Awesome brief. Here’s a **deep, practical reference** on key microservices design patterns—each with: what it is, when to use, **sub-types**, an **end-to-end C#/.NET example snippet**, common pitfalls, and **5 interview/exam questions with answers**. I’ve kept the domain consistent (an **Order → Payment → Inventory** flow) so snippets “compose” in your head.

# 1) Service Decomposition Pattern

**Goal:** Split a monolith into independently deployable services.

**Sub-types**

* **By Business Capability** (Product/Order/Payment/Inventory/Shipping)
* **By DDD Subdomain** (Core vs Supporting vs Generic); bounded contexts with anti-corruption layers

**When to use**

* Clear domain boundaries; teams map to services; independent release cadence.

**Example (DDD-style boundaries)**

// Domain contracts (shared only as contracts, not implementation)

public record OrderId(Guid Value);

public enum OrderStatus { Created, Paid, Cancelled, Shipped }

public interface IOrdersApi

{

Task<OrderDto> GetAsync(OrderId id);

Task<OrderId> CreateAsync(CreateOrderDto dto);

Task MarkPaidAsync(OrderId id);

}

// Order Service boundary – no direct DB coupling to others

public sealed class Order

{

public OrderId Id { get; private set; }

public OrderStatus Status { get; private set; }

public decimal Total { get; private set; }

public static Order Create(decimal total) => new()

{

Id = new OrderId(Guid.NewGuid()),

Status = OrderStatus.Created,

Total = total

};

public void MarkPaid()

{

if (Status != OrderStatus.Created) throw new InvalidOperationException("Not payable.");

Status = OrderStatus.Paid;

}

}

**Pitfalls**

* Nano-services (too fine-grained), chatty RPC, accidental shared DB.

**5 Q&A**

1. **Q:** Business capability vs subdomain? **A:** Capability focuses on “what the business does”; subdomain on DDD modeling; often align but not always.
2. **Q:** Signs boundaries are wrong? **A:** Excess cross-service chatter, cyclic dependencies, frequent cross-team commits.
3. **Q:** How to evolve boundaries? **A:** Use event logs + interaction heatmaps → split/merge; add ACLs.
4. **Q:** Shared libraries risk? **A:** Tight coupling via implicit contracts; prefer slim, versioned interfaces.
5. **Q:** How to test boundaries? **A:** Consumer-driven contract tests (Pact), plus chaos tests on interactions.

# 2) Database per Service Pattern

**Goal:** Each service owns its data store.

**Sub-types**

* Polyglot persistence: SQL for Orders, Document DB for Catalog, KV cache for Inventory.

**When to use**

* Independent scaling & schema evolution; autonomy; avoids lockstep releases.

**Example (EF Core in Order Service)**

public class OrderDbContext : DbContext

{

public DbSet<Order> Orders => Set<Order>();

public OrderDbContext(DbContextOptions<OrderDbContext> options) : base(options) { }

protected override void OnModelCreating(ModelBuilder b)

{

b.Entity<Order>(e =>

{

e.HasKey(x => x.Id);

e.Property(x => x.Status).HasConversion<string>();

});

}

}

**Pitfalls**

* Reporting across services; distributed transactions (avoid 2PC), duplication and eventual consistency concerns.

**5 Q&A**

1. **Q:** Why avoid shared DB? **A:** Coupling + unsafe schema changes break others.
2. **Q:** How to do cross-service reporting? **A:** Event duplication into a read model or data lake.
3. **Q:** Handling joins across services? **A:** Async composition (API aggregator) or precomputed projections.
4. **Q:** Migration strategy? **A:** Strangler + dual-write via outbox until consumers cut over.
5. **Q:** Cache ownership? **A:** Owned by the same service; others don’t mutate it.

# 3) Saga Pattern (Distributed Transactions)

**Goal:** Maintain **eventual consistency** across services using local transactions + compensations.

**Sub-types**

* **Orchestration** (central brain coordinates)
* **Choreography** (events drive peers; no central coordinator)

**When to use**

* Multi-step business process (Pay → Reserve Stock → Create Shipment).

**Orchestration Example (C# coordinator)**

public record StartOrderSaga(Guid OrderId, decimal Amount);

public record PaymentApproved(Guid OrderId);

public record StockReserved(Guid OrderId);

public record SagaFailed(Guid OrderId, string Reason);

public interface IMessageBus

{

Task PublishAsync<T>(T msg);

Task SendAsync<T>(string endpoint, T cmd);

}

public sealed class OrderSaga

{

private readonly IMessageBus \_bus;

private readonly IOrderRepository \_repo;

public OrderSaga(IMessageBus bus, IOrderRepository repo)

{ \_bus = bus; \_repo = repo; }

public async Task Handle(StartOrderSaga msg)

{

await \_bus.SendAsync("payment", new AuthorizePayment(msg.OrderId, msg.Amount));

}

public async Task Handle(PaymentApproved evt)

{

await \_bus.SendAsync("inventory", new ReserveStock(evt.OrderId));

}

public async Task Handle(StockReserved evt)

{

var order = await \_repo.Get(evt.OrderId);

order.MarkPaid();

await \_repo.Save(order);

// Emit OrderPaid event for downstream (shipping)

await \_bus.PublishAsync(new OrderPaid(evt.OrderId));

}

public async Task Compensate(Guid orderId, string reason)

{

await \_bus.SendAsync("payment", new RefundPayment(orderId));

await \_bus.SendAsync("inventory", new ReleaseStock(orderId));

await \_bus.PublishAsync(new SagaFailed(orderId, reason));

}

}

**Choreography Sketch**

* Payment publishes PaymentApproved; Inventory subscribes and reserves; on failure services publish compensating events.

**Pitfalls**

* Orchestrator can centralize too much logic; choreography can drift into “event spaghetti”; ensure idempotency.

**5 Q&A**

1. **Q:** Orchestration vs choreography tradeoff? **A:** Orchestration = clarity/observability; Choreography = autonomy/scalability.
2. **Q:** How to handle partial failures? **A:** Compensating actions per step.
3. **Q:** Idempotency in saga? **A:** Store processed message IDs; upserts; dedupe keys.
4. **Q:** Observability for sagas? **A:** Correlation IDs + distributed tracing spans per step.
5. **Q:** When not to use? **A:** If strong consistency is mandatory per user action → consider redesign (or single service).

# 4) CQRS (Command–Query Responsibility Segregation)

**Goal:** Separate **writes (commands)** from **reads (queries)** for scalability and UX.

**Sub-types**

* **Simple CQRS** (same DB, separate models)
* **Full CQRS** (separate read store populated by events/handlers)

**When to use**

* Heavy read load, complex view models, or eventual consistency acceptable.

**Example (Full CQRS: write → outbox → read projection)**

// Command handler (write)

public sealed class CreateOrderHandler

{

private readonly OrderDbContext \_db;

private readonly IOutbox \_outbox;

public CreateOrderHandler(OrderDbContext db, IOutbox outbox) { \_db = db; \_outbox = outbox; }

public async Task<OrderId> Handle(CreateOrderDto dto)

{

var order = Order.Create(dto.Total);

await \_db.AddAsync(order);

await \_outbox.AddAsync(new OrderCreated(order.Id.Value, order.Total));

await \_db.SaveChangesAsync(); // same tx w/ outbox

return order.Id;

}

}

// Projection (read model)

public sealed class OrderProjectionHandler

{

private readonly ReadDbContext \_read;

public OrderProjectionHandler(ReadDbContext read) { \_read = read; }

public async Task Handle(OrderCreated evt)

{

\_read.OrdersRead.Add(new OrderRead

{

OrderId = evt.OrderId,

Total = evt.Total,

Status = "Created"

});

await \_read.SaveChangesAsync();

}

}

**Pitfalls**

* Keeping projections up to date; replay speed; dual-write bugs (solve with outbox).

**5 Q&A**

1. **Q:** Does CQRS mandate event sourcing? **A:** No; they’re orthogonal.
2. **Q:** Eventual consistency UX? **A:** Show pending status; push updates via SignalR/webhooks.
3. **Q:** How to rebuild read models? **A:** Replay events; version handlers.
4. **Q:** Backpressure on projections? **A:** Partitioned consumers; retry + DLQ.
5. **Q:** When to avoid CQRS? **A:** Simple CRUD; when read/write asymmetry is low.

# 5) Outbox/Inbox (Exactly-Once Effects)

**Goal:** Prevent “lost messages” and double-processing with at-least-once brokers.

**Sub-types**

* **Transactional Outbox** (in producer)
* **Inbox/Dedup Store** (in consumer)
* **Outbox Poller** (background publisher)

**When to use**

* You write to DB and publish an event **atomically**.

**Outbox (producer)**

public record OutboxMessage(Guid Id, string Type, string Payload, DateTime CreatedUtc);

public interface IOutbox

{

Task AddAsync<T>(T evt);

}

public sealed class EfOutbox : IOutbox

{

private readonly OrderDbContext \_db;

public EfOutbox(OrderDbContext db) { \_db = db; }

public Task AddAsync<T>(T evt)

{

var msg = new OutboxMessage(Guid.NewGuid(), typeof(T).FullName!,

JsonSerializer.Serialize(evt), DateTime.UtcNow);

\_db.Set<OutboxMessage>().Add(msg); // in same transaction as domain writes

return Task.CompletedTask;

}

}

// Publisher background service

public sealed class OutboxPublisher : BackgroundService

{

private readonly OrderDbContext \_db;

private readonly IMessageBus \_bus;

public OutboxPublisher(OrderDbContext db, IMessageBus bus)

{ \_db = db; \_bus = bus; }

protected override async Task ExecuteAsync(CancellationToken ct)

{

while (!ct.IsCancellationRequested)

{

var pending = await \_db.Set<OutboxMessage>()

.OrderBy(x => x.CreatedUtc).Take(100).ToListAsync(ct);

foreach (var msg in pending)

{

await \_bus.PublishAsyncRaw(msg.Type, msg.Payload);

\_db.Remove(msg); // delete AFTER successful publish

}

await \_db.SaveChangesAsync(ct);

await Task.Delay(TimeSpan.FromSeconds(1), ct);

}

}

}

**Inbox (consumer)**

public sealed class InboxConsumer<T>

{

private readonly ConsumerDbContext \_db;

public InboxConsumer(ConsumerDbContext db) { \_db = db; }

public async Task HandleAsync(string messageId, T evt, Func<Task> handler)

{

if (await \_db.Inbox.AnyAsync(x => x.MessageId == messageId)) return; // dedupe

await handler();

\_db.Inbox.Add(new InboxRecord { MessageId = messageId, ReceivedAtUtc = DateTime.UtcNow });

await \_db.SaveChangesAsync();

}

}

**Pitfalls**

* Outbox table growth (needs TTL/archival); ordering guarantees; poison events.

**5 Q&A**

1. **Q:** Why not rely on broker exactly-once? **A:** Portability/perf; DB+broker XA is brittle; outbox is simpler.
2. **Q:** How to preserve ordering? **A:** Partition keys and sequence numbers.
3. **Q:** How to monitor? **A:** “Age of oldest outbox message” SLI; DLQ metrics.
4. **Q:** Idempotency and retries? **A:** Inbox dedupe + idempotent handlers.
5. **Q:** Schema evolution in payloads? **A:** Versioned event types; tolerant readers.

# 6) API Gateway Pattern

**Goal:** Single entry for clients; cross-cutting concerns centralized.

**Sub-types**

* **Gateway per Service**
* **Aggregator** (compose multiple backend calls)
* **BFF (Backend-for-Frontend)** (different gateways per UI surface)

**When to use**

* Policy, auth, rate limits, CORS, aggregation, zero-trust edges.

**Example (YARP minimal config + aggregator endpoint)**

// appsettings.json for YARP inside Gateway

{

"ReverseProxy": {

"Routes": [

{ "RouteId": "orders", "ClusterId": "orders", "Match": { "Path": "/orders/{\*\*catch-all}" } },

{ "RouteId": "payments", "ClusterId": "payments", "Match": { "Path": "/payments/{\*\*catch-all}" } }

],

"Clusters": {

"orders": { "Destinations": { "d1": { "Address": "http://orders-svc" } } },

"payments": { "Destinations": { "d1": { "Address": "http://payments-svc" } } }

}

}

}

// Aggregator endpoint (in Gateway)

app.MapGet("/order-summary/{id}", async (Guid id, IHttpClientFactory f) =>

{

var client = f.CreateClient();

var order = await client.GetFromJsonAsync<OrderDto>($"http://orders-svc/orders/{id}");

var payment = await client.GetFromJsonAsync<PaymentDto>($"http://payments-svc/payments/by-order/{id}");

return new { order, payment };

});

**Pitfalls**

* Over-fat gateway (becoming a monolith); tight coupling to UI; latency when aggregating many calls.

**5 Q&A**

1. **Q:** BFF vs single gateway? **A:** BFF tailors endpoints per client; reduces payload/round-trips.
2. **Q:** Where to put CORS? **A:** At gateway edge; lock down origins per app.
3. **Q:** Canary/blue-green? **A:** Gateway routes by header/cookie % splits.
4. **Q:** AuthN/Z? **A:** OIDC at gateway; propagate JWT to services (audience scoping).
5. **Q:** Request collapse? **A:** Cache + coalescing at gateway for hot reads.

# 7) Resilience Patterns (Polly)

**Goal:** Stay reliable under failure.

**Sub-types**

* **Timeout**, **Retry with jitter**, **Circuit Breaker**, **Bulkhead/Queue isolation**, **Fallback**, **Idempotency Keys**

**When to use**

* Any inter-service call or external dependency.

**Example (Polly + HttpClient)**

services.AddHttpClient("inventory")

.AddTransientHttpErrorPolicy(p => p.WaitAndRetryAsync(

3, i => TimeSpan.FromMilliseconds(100 \* Math.Pow(2, i)) + TimeSpan.FromMilliseconds(Random.Shared.Next(0, 50))))

.AddTransientHttpErrorPolicy(p => p.CircuitBreakerAsync(

handledEventsAllowedBeforeBreaking: 5,

durationOfBreak: TimeSpan.FromSeconds(30)))

.AddPolicyHandler(Policy.TimeoutAsync<HttpResponseMessage>(TimeSpan.FromSeconds(2)));

**Idempotency for POST**

app.MapPost("/payments", async (HttpRequest req, PaymentDbContext db) =>

{

var key = req.Headers["Idempotency-Key"].ToString();

if (await db.Idempotency.AnyAsync(x => x.Key == key)) return Results.StatusCode(409);

// process payment...

db.Idempotency.Add(new IdempotencyRecord { Key = key, CreatedUtc = DateTime.UtcNow });

await db.SaveChangesAsync();

return Results.Accepted();

});

**Pitfalls**

* Aggressive retries amplify load; long timeouts pin threads; breakers that never close (no probes).

**5 Q&A**

1. **Q:** Retry vs circuit breaker? **A:** Retry for transient faults; breaker for persistent failures.
2. **Q:** Why jitter? **A:** Avoid thundering herds during backoff.
3. **Q:** Timeout placement? **A:** At the **call site**; bound end-to-end latency.
4. **Q:** Bulkhead? **A:** Isolate thread pools/queues per dependency to stop cascading failures.
5. **Q:** Idempotency scope? **A:** Per resource/action; expire keys reasonably.

# 8) Event-Driven Communication

**Goal:** Loose coupling via async events.

**Sub-types**

* **Pub/Sub (notification events)**
* **Event-Carried State Transfer (ECST)** (event contains enough data for local view)
* **Event Sourcing** (source of truth is the event log; state = replay)

**When to use**

* Many subscribers, auditability, reactive projections.

**Pub/Sub Example (contract + handler)**

public record OrderPaid(Guid OrderId, decimal Amount, DateTime PaidAtUtc);

public sealed class InventoryOnOrderPaid

{

private readonly InventoryDbContext \_db;

public InventoryOnOrderPaid(InventoryDbContext db) { \_db = db; }

public async Task Handle(OrderPaid evt)

{

var reserved = await \_db.Reservations.SingleOrDefaultAsync(x => x.OrderId == evt.OrderId);

if (reserved is null) { /\* create reservation \*/ }

// update stock, emit further events as needed

}

}

**Event Sourcing Sketch (Aggregate + Append)**

public interface IEvent { Guid StreamId { get; } long Version { get; set; } }

public sealed class OrderAggregate

{

private readonly List<IEvent> \_changes = new();

public Guid Id { get; private set; }

public OrderStatus Status { get; private set; }

public static OrderAggregate Create(Guid id, decimal total)

{

var a = new OrderAggregate();

a.ApplyChange(new OrderCreatedEvent(id, total));

return a;

}

public void MarkPaid() => ApplyChange(new OrderPaidEvent(Id));

public void LoadFromHistory(IEnumerable<IEvent> history)

{

foreach (var e in history) Apply(e);

}

private void ApplyChange(IEvent e) { Apply(e); \_changes.Add(e); }

private void Apply(IEvent e)

{

switch (e)

{

case OrderCreatedEvent oc: Id = oc.StreamId; Status = OrderStatus.Created; break;

case OrderPaidEvent op: Status = OrderStatus.Paid; break;

}

}

public IEnumerable<IEvent> GetUncommittedChanges() => \_changes;

}

**Pitfalls**

* Versioning events; consumer drift; exactly-once illusions (use outbox/inbox).

**5 Q&A**

1. **Q:** ECST vs notification event? **A:** ECST carries full state needed for local cache; notification conveys “something happened”.
2. **Q:** Event ordering? **A:** Partition by key; sequence numbers; consumers enforce monotonicity.
3. **Q:** Schema evolution? **A:** New event types, defaults, upcasters for old events.
4. **Q:** Event sourcing pros/cons? **A:** +Audit/replay; −Complexity, projections, migration cost.
5. **Q:** Testing events? **A:** Given-When-Then on aggregates; contract tests on event schemas.

# 9) Service Discovery

**Goal:** Locate services dynamically in elastic environments.

**Sub-types**

* **Client-side discovery** (client queries registry like Consul/Eureka)
* **Server-side discovery** (ingress/load balancer resolves)
* **Sidecar/Service mesh** (Envoy/Istio)

**Example (Typed HttpClient + DNS/K8s)**

services.AddHttpClient<IInventoryClient, InventoryClient>(client =>

{

client.BaseAddress = new Uri("http://inventory-svc.default.svc.cluster.local"); // K8s DNS

});

**Pitfalls**

* Stale caches; endpoint health; partition/zone awareness.

**5 Q&A**

1. **Q:** When choose mesh? **A:** For mTLS, retries, traffic shaping without app changes.
2. **Q:** Health checks? **A:** Liveness/readiness + outlier detection in LB.
3. **Q:** Zone-aware routing? **A:** Prefer same AZ to cut latency.
4. **Q:** Blue/green with discovery? **A:** Register v2 and shift weights gradually.
5. **Q:** Failure modes? **A:** Registry down → cached endpoints + exponential refresh.

# 10) Observability (Logs, Metrics, Traces)

**Goal:** Understand and debug distributed flows.

**Sub-types**

* **Structured logging**, **RED/USE metrics**, **Distributed tracing** (W3C tracecontext)

**When to use**

* Always—especially sagas and async flows.

**Example (.NET Activity & Correlation)**

app.Use(async (ctx, next) =>

{

using var activity = new System.Diagnostics.Activity("HTTP " + ctx.Request.Path);

activity.SetIdFormat(ActivityIdFormat.W3C);

activity.Start();

activity.AddTag("user", ctx.User?.Identity?.Name ?? "anon");

await next();

activity.SetStatusCode(ctx.Response.StatusCode);

activity.Stop();

});

**Pitfalls**

* Cardinality explosions in metrics; missing cross-service correlation IDs.

**5 Q&A**

1. **Q:** Key RED metrics? **A:** Rate, Errors, Duration per endpoint.
2. **Q:** Trace sampling? **A:** Head/tail sampling to control cost; keep errors.
3. **Q:** Log structure? **A:** JSON with correlation/tenant/request IDs.
4. **Q:** What to trace in sagas? **A:** Steps, retries, compensations with a shared trace ID.
5. **Q:** Alerting? **A:** SLO-based alerts on user-visible latency/error budgets.

## Putting It Together (Minimal Flow)

1. **Order Service** writes order + **Outbox** OrderCreated.
2. **OutboxPublisher** publishes to broker.
3. **Saga** orchestrator triggers **Payment** then **Inventory**.
4. **CQRS projection** updates **OrderRead** store.
5. **API Gateway** exposes GET /order-summary/{id}.
6. **Resilience** on all HTTP calls; **Observability** traces the whole thing.

## Tips for Production-Ready Code

* Prefer **async messaging** for inter-service workflows; use **HTTP** for client → gateway and simple lookups.
* Keep **DTOs versioned**; add **compat layers** at the edges.
* Use **Polly** policies per dependency with **timeouts first**, then retries/breakers.
* Secure edges with **OIDC** (JWT audience scoping) and internal **mTLS** (mesh or service-to-service certs).
* Back every cross-service write with **Outbox**; verify with an SLI: “outbox max age < N secs”.

If you want, I can convert this into a **single printable cheat-sheet** or a **.md handbook** with a table of contents and runnable **Minimal APIs** for each service.

### ✅ Microservices Design Patterns: Explanation, Questions, Use Cases, Expected Answers, and C# Code Snippets

## 1. API Gateway Pattern

**Explanation:** Central entry point for all client requests to microservices.

**Use Case:** Client → API Gateway → routes to Product, Order, Auth services.

**C# Code (Ocelot Gateway config example):**

{

"Routes": [

{

"DownstreamPathTemplate": "/api/products",

"UpstreamPathTemplate": "/products",

"DownstreamHostAndPorts": [

{ "Host": "product-service", "Port": 80 }

]

}

]

}

**Questions and Answers:**

1. **What is the role of an API Gateway in microservices?**
   * **A:** It routes requests, handles authentication, throttling, and aggregates responses.
2. **How does API Gateway improve security?**
   * **A:** It enforces centralized authentication and SSL termination.
3. **What are some drawbacks of API Gateway?**
   * **A:** Single point of failure, increased latency, and complexity.
4. **How do you handle failure in downstream services?**
   * **A:** Use fallback, retry policies, and circuit breakers.
5. **When to avoid API Gateway?**
   * **A:** In simple apps or when you don't need request aggregation.

## 2. Circuit Breaker Pattern

**Explanation:** Prevents cascading failure by breaking the circuit when a service fails repeatedly.

**Use Case:** Product service fails → Circuit opens → Gateway returns fallback.

**C# Example (Polly):**

Policy.Handle<Exception>()

.CircuitBreaker(2, TimeSpan.FromMinutes(1));

**Questions and Answers:**

1. **What is the purpose of a circuit breaker?**
   * **A:** To prevent repeated requests to a failing service.
2. **What are the circuit states?**
   * **A:** Closed (normal), Open (fail-fast), Half-open (test mode).
3. **How long does the circuit remain open?**
   * **A:** Until timeout expires or a health check succeeds.
4. **Where do you implement circuit breakers?**
   * **A:** At API Gateway or inside service consumers.
5. **How is this different from retry?**
   * **A:** Retry keeps retrying, circuit breaker gives up early and prevents new calls.

## 3. Saga Pattern

**Explanation:** Manages distributed transactions across microservices using compensating actions.

**Use Case:** Order service → Payment → Inventory → If failure, undo previous steps.

**Choreography C# Example:**

// On OrderCreated event

await paymentService.Charge(orderId);

// If failure → emit OrderCancelled

**Questions and Answers:**

1. **Why use Saga instead of 2PC?**
   * **A:** 2PC doesn’t scale and introduces tight coupling.
2. **What are types of Saga?**
   * **A:** Choreography (event-driven), Orchestration (central coordinator).
3. **When is Saga preferred?**
   * **A:** In long-running or asynchronous business workflows.
4. **How do you implement compensation logic?**
   * **A:** Implement reversal operations for each step (e.g., refund, restock).
5. **What are the challenges of Saga?**
   * **A:** Complex error handling and testing.

## 4. CQRS (Command Query Responsibility Segregation)

**Explanation:** Separates read and write models for scalability and optimization.

**Use Case:** Order commands go through service bus; queries go to read DB.

**C# Example:**

public class CreateOrderCommand : IRequest<bool> {

public string ProductId; public int Quantity;

}

public class GetOrdersQuery : IRequest<List<OrderDto>> {}

**Questions and Answers:**

1. **What are the benefits of CQRS?**
   * **A:** Optimized read/write paths, scalability, better performance.
2. **When not to use CQRS?**
   * **A:** In simple CRUD apps where complexity outweighs benefits.
3. **Can CQRS work without Event Sourcing?**
   * **A:** Yes—Event Sourcing is optional.
4. **What is eventual consistency in CQRS?**
   * **A:** Reads may lag behind writes due to async syncing.
5. **What frameworks support CQRS in .NET?**
   * **A:** MediatR, CAP, NServiceBus.

## 5. Event Sourcing Pattern

**Explanation:** Persist state as a sequence of events instead of current snapshot.

**Use Case:** Bank account records deposits/withdrawals as events.

**C# Example:**

public class AccountEvent {

public string EventType;

public decimal Amount;

public DateTime OccurredOn;

}

**Questions and Answers:**

1. **What are the benefits of event sourcing?**
   * **A:** Full audit trail, time-travel debugging, rehydration.
2. **When to avoid event sourcing?**
   * **A:** When domain is simple or you don't need audit history.
3. **What is the role of snapshots?**
   * **A:** Improve performance by storing state at intervals.
4. **What tools help with event sourcing in .NET?**
   * **A:** Marten, EventStoreDB, NEventStore.
5. **How do you rebuild state from events?**
   * **A:** Replay the event stream to reconstruct the entity.

(The rest of the patterns — including Strangler Fig, Retry, Bulkhead, Backpressure, Service Mesh, Sidecar, Transactional Outbox, Idempotency Token, Service Discovery, DDD, Choreography vs Orchestration, Event Replay, Outbox Polling, API Composition, Aggregator, BFF, Registry with Heartbeat — will follow in the next update...)

Let me know if you'd like me to proceed with appending the remaining patterns right away.

### ✅ Microservices Design Patterns: Explanation, Questions, Use Cases, Expected Answers, and C# Code Snippets

...(previous patterns 1 to 5 here)...

## 6. Strangler Fig Pattern

**Explanation:** Incrementally replace legacy systems by routing some functionality to new services.

**Use Case:** Gradually migrate a monolithic order service to microservices.

**Questions and Answers:**

1. **What is the benefit of the Strangler Fig pattern?**
   * **A:** Allows gradual migration without a big bang rewrite.
2. **How is routing handled in this pattern?**
   * **A:** Use API Gateway or reverse proxy to direct traffic.
3. **What challenges come with this pattern?**
   * **A:** Complexity in maintaining dual systems during transition.
4. **What’s a typical use case?**
   * **A:** Migrating a legacy .NET monolith to modern .NET Core microservices.
5. **How do you know when to deprecate legacy parts?**
   * **A:** When all traffic for a function routes through the new service.

## 7. Retry Pattern

**Explanation:** Automatically retries transient failures like network timeouts.

**C# Polly Example:**

Policy.Handle<TimeoutException>()

.WaitAndRetry(3, retryAttempt => TimeSpan.FromSeconds(retryAttempt));

**Questions and Answers:**

1. **When should you use retry?**
   * **A:** For transient issues like timeouts or temporary unavailability.
2. **What is the risk of retrying?**
   * **A:** It can overwhelm the service and delay failure detection.
3. **What is exponential backoff?**
   * **A:** Increasing wait time between retries to reduce load.
4. **Where is retry best implemented?**
   * **A:** On the client side, service proxy, or middleware.
5. **When should retry be avoided?**
   * **A:** For non-idempotent operations like payments.

## 8. Bulkhead Pattern

**Explanation:** Isolates resources into pools to prevent cascading failures.

**Use Case:** Separate thread pools per microservice call.

**Questions and Answers:**

1. **What problem does the bulkhead pattern solve?**
   * **A:** Prevents failure in one part of the system from affecting others.
2. **What’s a .NET tool to implement it?**
   * **A:** Polly supports isolation policies.
3. **What analogy is used for this pattern?**
   * **A:** Ship compartments preventing full flooding.
4. **When is this pattern useful?**
   * **A:** In services making multiple downstream calls.
5. **What are implementation approaches?**
   * **A:** Thread pools, queues, connection limits.

## 9. Backpressure Pattern

**Explanation:** Controls the flow of data when consumers can't keep up with producers.

**Questions and Answers:**

1. **Why is backpressure important in messaging systems?**
   * **A:** Prevents message queues from being overwhelmed.
2. **What’s an example of applying backpressure?**
   * **A:** Blocking publisher when Kafka or RabbitMQ queue is full.
3. **How does this apply in HTTP?**
   * **A:** Use 429 Too Many Requests response.
4. **What are tools for backpressure in .NET?**
   * **A:** Channels, Semaphores, reactive streams.
5. **What happens without backpressure?**
   * **A:** System crashes or memory overflow.

## 10. Service Mesh Pattern

**Explanation:** A dedicated infrastructure layer for managing service-to-service communication.

**Use Case:** Istio or Linkerd handling retries, TLS, and metrics for Kubernetes services.

**Questions and Answers:**

1. **Why use a service mesh?**
   * **A:** For observability, security, and traffic control.
2. **What does a sidecar proxy do?**
   * **A:** Intercepts service communication for monitoring and control.
3. **What’s the overhead of using a mesh?**
   * **A:** Resource usage, latency, and complexity.
4. **What are popular service mesh tools?**
   * **A:** Istio, Linkerd, Consul Connect.
5. **How is security enhanced?**
   * **A:** Mutual TLS between services.

...

(Next batch will include: Sidecar, Transactional Outbox, Idempotency Token, Service Discovery, DDD, Choreography vs Orchestration, Event Replay, Outbox Polling, API Composition, Aggregator, BFF, Service Registry)

Let me know to continue appending.

### ✅ Microservices Design Patterns: Explanation, Questions, Use Cases, Expected Answers, and C# Code Snippets

...(patterns 1 to 10 here)...

## 11. Sidecar Pattern

**Explanation:** Deploys helper components (e.g., logging, monitoring) alongside service in the same container pod.

**Use Case:** A .NET app with a sidecar for logging using Fluent Bit.

**Questions and Answers:**

1. **Why use a sidecar?**
   * **A:** To offload cross-cutting concerns like logging, monitoring, and proxying.
2. **How is it deployed?**
   * **A:** As a separate container in the same Kubernetes pod.
3. **What are popular use cases?**
   * **A:** Envoy for proxying, Fluent Bit for logs.
4. **What are advantages over shared libraries?**
   * **A:** Language-agnostic, reusable, consistent.
5. **What are the risks?**
   * **A:** Complexity, resource sharing issues.

## 12. Transactional Outbox Pattern

**Explanation:** Ensures reliable event publishing from a local database using outbox table.

**Use Case:** Order service saves order and event in same transaction.

**C# Example:**

using var tx = \_db.Database.BeginTransaction();

\_db.Orders.Add(order);

\_db.OutboxMessages.Add(orderCreatedEvent);

await \_db.SaveChangesAsync();

tx.Commit();

**Questions and Answers:**

1. **Why use outbox instead of direct event publishing?**
   * **A:** To ensure atomicity and avoid message loss.
2. **How are outbox messages published?**
   * **A:** Poller picks them and sends to message broker.
3. **What happens on poller failure?**
   * **A:** Messages remain in DB and are retried.
4. **How to ensure idempotency?**
   * **A:** Use unique message IDs and deduplication logic.
5. **What database changes are needed?**
   * **A:** Outbox table, possibly triggers or timestamp columns.

## 13. Idempotency Token Pattern

**Explanation:** Prevents duplicate processing of the same request.

**Use Case:** Ensure payment isn’t double-processed on retry.

**C# Example:**

if (\_db.RequestLog.Any(x => x.IdempotencyKey == key))

return Conflict("Duplicate");

\_db.RequestLog.Add(new RequestLog { Key = key });

**Questions and Answers:**

1. **Why is idempotency important in APIs?**
   * **A:** To prevent duplicate operations on retries.
2. **Where is the token stored?**
   * **A:** In headers or request body.
3. **What operations need idempotency?**
   * **A:** Payments, account creation, email triggers.
4. **What do you return for repeated requests?**
   * **A:** Same result or conflict response.
5. **How do you store processed tokens?**
   * **A:** In a DB or distributed cache like Redis.

## 14. Service Discovery Pattern

**Explanation:** Enables dynamic location of services without hardcoded addresses.

**Use Case:** Order service finds Shipping service using Consul or Eureka.

**Questions and Answers:**

1. **Why avoid hardcoded service addresses?**
   * **A:** To enable scalability and dynamic environments.
2. **Client-side vs server-side discovery?**
   * **A:** Client handles routing vs load balancer handles it.
3. **Tools for service discovery?**
   * **A:** Consul, Eureka, etcd.
4. **How do services register themselves?**
   * **A:** On startup via API or agent.
5. **What are fallback strategies?**
   * **A:** Static config, retry with exponential backoff.

## 15. Domain-Driven Design (DDD) Patterns

**Explanation:** Aligns microservices with business domains using aggregates, entities, and bounded contexts.

**Use Case:** Patient, Billing, and Appointment services each map to their bounded contexts.

**C# Aggregate Root Example:**

public class Order : AggregateRoot {

public void AddItem(Product product, int quantity) {

// business rule, raise event

}

}

**Questions and Answers:**

1. **What is a bounded context?**
   * **A:** A logical boundary with its own domain model.
2. **What is an aggregate root?**
   * **A:** Main entity that enforces business rules for a group.
3. **When are value objects used?**
   * **A:** When identity isn’t needed (e.g., Address).
4. **What is ubiquitous language?**
   * **A:** Shared terms used by devs and business.
5. **How does DDD help microservice design?**
   * **A:** Defines service boundaries clearly.

## 16. Choreography vs Orchestration

**Explanation:** Choreography = event-driven, decentralized. Orchestration = controlled by a central service.

**Use Case:** OrderPlaced event → Inventory & Payment react (choreography). Orchestrator calls steps in sequence.

**Questions and Answers:**

1. **Which is more loosely coupled?**
   * **A:** Choreography.
2. **When to use orchestration?**
   * **A:** When needing rollback, monitoring.
3. **What tools support orchestration?**
   * **A:** Dapr Workflow, Temporal, Camunda.
4. **Drawbacks of choreography?**
   * **A:** Harder to trace and debug.
5. **Can they coexist?**
   * **A:** Yes. Mix for optimal design.

## 17. Event Replay & Snapshotting Pattern

**Explanation:** In event sourcing, replaying all events builds state; snapshots improve performance.

**Questions and Answers:**

1. **Why use snapshotting?**
   * **A:** Avoid long replay times.
2. **When should you take a snapshot?**
   * **A:** After N events or time-based.
3. **How are snapshots stored?**
   * **A:** As JSON or binary alongside events.
4. **Are snapshots final state?**
   * **A:** No—they are starting points for replay.
5. **What happens if snapshot is lost?**
   * **A:** Replay from beginning or previous snapshot.

## 18. Outbox Polling Publisher Pattern

**Explanation:** Polls DB table for new events to publish to broker.

**Questions and Answers:**

1. **Why is this reliable?**
   * **A:** Events are persisted before publishing.
2. **What triggers the poller?**
   * **A:** Timer, cron job, or hosted service.
3. **What ensures no duplicates?**
   * **A:** Message IDs and published flag.
4. **Where should the poller run?**
   * **A:** Same service or background worker.
5. **How often should polling happen?**
   * **A:** Every few seconds or based on SLA.

## 19. API Composition Pattern

**Explanation:** Aggregates multiple service calls into one response.

**C# Example:**

var user = await userService.GetUser();

var orders = await orderService.GetOrders(user.Id);

return new UserDashboard(user, orders);

**Questions and Answers:**

1. **Why use API composition?**
   * **A:** Avoids client making multiple calls.
2. **Where is it implemented?**
   * **A:** API Gateway or composition layer.
3. **How do you handle failure?**
   * **A:** Return partial results, retries, fallback.
4. **What are performance concerns?**
   * **A:** Parallelism and caching are key.
5. **How to handle pagination?**
   * **A:** Each service paginates independently.

## 20. Aggregator Pattern

**Explanation:** Like API composition but with transformation/orchestration.

**Questions and Answers:**

1. **When is it better than composition?**
   * **A:** When business logic must combine results.
2. **Where should this logic reside?**
   * **A:** In a service/facade layer.
3. **Can this degrade performance?**
   * **A:** Yes, if not parallelized or optimized.
4. **How is data modeled?**
   * **A:** With DTOs combining multiple service outputs.
5. **Is it reusable across frontends?**
   * **A:** Yes, makes frontend simpler.

## 21. Backend for Frontend (BFF)

**Explanation:** API customized for specific frontend (e.g., mobile, web).

**Questions and Answers:**

1. **Why use BFF?**
   * **A:** Tailors API to frontend needs.
2. **Where is BFF hosted?**
   * **A:** As standalone or along with frontend.
3. **Can one BFF serve multiple clients?**
   * **A:** Usually one per UI, but may reuse logic.
4. **How is it secured?**
   * **A:** Use frontend-specific tokens.
5. **What tech is used for BFF in .NET?**
   * **A:** ASP.NET Core Minimal APIs, GraphQL.

## 22. Service Registry with Heartbeat

**Explanation:** Dynamic registry of service instances with health monitoring.

**Questions and Answers:**

1. **What is heartbeat in this context?**
   * **A:** Periodic ping/HTTP call to check health.
2. **Why is this pattern needed?**
   * **A:** For load balancers to detect unhealthy services.
3. **What tools implement it?**
   * **A:** Consul, Eureka.
4. **How to integrate in .NET?**
   * **A:** Register via HTTP, expose /health endpoints.
5. **What if a node fails silently?**
   * **A:** Heartbeat timeout removes it from registry.

🎯 This concludes all 22 core microservices patterns with explanation, use cases, questions, answers, and code. Let me know if you'd like a summary table or export to PDF/Word!