexity

✓ Cons

- Hard to scale features independently
- One change → redeploy full app
- Tight coupling

2️⃣ N-Tier / Layered Architecture (Most Common in .NET Projects)

Typical layers:

[Presentation Layer (UI - MVC/API)]

[Business Logic Layer (Services)]

[Data Access Layer (Repository)]

[Database (SQL Server)]

✓ When Used

- Medium-sized enterprise apps

- CRUD-heavy systems
- Traditional .NET corporate projects

✓ **Pros**

- Separation of concerns
- Testable & maintainable
- Beginner friendly

✓ **Cons**

- Can become tightly coupled
- Hard to scale per feature
- All layers deployed together

3️⃣ Hexagonal Architecture (Ports & Adapters)

Focuses on keeping business logic isolated from external systems.

✓ **Structure**

```
        [ UI ]
        [ API ]
         |
   ------------------
   |  Application   |
   |    Core        |
   ------------------
  /       |       \
[DB Adapter] [Email Adapter] [External Services]
```

✓ **When Used**

- Domain-driven design (DDD) projects
- Systems needing high testability

- Complex enterprise apps

✓ **Pros**

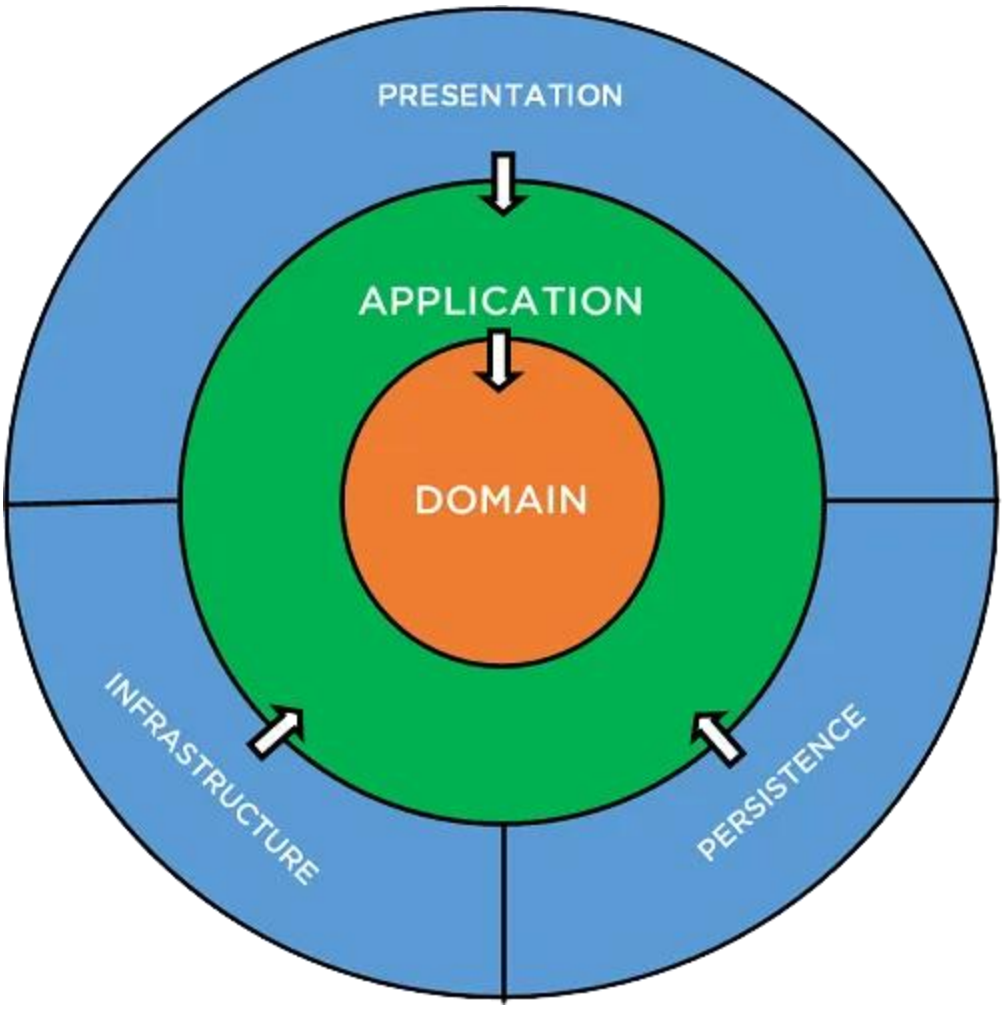
- Highly testable
- Technology independent
- Clean separation of concerns

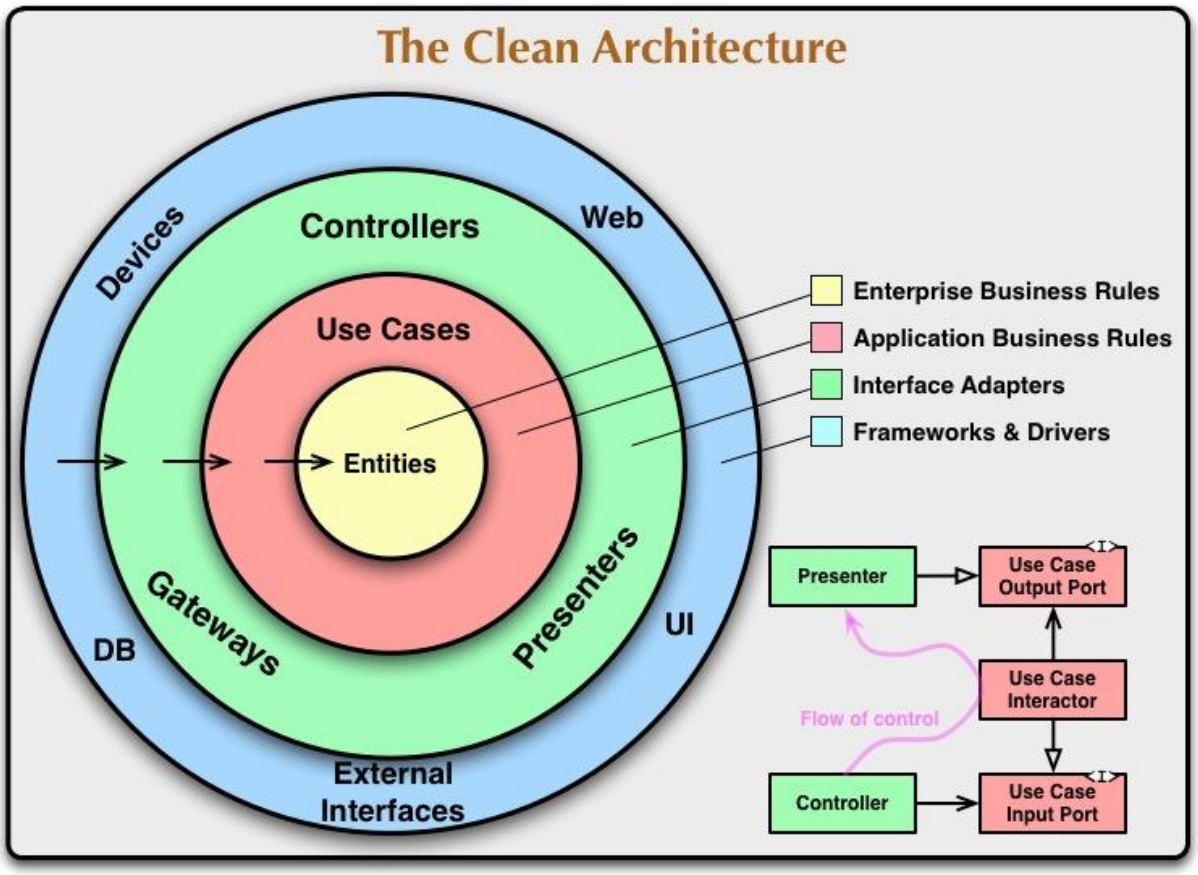
✓ **Cons**

- Slightly complex for beginners
- More boilerplate code

4️⃣ Clean Architecture (Most Popular in Modern .NET 6/7/8)

Designed by Robert C. Martin (Uncle Bob).
Core business logic is protected from external frameworks.


PRESENTATION
APPLICATION
DOMAIN
INFRASTRUCTURE
PERSISTENCE



✓ Layer Structure

- **Entities**
- **Use Cases / Application Layer**
- **Infrastructure Layer**
- **Presentation Layer (API/MVC)**

✓ When Used

- Enterprise apps, banking, healthcare
- Apps needing long-term maintainability
- .NET with CQRS, DDD

✓ Pros

- Highly maintainable

- Highly testable
- Technology/framework independent

✓ **Cons**

- Requires design understanding
- Higher initial setup effort

5️⃣ Microservices Architecture

Application is broken into small services — deployed and scaled independently.

✓ **Structure**

[Order Service] [Payment Service] [Product Service]

SQL DB SQL DB MongoDB

REST API REST API REST API

Gateway (Ocelot)

✓ **When Used**

- Large enterprise apps
- E-commerce, financial, distributed systems
- Apps requiring independent scaling

✓ **Pros**

- Highly scalable
- Independent deployment
- Technology flexibility (SQL, NoSQL, .NET, Node etc.)

✓ **Cons**

- Complex DevOps
- Services communication overhead

- Requires CI/CD, Docker, Kubernetes

6️⃣ Serverless Architecture (Azure Functions + .NET)

Code executes in the cloud without managing servers.

✓ When Used

- Event-driven systems
- Background jobs
- Pay-per-use workloads

✓ Pros

- No server management
- Scales automatically
- Cost efficient

✓ Cons

- Cold start issues
- Vendor lock-in (Azure)

7️⃣ Event-Driven Architecture

Applications communicate using events (Kafka, Azure Service Bus, RabbitMQ).

✓ Suitable For

- E-commerce: OrderPlaced → PaymentProcessing
- High performance + asynchronous systems

✓ Pros

- Loose coupling
- Highly scalable
- Async processing

✓ **Cons**

- Hard debugging
- Event failures need dead-letter queues

8️⃣ CQRS (Command Query Responsibility Segregation)

Read and Write operations are separated.

✓ **Structure**

Commands → Modify Data

Queries → Read Data

Often used with:

- **MediatR**
- **Clean Architecture**
- **Event Sourcing**

✓ **Pros**

- High performance
- Optimized reads
- Works well for large data systems

✓ **Cons**

- More complexity
- Requires careful design

9️⃣ Onion Architecture

Similar to Clean Architecture but with a strict dependency inward rule.

✓ **Pros**

- High maintainability

- Clear separation of core logic and infrastructure

✓ **Cons**

- More initial planning required

🔥 **Summary Table**

Architecture	**Complexity**	**Scalability**	**Best For**
Monolithic	Low	Low	Small apps
N-Tier	Low–Medium	Medium	Traditional .NET apps
Hexagonal	Medium	Medium	DDD, clean code
Clean Architecture	Medium–High	High	Enterprise systems
Microservices	High	Very High	Distributed systems
Serverless	Medium	Very High	Event-driven workloads
Event Driven	Medium	High	Async systems
CQRS	Medium–High	High	High-read apps