**Distributed Transaction Patterns in Event-Driven Microservices**

[**https://www.figma.com/board/R33nvmnqGinwz34I4ncYbp/Untitled?node-id=0-1&p=f&t=zlQqWHDY62LF1LAu-0**](https://www.figma.com/board/R33nvmnqGinwz34I4ncYbp/Untitled?node-id=0-1&p=f&t=zlQqWHDY62LF1LAu-0)

A diagram of a product

AI-generated content may be incorrect.

**Event-driven architecture (EDA)**

An event-driven architecture (EDA) can be effectively used in an order processing system for an e-commerce platform. Here’s a detailed explanation of how it works, broken down into its components, flow, and benefits.

A diagram of an event channel

Description automatically generated

**I- Business Requirements**  
- Process orders efficiently when customers place them.  
- Notify customers about their order status (confirmation, shipment, delivery).  
- Integrate with inventory management to update stock levels in real-time.  
- Handle payments and coordinate with a payment gateway.  
- Support future integrations (e.g., analytics, fraud detection).

**II- Event-Driven Architecture Overview**  
1- Producers: Components that generate events based on actions.  
2- Consumers: Services that react to these events to perform tasks.  
3- Broker: Middleware (like Kafka / RabbitMQ / AWS EventBridge/ Azure Event grid) that routes events from producers to consumers.

**III- System Components**  
Order Service: Handles order creation.  
Inventory Service: Updates stock levels for ordered products.  
Payment Service: Processes payments.  
Notification Service: Sends email/SMS updates to customers.  
Shipping Service: Manages shipment details.  
Event Broker: Routes events between producers and consumers.

**IV- Step-by-Step Flow**

*Step 1: Order Placement*  
The customer places an order via the e-commerce website or mobile app.  
The Order Service creates the order and publishes an OrderCreated event to the Event Broker.

Event: { “eventType”: “OrderCreated”, “orderId”: 12345, “items”: […], “userId”: 789 }

*Step 2: Inventory Update*  
The Inventory Service, subscribed to the OrderCreated event, receives it from the Event Broker.  
It checks the availability of the ordered items in stock.  
If items are available:

It reserves the stock.  
Publishes an InventoryUpdated event:  
{ “eventType”: “InventoryUpdated”, “orderId”: 12345, “status”: “success” }

If items are unavailable:

Publishes an InventoryUpdateFailed event: { “eventType”: “InventoryUpdateFailed”, “orderId”: 12345, “reason”: “OutOfStock” }

*Step 3: Payment Processing*The Payment Service, subscribed to the InventoryUpdated event, processes the payment for the order.

Publishes a PaymentProcessed event:  
{ “eventType”: “PaymentProcessed”, “orderId”: 12345, “status”: “success” }

If the payment fails:

Publishes a PaymentFailed event:  
{ “eventType”: “PaymentFailed”, “orderId”: 12345, “reason”: “InsufficientFunds” }

*Step 4: Order Fulfillment*The Shipping Service, subscribed to the PaymentProcessed event, creates a shipment for the order.

Publishes a ShipmentCreated event:  
{ “eventType”: “ShipmentCreated”, “orderId”: 12345, “trackingId”: “XYZ987” }

*Step 5: Notifications*  
The Notification Service listens to all relevant events (OrderCreated, PaymentProcessed, ShipmentCreated).  
It sends real-time updates to the customer:

Order confirmation.  
Payment receipt.  
Shipment tracking information.

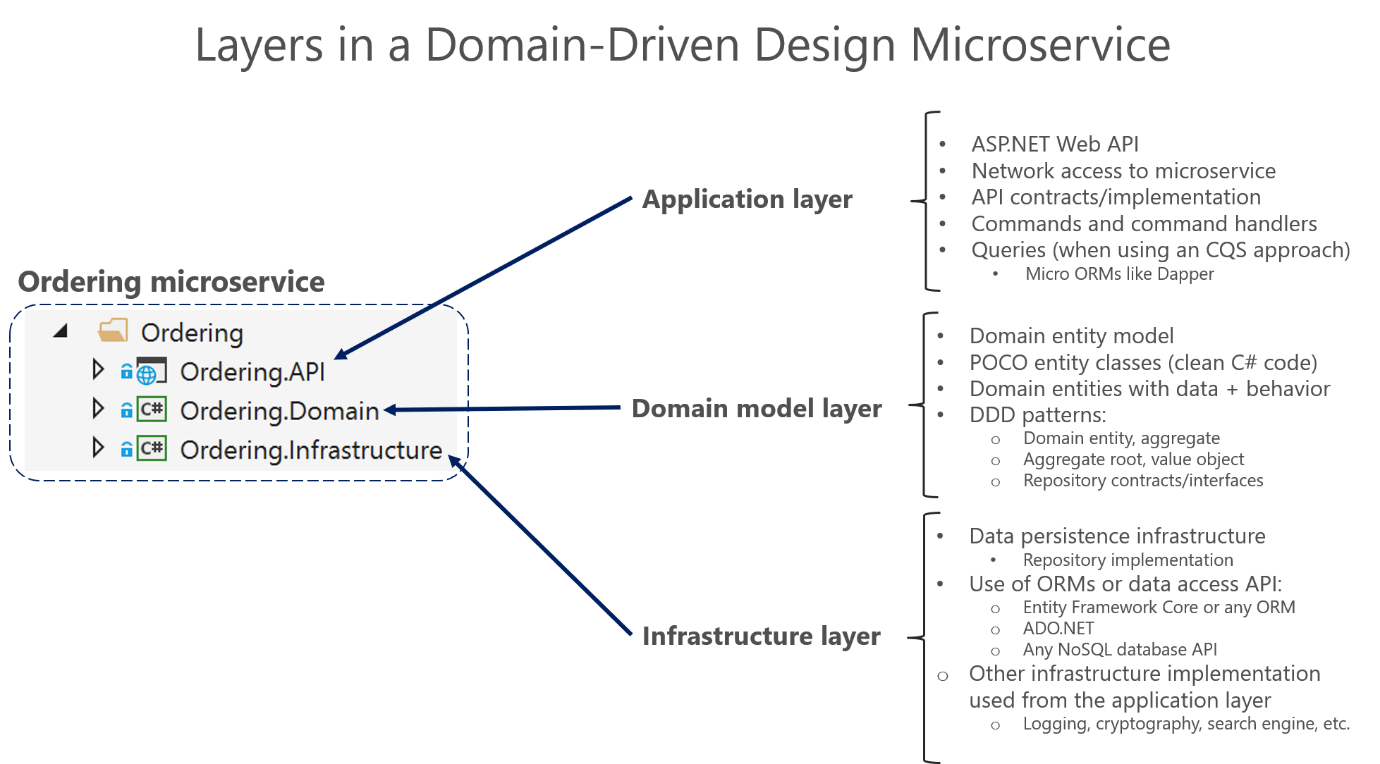
System Characteristics  
Event Broker  
Middleware like Apache Kafka or RabbitMQ is used to handle events.  
Ensures loose coupling between services.  
Supports durability and replaying of events for fault tolerance.

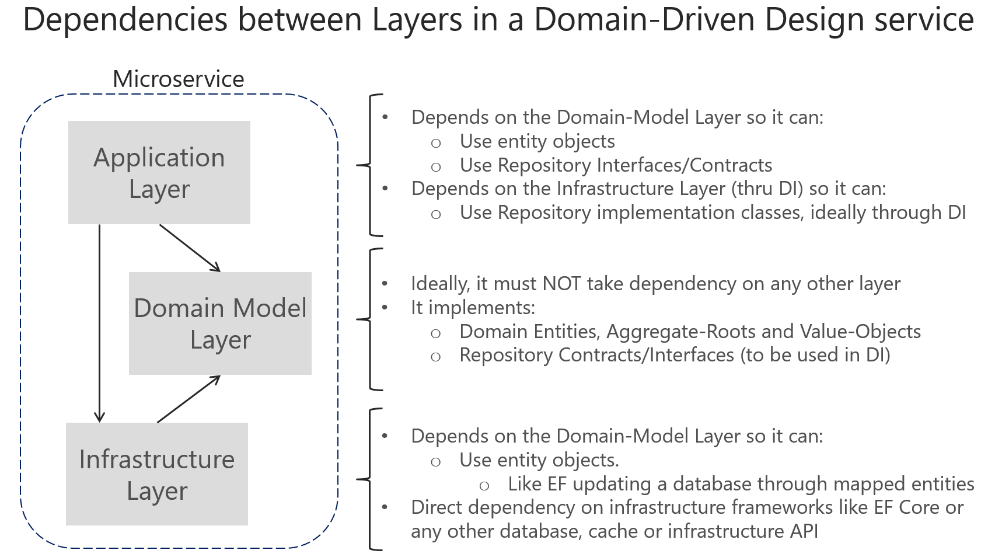
**Domain-Driven Design: A Brief Overview**

Domain-Driven Design, pioneered by Eric Evans, focuses on aligning software design closely with business domains. DDD emphasizes understanding the business domain deeply and modeling it accurately within the software. The key concepts of DDD include:

1. **Domain**: The sphere of knowledge and activity around which the business is centered.
2. **Entities**: Objects that have a distinct identity within the domain.
3. **Value Objects**: Objects that describe certain aspects of the domain without needing a distinct identity.
4. **Aggregates**: Clusters of entities and value objects that are treated as a single unit.
5. **Repositories**: Mechanisms for retrieving and storing aggregates.
6. **Services**: Operations or logic that don’t naturally fit within entities or value objects.

DDD also introduces strategic design principles like Bounded Contexts, which help manage the complexity by dividing the domain into distinct sections with clear boundaries.





<https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice>

|  |
| --- |
| **1️⃣ Aggregate Structure**  Order (Aggregate Root)  ├── OrderItem (Entity)  ├── PaymentDetails (Value Object)  ├── ShippingAddress (Value Object) |
| 2️⃣ Aggregate Root (Order)  public class Order  {  public int Id { get; private set; }  public List<OrderItem> Items { get; private set; } = new();  public PaymentDetails Payment { get; private set; }  public void AddItem(OrderItem item)  {  Items.Add(item); // Ensuring modification is done through the root  }  } |
| In **Domain-Driven Design (DDD)**, an **Aggregate** is a cluster of related domain objects that should be treated as a single unit. The **Aggregate Root** is the main entity that ensures consistency across the Aggregate.  public class Order // This is the Aggregate Root  {  public int Id { get; private set; }  public List<OrderItem> Items { get; private set; } = new();  public PaymentDetails Payment { get; private set; }  public void AddItem(OrderItem item)  {  Items.Add(item); // Ensuring modification is done through the root  }  } |
| 🛠 Supporting Entities Inside the Aggregate  public class OrderItem // Part of the Order Aggregate, NOT an Aggregate Root  {  public int ProductId { get; private set; }  public int Quantity { get; private set; }  public OrderItem(int productId, int quantity)  {  ProductId = productId;  Quantity = quantity;  }  }  public class PaymentDetails // Value Object, part of the Order Aggregate  {  public string CardNumber { get; private set; }  public string ExpiryDate { get; private set; }  public PaymentDetails(string cardNumber, string expiryDate)  {  CardNumber = cardNumber;  ExpiryDate = expiryDate;  }  } |

**Key Differences**

| **Concept** | **Aggregate** | **Aggregate Root** |
| --- | --- | --- |
| **Definition** | A group of related domain objects. | The main entity that controls access and ensures consistency. |
| **Example** | Order + OrderItem + PaymentDetails | Order |
| **Access Control** | Other entities inside the aggregate cannot be accessed directly. | All changes must go through the Aggregate Root (Order). |
| **Data Storage** | All objects are saved/loaded together. | The Aggregate Root is the entry point for data operations. |

**Why Use an Aggregate Root?**

✔ Ensures **data consistency** across related entities.  
✔ Prevents **direct modification** of child objects from outside.  
✔ Helps enforce **business rules** at the Aggregate level.

**🎯 E-Commerce Architecture Design (Event-Driven + DDD)**

**✔ Approach Breakdown:**

* **Domain-Driven Design (DDD) for Cart & Product Selection** ✅
* **Event-Driven Architecture (EDA) for Order Processing** ✅
* **Service Bus for Payment (Reliable, FIFO Guaranteed)** ✅
* **Event Grid for Inventory & Notifications (Low Latency, Fan-Out)** ✅
* **Event Hub + Log Analytics for Observability & Telematics** ✅

**📌 Technologies Used:**

* **Azure Service Bus** → Payment Processing (Reliable & FIFO)
* **Azure Event Grid** → Order Completion Event (Fan-Out)
* **Azure Event Hub + Log Analytics** → Event Telemetry
* **.NET 7+ (C#)** → Microservices

**🚀 Step-by-Step: Choosing a Product & Placing an Order**

**1️⃣ Product & Cart Management (DDD-Based Approach)**

✔ **Products Service (Product Domain)** → REST API (CRUD)  
✔ **Cart Service (Cart Domain)** → **Maintains session-based cart storage**  
✔ **Orders Service (Order Domain)** → Orders are created only after payment is confirmed

💡 **Why DDD?**

* Product & Cart **don’t require async communication**, so **REST APIs** are sufficient.
* Orders depend on **successful payment**, which requires **strong consistency** (hence no event-driven model for this step).

**🚀 Step 2: Order Placement & Payment Processing**

**2️⃣ Payment Processing (Using Azure Service Bus for Reliability)**

✔ When a user clicks **“Pay Now”**, the **Order Service** sends a message to **Azure Service Bus Queue**  
✔ **Payment Processor Service** picks up the message and processes the payment  
✔ After successful payment, **Payment Processor Service** sends a **confirmation message** back to Order Service

💡 **Why Azure Service Bus?**

* **FIFO Guarantee** ensures payments are processed **sequentially**
* **Dead-Letter Queue** ensures failed transactions are retried or logged
* **Secure and Decoupled Communication** between Order & Payment Services

**🚀 Step 3: Event-Driven Processing (Inventory & Notifications)**

**3️⃣ Order Completion Event (Using Azure Event Grid)**

✔ Once **payment is successful**, the **Order Service** publishes an OrderCompleted event to **Azure Event Grid**  
✔ Event Grid fans out the event to multiple subscribers:

* **Inventory Service** 🏪 → Deducts stock
* **Notification Service** 📩 → Sends an email/SMS confirmation
* **Shipping Service** 🚚 → Generates tracking details

💡 **Why Azure Event Grid?**

* **Low Latency**: Push-based eventing
* **Fan-Out**: **Multiple consumers can react to the event**
* **Highly Scalable**: No need to predefine consumers

**🚀 Step 4: Telematics & Observability**

**4️⃣ Event Hub for Event Monitoring & Analytics**

✔ **All important events (OrderCreated, PaymentProcessed, OrderCompleted)** are **streamed to Azure Event Hub**  
✔ **Event Hub stores telemetry data in Azure Log Analytics** for:

* **Monitoring and Debugging** 🔍
* **Audit Trail (Order ID, Correlation ID, Timestamp, Event Status, Failure Reasons)**
* **Business Intelligence (KPIs on payment success rates, inventory updates, etc.)**

💡 **Why Azure Event Hub?**

* **High-throughput event ingestion** (supports millions of events per second)
* **Real-time monitoring** via Log Analytics
* **Kafka-Compatible** (if future adoption needed)

**📌 Final Architecture Diagram**

User 🛒

|

[Cart API] ---> [Order Service]

|

|----> [Azure Service Bus] (Ensures Reliable Payment Processing)

| |

| [Payment Service]

|

|----> [Azure Event Grid] (For Order Completed Event)

|

--------------------------------------

| | |

[Inventory Service] [Notification Service] [Shipping Service]

(Updates Stock) (Sends Email) (Generates Tracking)

|

[Azure Event Hub] (Telemetry for Monitoring)

|

[Azure Log Analytics] (Dashboards & Insights)

**✅ Summary: When to Use What?**

| **Component** | **Technology** | **Why?** |
| --- | --- | --- |
| **Product & Cart** | **REST API (DDD-Based)** | CRUD & no async needed |
| **Order to Payment** | **Azure Service Bus** | FIFO, Reliable, Transactional |
| **Payment Success → Next Steps** | **Azure Event Grid** | Low latency, Fan-Out |
| **Telemetry & Monitoring** | **Azure Event Hub + Log Analytics** | High-throughput event ingestion |

**🎯 Final Thought: Is This the Right Approach?**

✅ **Yes, this is the industry best practice for modern e-commerce solutions.**  
🚀 **Next Step:** Do you need help with **hands-on implementation** (Azure setup + .NET code)?

|  |
| --- |
| Order Service (REST API) - Sends Order to Service Bus Queue for Payment Processing  using Azure.Messaging.ServiceBus;  using Microsoft.AspNetCore.Mvc;  using System.Text.Json;  using System.Threading.Tasks;  [ApiController]  [Route("api/orders")]  public class OrderController : ControllerBase  {  private readonly string serviceBusConnection = "YOUR\_SERVICE\_BUS\_CONNECTION";  private readonly string queueName = "payment-queue";  [HttpPost]  public async Task<IActionResult> CreateOrder([FromBody] Order order)  {  await SendToServiceBus(order);  return Ok(new { message = "Order Placed. Awaiting Payment Processing." });  }  private async Task SendToServiceBus(Order order)  {  await using var client = new ServiceBusClient(serviceBusConnection);  ServiceBusSender sender = client.CreateSender(queueName);  string orderJson = JsonSerializer.Serialize(order);  await sender.SendMessageAsync(new ServiceBusMessage(orderJson));  }  }  public class Order { public string OrderId { get; set; } public double Amount { get; set; } } |
| 2️⃣ Payment Service (Consumes from Service Bus, Triggers Event Grid) - Processes Payment & Publishes Order Completion Event  using Azure.Messaging.ServiceBus;  using Azure.Messaging.EventGrid;  using System;  using System.Text.Json;  using System.Threading.Tasks;  class PaymentProcessor  {  private static string serviceBusConnection = "YOUR\_SERVICE\_BUS\_CONNECTION";  private static string queueName = "payment-queue";  private static string eventGridEndpoint = "YOUR\_EVENT\_GRID\_ENDPOINT";  private static string eventGridKey = "YOUR\_EVENT\_GRID\_KEY";  static async Task Main()  {  await using var client = new ServiceBusClient(serviceBusConnection);  ServiceBusProcessor processor = client.CreateProcessor(queueName);  processor.ProcessMessageAsync += async args =>  {  string orderJson = args.Message.Body.ToString();  var order = JsonSerializer.Deserialize<Order>(orderJson);  Console.WriteLine($"Processing Payment for Order: {order.OrderId}");  // Simulate payment success  await Task.Delay(2000);  // Publish event to Event Grid  var client = new EventGridPublisherClient(new Uri(eventGridEndpoint), new AzureKeyCredential(eventGridKey));  await client.SendEventAsync(new EventGridEvent(order.OrderId, "Order.Completed", "1.0", order));  Console.WriteLine($"Payment Processed, Order {order.OrderId} Completed.");  await args.CompleteMessageAsync(args.Message);  };  await processor.StartProcessingAsync();  Console.WriteLine("Listening for payments...");  Console.ReadLine();  await processor.StopProcessingAsync();  }  } |
| **3️⃣ Inventory & Notification Services (Event Grid Subscribers)**  📌 **Receives Order.Completed Event & Updates Inventory**  **Updates Stock when Event Grid Publishes Order Completed Event**  using Azure.Messaging.EventGrid;  using Microsoft.AspNetCore.Mvc;  using System.Text.Json;  [ApiController]  [Route("api/inventory")]  public class InventoryController : ControllerBase  {  [HttpPost("eventgrid")]  public IActionResult UpdateStock([FromBody] EventGridEvent eventGridEvent)  {  var order = JsonSerializer.Deserialize<Order>(eventGridEvent.Data.ToString());  Console.WriteLine($"Inventory Updated for Order: {order.OrderId}");  return Ok();  }  } |
| **4️⃣ Telemetry (Sends Data to Event Hub)**  📌 **Stores Events in Event Hub for Analytics**  using Azure.Messaging.EventHubs;  using Azure.Messaging.EventHubs.Producer;  using System;  using System.Text;  using System.Threading.Tasks;  class TelemetryLogger  {  private static string eventHubConnection = "YOUR\_EVENT\_HUB\_CONNECTION";  private static string eventHubName = "telemetry";  static async Task Main()  {  var producerClient = new EventHubProducerClient(eventHubConnection, eventHubName);  using EventDataBatch eventBatch = await producerClient.CreateBatchAsync();  eventBatch.TryAdd(new EventData(Encoding.UTF8.GetBytes("{ \"orderId\": \"12345\", \"status\": \"Completed\" }")));  await producerClient.SendAsync(eventBatch);  Console.WriteLine("Telemetry Sent.");  }  } |
| Testing  curl -X POST "http://localhost:5000/api/orders" -H "Content-Type: application/json" -d '{ "orderId": "12345", "amount": 199.99 }' |
| ✅ **Check Payment Processing Logs** ✅ **Check Event Grid Delivery (Inventory & Notification Logs)** ✅ **View Telemetry in Azure Log Analytics** |

A diagram of a function

AI-generated content may be incorrect.

**🔹 Understanding the Architectural Split**

✅ **Microservices Architecture with Domain-Driven Design (DDD)**

* Services like **Product API and Cart API** follow a **request-response pattern** (RESTful or gRPC).
* These services **don’t rely on event-driven communication** because they involve real-time user interactions.
* **Example:**
  + **Product API** → Fetches product details (GET /products/{id})
  + **Cart API** → Manages cart operations (POST /cart, DELETE /cart/item)

✅ **Event-Driven Architecture (EDA)**

* Once the user places an order, the system shifts to **asynchronous processing**.
* **Order, Payment, Inventory, Notification services** communicate via **events**, ensuring **scalability, decoupling, and reliability**.
* **Example:**
  + **Order Service** → Publishes "OrderPlaced" event to Azure Event Grid.
  + **Payment Service** → Listens to "OrderPlaced" event and processes payment asynchronously.

**📌 How They Work Together in an E-Commerce System**

1. **Product API Service (Microservice - DDD)**
   * Fetches product details when a user browses.
   * Not event-driven, as it follows **real-time API calls**.
2. **Cart API Service (Microservice - DDD)**
   * Manages the shopping cart, adding/removing items.
   * Also not event-driven, as it requires **instant responses**.
3. **Order Service (Event-Driven)**
   * Once a user places an order, it **publishes an event** (OrderPlaced).
   * Other services consume this event asynchronously.
4. **Payment, Inventory, Notification Services (Event-Driven)**
   * **Listen** for OrderPlaced events and act accordingly.
   * Example: Payment Service listens → If successful, publishes PaymentSuccess event.

**🚀 Summary (When to Use What?)**

| **Component** | **Architecture** | **Pattern** |
| --- | --- | --- |
| **Product API** | Microservices | Request-Response (DDD) |
| **Cart API** | Microservices | Request-Response (DDD) |
| **Order Service** | Event-Driven | Publishes OrderPlaced event |
| **Payment Service** | Event-Driven | Consumes OrderPlaced, publishes PaymentSuccess |
| **Inventory Service** | Event-Driven | Consumes OrderCompleted, updates stock |
| **Notification Service** | Event-Driven | Consumes OrderCompleted, sends notifications |

**📌 Key Takeaways**

* **Product & Cart APIs** → **Microservices using DDD**, work in real-time (Request-Response).
* **Order, Payment, Inventory, Notification** → **Event-Driven**, work asynchronously (Event-Based Communication).
* **Once an order is placed, everything shifts from Microservices to Event-Driven Architecture.**

Would you like an updated architecture diagram with this clear distinction?

**📌 Day 1 – Multi-Region Active-Active Architecture in Azure**

**✅ Learn Active-Active vs. Active-Passive design patterns  
✅ Explore Azure Front Door, Azure Traffic Manager, and Cosmos DB multi-region setup  
✅ Hands-on: Deploy a simple multi-region web app using Azure Front Door**

**📌 Outcome: Understanding of high availability & disaster recovery strategies**

**📌 Day 2 – Data Strategy: Sharding vs. Replication in Azure SQL & Cosmos DB**

**✅ Learn when to use sharding vs. replication  
✅ Compare Azure SQL Hyperscale vs. Cosmos DB partitioning  
✅ Hands-on: Implement a simple sharded database in Azure SQL**

**📌 Outcome: Stronger data architecture skills for scalable solutions**

**📌 Day 3 – Enterprise API Security & Gateway Design**

**✅ Learn OAuth 2.0, OpenID Connect, API Keys  
✅ Explore Azure API Management (APIM) for security  
✅ Hands-on: Secure an API using Azure AD Authentication**

**📌 Outcome: Ability to design secure APIs for enterprise applications**

**📌 Day 4 – Cloud Migration Strategy for .NET Apps**

**✅ Study Lift-and-Shift vs. Replatforming vs. Refactoring  
✅ Explore Azure App Service, AKS (Kubernetes), and Azure Functions  
✅ Hands-on: Migrate a simple .NET app to Azure App Service**

**📌 Outcome: Understanding of modernization approaches for enterprises**