# Mixed-Domain Architect Interview Set (Azure as Default Cloud)

Use this pack to rehearse architect-level interviews across e-commerce, healthcare, finance, and distributed systems. Each item includes a problem, reasoning/trade-offs, C# snippets, and real-world context.

## Section 1: High-Level Design (HLD) – 10 Questions

### HLD-1: Global E‑Commerce Checkout Service with APIM + Payment Orchestration

**Problem:** Design a globally available checkout flow that supports multiple payment providers (Razorpay, Stripe, Adyen), with Azure API Management (APIM), retries, idempotency, and PCI scope minimization.

**Reasoning & Trade-offs:**

* **APIM**: Central entry, throttling, JWT validation (Azure AD B2C), versioning.
* **Orchestrator**: Durable Functions or Azure Kubernetes Service (AKS) + Saga orchestrator. Durable Functions simplify state & retry, but lock you into Azure programming model. AKS adds portability but higher ops overhead.
* **Idempotency**: Use Idempotency-Key header stored in Cosmos DB (partitioned by tenant+orderId). TTL for pending entries.
* **Payments**: Use provider-specific adapters behind an interface; implement circuit breakers (Polly) and exponential backoff.
* **Data**: Orders in Cosmos DB (logical partition by tenantId), payment events in Event Hubs → Synapse for analytics.
* **Security**: Tokenization, store only payment tokens; secrets in Key Vault; private endpoints + VNet integration.

**C# Snippet (Idempotency check at API layer):**

[HttpPost("/checkout")]

public async Task<IActionResult> Checkout([FromHeader(Name="Idempotency-Key")] string key, [FromBody] CheckoutRequest req)

{

if (string.IsNullOrWhiteSpace(key)) return BadRequest("Missing Idempotency-Key");

var lockItem = await \_idemRepo.AcquireAsync(key, req.OrderId);

if (!lockItem.Acquired) return StatusCode(409, "In-Progress or Completed");

try

{

var result = await \_orchestrator.RunAsync(req);

await \_idemRepo.MarkCompletedAsync(key, result.PaymentId);

return Ok(result);

}

catch (Exception ex)

{

await \_idemRepo.MarkFailedAsync(key, ex.Message);

throw;

}

}

**Context:** Used by marketplaces to avoid duplicate charges during flaky mobile networks.

### HLD-2: Prior Authorization (Healthcare) with PHI Segregation

**Problem:** Build a prior-authorization intake & review system that stores PHI, enforces data minimization, and supports audit trails.

**Reasoning & Trade-offs:**

* **Storage**: PHI in Azure SQL (Always Encrypted with CMK in Key Vault) or Cosmos DB with client-side encryption; attachments in Blob Storage with private endpoints.
* **Access**: Attribute-based access control (ABAC) via claims (role, org, jurisdiction). Fine-grained masks at API.
* **Events**: Use Service Bus topics for case state changes; activity logs to Log Analytics/Workspace.
* **Audit**: Append-only event store (Event Hubs Capture → ADLS) + immutable hash chain per case.

**C# Snippet (policy-based masking):**

public string MaskIfNeeded(UserContext user, string fieldName, string value)

{

if (\_policy.CanView(user, fieldName)) return value;

return new string('\*', Math.Min(value?.Length ?? 0, 6));

}

**Context:** HIPAA/GDPR-driven healthcare workflows with strict auditability.

### HLD-3: Real-Time Fraud Detection (Finance) with Streaming

**Problem:** Detect fraud on card transactions within 200ms P99 latency.

**Reasoning & Trade-offs:**

* **Ingress**: Event Hubs (Kafka protocol) for transaction stream.
* **Compute**: Azure Functions (Premium) for low-latency enrichment + Redis for hot features; Azure Stream Analytics or Flink on AKS for sliding windows.
* **Models**: Real-time features in Redis; batch features in Synapse. Model served via Azure ML real-time endpoint.
* **Path**: Synchronous rules + async ML; fallback to rules if endpoint exceeds 80ms SLA.

**C# Snippet (rules then ML with timeout):**

if (ruleEngine.IsHighRisk(tx)) return Decline("RULES");

var pred = await \_mlClient.ScoreAsync(tx).TimeoutAfter(TimeSpan.FromMilliseconds(80));

return pred.IsFraud ? Decline("ML") : Approve();

**Context:** Card networks/fronting gateways where hard real-time SLA is essential.

### HLD-4: Multitenant SaaS with Strong Isolation

**Problem:** Design a multitenant B2B SaaS with tiered isolation: shared DB, schema-per-tenant, and database-per-tenant.

**Reasoning & Trade-offs:**

* **Routing**: TenantResolver reads tenantId from JWT → route to correct connection string.
* **Provisioning**: AAD App Roles + ARM/Bicep to spin per-tenant resources.
* **Billing**: Metering via Event Grid on usage events.
* **Observability**: Add tenantId to all logs/metrics.

**C# Snippet (tenant scoping in EF Core):**

modelBuilder.Entity<Order>().HasQueryFilter(o => o.TenantId == \_tenantProvider.CurrentTenantId);

**Context:** Enterprise SaaS with noisy-neighbor and compliance concerns.

### HLD-5: Content Delivery for Media with DRM

**Problem:** Deliver DRM-protected video globally, with low startup time.

**Reasoning & Trade-offs:**

* **Media Services**: Azure Media Services for packaging (HLS/DASH) + DRM (Widevine/PlayReady/FairPlay).
* **Edge**: Azure Front Door + CDN for geo-caching; token-based URL signing.
* **Startup**: Pre-warm popular titles; segment size tuning.

**C# Snippet (short-lived signed URL generation):**

var token = \_signer.Generate(contentId, TimeSpan.FromMinutes(5), userId);

var url = $"{cdnBase}/{contentId}/manifest.m3u8?token={token}";

**Context:** OTT platforms and internal training portals.

### HLD-6: Event-Driven Inventory with Exactly-Once Semantics (E‑Commerce)

**Problem:** Keep inventory consistent across order placements, returns, and third-party sellers.

**Reasoning & Trade-offs:**

* **Pattern**: Outbox + Inbox with de-duplication IDs.
* **Transport**: Service Bus topics for reliability; Azure Functions for consumers.
* **Consistency**: Reserve stock in the same DB transaction as publishing to outbox.

**C# Snippet (Outbox pattern):**

using var tx = db.Database.BeginTransaction();

// write order + reservation

await db.Outbox.AddAsync(new OutboxMessage(evt));

await db.SaveChangesAsync();

tx.Commit();

**Context:** Prevents overselling under high concurrency and failures.

### HLD-7: Data Lakehouse for Mixed Domains

**Problem:** Build a Lakehouse ingesting e-commerce, healthcare, and finance data with governance.

**Reasoning & Trade-offs:**

* **Storage**: ADLS Gen2; medallion architecture (Bronze → Silver → Gold) with Delta Lake.
* **Ingest**: Data Factory for batch; Event Hubs for streaming.
* **Governance**: Purview for lineage and classification; RBAC/ABAC for PHI/PII.

**C# Snippet (Event serialization contract):**

public record PurchaseEvent(string TenantId, string OrderId, decimal Amount, string Currency, DateTimeOffset Ts);

**Context:** Enterprise analytics and compliance reporting.

### HLD-8: Zero-Downtime Schema Evolution

**Problem:** Evolve schemas without downtime across microservices.

**Reasoning & Trade-offs:**

* **API**: Backward-compatible changes, tolerant readers.
* **Messaging**: Versioned Avro/JSON schemas in Schema Registry; producers add fields with defaults.
* **DB**: Expand/contract pattern with dual-write canaries.

**C# Snippet (tolerant reader):**

var node = JsonNode.Parse(payload)!;

var status = node["status"]?.GetValue<string>() ?? "Unknown";

**Context:** Avoid Friday-night outages during migrations.

### HLD-9: Secure B2C Auth with APIM + SPA + Native Apps

**Problem:** Unify auth for SPA and mobile apps using Azure AD B2C, with API gateway enforcement.

**Reasoning & Trade-offs:**

* PKCE for public clients; confidential client for server-to-server.
* APIM validates tokens and enforces scopes; per-route policies.
* Conditional Access and risk-based MFA.

**C# Snippet (scope check):**

if (!user.HasScope("orders.read")) return Forbid();

**Context:** Consumer-grade apps needing social identity providers.

### HLD-10: Disaster Recovery Strategy (RPO/RTO)

**Problem:** Achieve RPO ≤ 5 minutes and RTO ≤ 1 hour across regions.

**Reasoning & Trade-offs:**

* **Active/Passive**: Front Door health probes; Cosmos DB multi-region write or SQL Geo-DR.
* **State**: Redis geo-replication or cold start with cache warming.
* **Infra as Code**: Bicep/Terraform with blueprints; regular DR drills.

**C# Snippet (feature flag to fail forward):**

if (\_ff.IsEnabled("ForceSecondaryRegion")) \_regionRouter.UseSecondary();

**Context:** Business continuity for regulated industries.

## Section 2: Low-Level Design (LLD) – 10 Questions

### LLD-1: Payment Provider Strategy + Factory

**Problem:** Switch between multiple payment gateways without changing business logic.

**Reasoning & Trade-offs:** Strategy isolates provider behavior; Factory resolves at runtime; DI container composes.

**C# Snippet:**

public interface IPaymentProvider { Task<PayResult> ChargeAsync(PayRequest req, CancellationToken ct); }

public class StripeProvider : IPaymentProvider { /\* ... \*/ }

public class RazorpayProvider : IPaymentProvider { /\* ... \*/ }

public class PaymentFactory

{

private readonly IServiceProvider \_sp;

public PaymentFactory(IServiceProvider sp) => \_sp = sp;

public IPaymentProvider Create(string name) => name switch

{

"stripe" => \_sp.GetRequiredService<StripeProvider>(),

"razorpay" => \_sp.GetRequiredService<RazorpayProvider>(),

\_ => throw new NotSupportedException(name)

};

}

**Context:** Feature flags route traffic during A/B testing of gateways.

### LLD-2: Domain Events with MediatR (DDD)

**Problem:** Decouple aggregate changes from side-effects (email, projections).

**Reasoning:** Aggregate raises events; Unit of Work dispatches after commit.

**C# Snippet:**

public abstract class AggregateRoot

{

private readonly List<INotification> \_events = new();

protected void AddEvent(INotification evt) => \_events.Add(evt);

public IReadOnlyCollection<INotification> Events => \_events;

}

public class Order : AggregateRoot

{

public void Pay(decimal amount) { /\* ... \*/ AddEvent(new OrderPaid(Id, amount)); }

}

**Context:** Clean separation improves testability.

### LLD-3: Outbox/Inbox De-duplication

**Problem:** Guarantee exactly-once effects over at-least-once messaging.

**Reasoning:** Persist event + process marker; idempotent handlers.

**C# Snippet:**

if (await \_inbox.HasProcessedAsync(message.Id)) return;

await \_handler.HandleAsync(message);

await \_inbox.MarkProcessedAsync(message.Id);

**Context:** Inventory and billing systems.

### LLD-4: API Request Validation Pipeline

**Problem:** Centralize validation and return problem+json.

**C# Snippet:**

public class ValidationBehavior<TReq,TRes> : IPipelineBehavior<TReq,TRes>

{

private readonly IEnumerable<IValidator<TReq>> \_validators;

public async Task<TRes> Handle(TReq request, RequestHandlerDelegate<TRes> next, CancellationToken ct)

{

var errors = \_validators.Select(v => v.Validate(request))

.SelectMany(r => r.Errors).Where(e => e != null).ToList();

if (errors.Any()) throw new ValidationException(errors);

return await next();

}

}

**Context:** Avoids scattered validation logic.

### LLD-5: Rate Limiting + Circuit Breaker (Polly)

**Problem:** Protect backend services from overload/downstream failures.

**C# Snippet:**

var breaker = Policy.Handle<HttpRequestException>()

.CircuitBreakerAsync(5, TimeSpan.FromSeconds(30));

var retry = Policy.Handle<Exception>().WaitAndRetryAsync(3, i => TimeSpan.FromMilliseconds(50 \* i));

var policy = Policy.WrapAsync(retry, breaker);

**Context:** Payment and shipping integrations.

### LLD-6: CQRS Read Model Projection

**Problem:** Build fast read models for orders list.

**C# Snippet:**

public class OrderPaidHandler : INotificationHandler<OrderPaid>

{

public async Task Handle(OrderPaid evt, CancellationToken ct)

{

await \_readDb.UpsertAsync(new OrderReadModel{ Id = evt.OrderId, Status = "Paid" });

}

}

**Context:** Dashboard views with low latency.

### LLD-7: PII/PHI Field-Level Encryption

**Problem:** Encrypt specific fields transparently.

**C# Snippet:**

public class EncryptedString

{

public string CipherText { get; private set; }

public static EncryptedString FromPlain(string plain, IEncryptor enc) =>

new() { CipherText = enc.Encrypt(plain) };

public string Decrypt(IEncryptor enc) => enc.Decrypt(CipherText);

}

**Context:** Compliance without encrypting entire records.

### LLD-8: Saga for Order→Payment→Shipment

**Problem:** Implement saga with compensations.

**Reasoning:** Orchestrator sends commands; compensations on failure.

**C# Snippet (pseudo):**

await ReserveInventory();

try { await ChargePayment(); await CreateShipment(); }

catch { await ReleaseInventory(); await RefundPayment(); throw; }

**Context:** Ensures end-to-end consistency.

### LLD-9: Feature Flags + Dark Launch

**Problem:** Gradually enable features per cohort.

**C# Snippet:**

if (\_flags.IsEnabled("NewPricing", user.TenantId))

return \_pricingV2.Calculate(cart);

return \_pricingV1.Calculate(cart);

**Context:** Risk mitigation during rollouts.

### LLD-10: Telemetry Correlation

**Problem:** Correlate logs/traces across services.

**C# Snippet:**

using var op = \_telemetry.StartOperation("Checkout", correlationId);

\_logger.LogInformation("{CorrelationId} Starting checkout", correlationId);

**Context:** Root-cause analysis in distributed systems.

## Section 3: System Design Coding in C# (Non-DSA) – 10 Questions

### Code-1: Idempotent Command Handler

**Problem:** Implement an idempotent handler for CreateOrderCommand with an Idempotency-Key.

**Reasoning:** Prevent duplicate creation under retries.

**C# Solution (essentials):**

public async Task<OrderDto> Handle(CreateOrderCommand cmd)

{

var key = cmd.IdempotencyKey;

if (await \_store.ExistsAsync(key)) return await \_store.GetResultAsync<OrderDto>(key);

using var tx = await \_db.Database.BeginTransactionAsync();

var order = Order.Create(cmd.CustomerId, cmd.Items);

\_db.Orders.Add(order);

await \_db.SaveChangesAsync();

await \_store.SaveResultAsync(key, new OrderDto(order.Id));

await tx.CommitAsync();

return new OrderDto(order.Id);

}

**Context:** API retries from mobile clients.

### Code-2: Outbox Publisher (EF Core)

**Problem:** Add outbox publishing within the same transaction.

**C# Solution:**

public async Task SaveWithOutboxAsync<TEntity>(TEntity entity, IEvent evt)

{

using var tx = await \_db.Database.BeginTransactionAsync();

\_db.Add(entity);

\_db.Outbox.Add(new OutboxMessage(evt));

await \_db.SaveChangesAsync();

await tx.CommitAsync();

}

**Context:** Reliable event-driven communication.

### Code-3: Resilient HTTP Client with Retry + Timeout

**Problem:** Wrap HttpClient calls with timeout and retries.

**C# Solution:**

var timeout = Policy.TimeoutAsync<HttpResponseMessage>(TimeSpan.FromSeconds(2));

var retry = Policy.HandleResult<HttpResponseMessage>(r => !r.IsSuccessStatusCode)

.WaitAndRetryAsync(3, i => TimeSpan.FromMilliseconds(100 \* i));

var policy = Policy.WrapAsync(timeout, retry);

var resp = await policy.ExecuteAsync(() => \_http.SendAsync(req));

**Context:** Unstable partner APIs.

### Code-4: Bulkhead Isolation for CPU-Heavy Work

**Problem:** Limit parallel execution of CPU-intensive operations.

**C# Solution:**

var throttler = new SemaphoreSlim(4);

var tasks = inputs.Select(async x => { await throttler.WaitAsync(); try { await Work(x); } finally { throttler.Release(); } });

await Task.WhenAll(tasks);

**Context:** Prevents thread pool exhaustion in web apps.

### Code-5: Cache-Aside with Redis

**Problem:** Implement read-through cache with fallbacks.

**C# Solution:**

var cacheKey = $"order:{id}";

var cached = await \_cache.GetStringAsync(cacheKey);

if (cached != null) return JsonSerializer.Deserialize<OrderDto>(cached);

var order = await \_repo.GetAsync(id);

await \_cache.SetStringAsync(cacheKey, JsonSerializer.Serialize(order), new DistributedCacheEntryOptions{ AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(5)});

return order;

**Context:** Hot entities like product details.

### Code-6: Streaming Upload to Blob Storage

**Problem:** Upload large files without buffering in memory.

**C# Solution:**

await using var stream = file.OpenReadStream();

await \_blobClient.UploadAsync(stream, overwrite: true);

**Context:** Medical imaging, invoices.

### Code-7: Saga Compensation Sample

**Problem:** Implement compensation logic around a multi-step workflow.

**C# Solution (simplified):**

try

{

await \_inventory.Reserve(orderId);

await \_payment.Charge(orderId);

await \_shipping.Create(orderId);

}

catch

{

await \_shipping.Cancel(orderId);

await \_payment.Refund(orderId);

await \_inventory.Release(orderId);

throw;

}

**Context:** Ensures eventual consistency.

### Code-8: Masking Middleware for Logs

**Problem:** Redact PII/PHI from logs.

**C# Solution:**

public class RedactionMiddleware

{

private static readonly Regex Card = new("\\b(\\d{4}[- ]?){3}\\d{4}\\b", RegexOptions.Compiled);

public async Task Invoke(HttpContext ctx)

{

using var sw = new StringWriter();

var original = ctx.Response.Body; var buffer = new MemoryStream();

ctx.Response.Body = buffer; await \_next(ctx);

buffer.Position = 0; var body = await new StreamReader(buffer).ReadToEndAsync();

body = Card.Replace(body, "\*\*\*\*-\*\*\*\*-\*\*\*\*-\*\*\*\*");

var bytes = Encoding.UTF8.GetBytes(body); await original.WriteAsync(bytes);

}

}

**Context:** Compliance-friendly observability.

### Code-9: Concurrency Control with ETags

**Problem:** Implement optimistic concurrency for updates.

**C# Solution:**

var existing = await \_repo.GetAsync(id);

if (existing.ETag != cmd.ETag) throw new ConcurrencyException();

existing.Apply(cmd);

await \_repo.UpdateAsync(existing);

**Context:** Prevents lost updates in collaborative UIs.

### Code-10: Health Checks + Readiness for AKS

**Problem:** Expose liveness/readiness endpoints.

**C# Solution:**

builder.Services.AddHealthChecks()

.AddSqlServer(connString, name: "sql")

.AddAzureBlobStorage(storageConn, name: "blob");

app.MapHealthChecks("/health/live");

app.MapHealthChecks("/health/ready");

**Context:** Rolling updates with zero downtime.

## Tips for Using This Set

* Tailor each HLD with a **sequence diagram** and **Bicep/Terraform outline** if asked.
* Always discuss **observability (metrics, logs, traces)**, **resilience (retry, timeout, CB)**, **security (AAD/B2C, Key Vault, private endpoints)**, and **cost**.
* When coding, highlight **threading model**, **memory usage**, and **failure modes**.

### Appendix: Quick Azure Components Mapping

* **Gateway:** Azure API Management, Front Door
* **Compute:** AKS, App Service, Functions, Durable Functions
* **Data:** Cosmos DB, Azure SQL, Redis, Blob Storage
* **Messaging/Streaming:** Service Bus, Event Hubs, Event Grid
* **Analytics:** Synapse, Data Factory, Stream Analytics, Purview
* **Security:** Azure AD/B2C, Key Vault, Defender for Cloud
* **Observability:** Application Insights, Log Analytics, OpenTelemetry

Alright — I’ll prepare the **combined MFQ flashcard set** so that each system has:

1. **HLD** – context, architecture breakdown, design decisions, scalability points.
2. **LLD in C#** – class design, key code snippet, explanation.

This way, you can revise **architecture and coding** together for interviews.

## **MFQ – HLD + LLD Combined (20 Questions)**

### **1. Parking Lot System**

**Q:** Design a parking lot system (HLD + LLD).

**HLD:**

* **Context:** Multi-level parking lot for cars, bikes, buses.
* **Components:** UI (kiosk/mobile), ParkingService, PaymentService, ParkingSpotRepository, PostgreSQL DB.
* **Design Decisions:** DDD for entities, REST APIs, vehicle hierarchy via inheritance.
* **Scalability:** Multi-location support via ParkingLot table, distributed parking services.

**LLD in C#:**

public abstract class Vehicle { public string PlateNumber { get; set; } }

public class Car : Vehicle { }

public class ParkingSpot { public int SpotNumber; public bool IsAvailable = true; }

**Reasoning:** Inheritance for vehicle types; composition for spots.

### **2. Movie Ticket Booking**

**Q:** Design an online movie booking system.

**HLD:**

* **Components:** Frontend app, BookingService, ShowService, SeatRepository, Payment Gateway.
* **Design Decisions:** Optimistic locking to avoid seat double-booking.
* **Scalability:** Microservices for search and booking, horizontal scaling.

**LLD in C#:**

public class Seat { public int SeatNumber; public bool IsBooked; public void Book() => IsBooked = true; }

public class Show { public string MovieName; public List<Seat> Seats = new(); }

### **3. Library Management**

**Q:** Design a system to manage library books.

**HLD:**

* **Components:** Staff UI, BookService, MemberService, BookRepository.
* **Design Decisions:** CQRS for query/command separation.
* **Scalability:** Cache for popular book searches.

**LLD in C#:**

public class Book { public string ISBN; public string Title; public bool IsBorrowed; }

public interface IBookRepository { void Add(Book book); Book Get(string isbn); }

### **4. Elevator System**

**HLD:**

* **Components:** ElevatorController, MotorControl API, RequestQueue.
* **Design Decisions:** Shortest-distance-first dispatching.
* **Scalability:** Multi-building central controller.

**LLD:**

public enum ElevatorState { Idle, MovingUp, MovingDown }

public class Elevator { public int CurrentFloor; public ElevatorState State; public void MoveTo(int floor) { /\* logic \*/ } }

### **5. Food Delivery**

**HLD:**

* **Components:** Mobile App, OrderService, RestaurantService, Maps API, Payment.
* **Design Decisions:** Event-driven updates.
* **Scalability:** Microservices per restaurant region.

**LLD:**

public class Order { public int Id; public string Status = "Pending"; }

public class Restaurant { public string Name; public List<Order> Orders = new(); }

### **6. Chess Game**

**HLD:**

* **Components:** Game UI, GameService, Matchmaking, WebSockets for real-time.
* **Design Decisions:** Store moves in FEN.
* **Scalability:** Region-based game servers.

**LLD:**

public abstract class ChessPiece { public string Color; public abstract bool CanMove(int sx, int sy, int ex, int ey); }

public class Knight : ChessPiece { public override bool CanMove(int sx, int sy, int ex, int ey) => ...; }

### **7. ATM**

**HLD:**

* **Components:** ATM UI, AccountService, TransactionService, Core Banking API.
* **Design Decisions:** ACID compliance.
* **Scalability:** Load-balanced backend.

**LLD:**

public class Account { public decimal Balance { get; private set; } public void Withdraw(decimal amt) { /\* validation \*/ } }

### **8. Hotel Booking**

**HLD:**

* **Components:** SearchService, BookingService, Payment, Email.
* **Design Decisions:** Eventual consistency for availability sync.
* **Scalability:** Partition hotel data by location.

**LLD:**

public class Room { public int Number; public string Type; }

public static class RoomFactory { public static Room Create(string type, int num) => new Room { Type = type, Number = num }; }

### **9. Ride-Sharing**

**HLD:**

* **Components:** Passenger/Driver apps, MatchingService, PricingService.
* **Design Decisions:** WebSocket location streaming.
* **Scalability:** Regional clusters.

**LLD:**

public interface IVehicle { string GetDetails(); }

public class Car : IVehicle { public string GetDetails() => "Car ride"; }

### **10. Notification System**

**HLD:**

* **Components:** NotificationService, Channel Handlers, Kafka.
* **Design Decisions:** Async queue processing.
* **Scalability:** Per-channel microservices.

**LLD:**

public interface INotification { void Send(string msg); }

public class EmailNotification : INotification { public void Send(string msg) => Console.WriteLine(msg); }

### **11. Shopping Cart**

**HLD:**

* **Components:** CartService, PricingService, Redis cache.
* **Design Decisions:** In-memory cart storage.
* **Scalability:** Stateless API servers.

**LLD:**

public class CartItem { public decimal Price; public int Qty; }

public class Cart { public List<CartItem> Items = new(); public decimal Total() => Items.Sum(i => i.Price \* i.Qty); }

### **12. Payment Gateway**

**HLD:**

* **Components:** API Gateway, PaymentService, Adapters for each provider.
* **Design Decisions:** Adapter pattern.
* **Scalability:** Multi-threaded transactions.

**LLD:**

public interface IPayment { void Pay(decimal amt); }

public class PayPalAdapter : IPayment { public void Pay(decimal amt) => Console.WriteLine($"Paid: {amt}"); }

### **13. URL Shortener**

**HLD:**

* **Components:** ShortenerService, Key-Value DB.
* **Design Decisions:** Base62 encoding.
* **Scalability:** Shard by prefix.

**LLD:**

public class UrlShortener { private Dictionary<string,string> map = new(); public string Shorten(string url) { string code = Guid.NewGuid().ToString()[..6]; map[code]=url; return code; } }

### **14. Banking System**

**HLD:**

* **Components:** AccountService, TransactionService.
* **Design Decisions:** Transaction isolation.
* **Scalability:** Partition by branch.

**LLD:**

public class Transaction { public decimal Amount; public DateTime Date; }

### **15. File Storage**

**HLD:**

* **Components:** Upload/Download services, Object Storage, Metadata DB.
* **Design Decisions:** Composite Pattern for folders.
* **Scalability:** CDN delivery.

**LLD:**

public interface IFileSystemItem { void Display(); }

public class FileItem : IFileSystemItem { public string Name; public void Display() => Console.WriteLine(Name); }

### **16. Messaging App**

**HLD:**

* **Components:** MessageService, WebSocket server, NoSQL DB.
* **Design Decisions:** Fast writes with NoSQL.
* **Scalability:** Partition chats by user.

**LLD:**

public class MessageDto { public string Sender; public string Receiver; public string Content; }

### **17. Task Scheduler**

**HLD:**

* **Components:** Scheduler, Worker Service, Job DB.
* **Design Decisions:** Delayed queues.
* **Scalability:** Distributed workers.

**LLD:**

public record ScheduledTask(DateTime Time, string Action);

### **18. Quiz App**

**HLD:**

* **Components:** QuizService, ResultService, DB.
* **Design Decisions:** Cache questions.
* **Scalability:** CDN for static content.

**LLD:**

public class Question { public string Text; public string Answer; }

### **19. Workflow Engine**

**HLD:**

* **Components:** Workflow Executor, Step DB, Queue.
* **Design Decisions:** Chain-of-responsibility.
* **Scalability:** Microservice steps.

**LLD:**

public abstract class WorkflowStep { public WorkflowStep Next; public abstract void Execute(); }

### **20. Logger**

**HLD:**

* **Components:** Logger API, Log DB, Analytics Dashboard.
* **Design Decisions:** Async writes.
* **Scalability:** Queue-based ingestion.

**LLD:**

public sealed class Logger { private static readonly Logger instance = new(); private Logger() { } public static Logger Instance => instance; }

If you want, I can now package this into a **ready-to-import MFQ/Anki flashcard file** so that:

* **Front:** “Design XYZ System”
* **Back:** HLD + LLD + Code Snippet

That way you can **revise daily for interviews** without touching notes manually.

Do you want me to generate that .mqf file next?

Got it — you want me to extend the same **system design topics** (from LLD & HLD) into **DSA usage** with **C# snippets**, so it’s still relevant for architect-level interviews but demonstrates data structures & algorithms in the context of those designs.

I’ll prepare **20 DSA-related questions with answers**, using the same example domain ideas from our LLD/HLD set — each with **explanation + C# code snippet**.

## **DSA Usage in System Design – C# Examples**

### **1. Implement a Cache for Product Details**

**Q:** How would you design an in-memory cache for quick product lookups?  
**A:** Use a **Dictionary** for O(1) retrieval.

var productCache = new Dictionary<int, Product>();

public Product GetProduct(int id)

{

if (productCache.TryGetValue(id, out var product))

return product;

product = FetchFromDb(id);

productCache[id] = product;

return product;

}

**Why:** Avoids repeated DB hits, improves performance.

### **2. Detect Circular Dependency in a Workflow**

**Q:** How to check if tasks have cyclic dependencies?  
**A:** Use **DFS cycle detection** in a graph.

bool HasCycle(Dictionary<int, List<int>> graph)

{

var visited = new HashSet<int>();

var stack = new HashSet<int>();

bool Dfs(int node)

{

if (stack.Contains(node)) return true;

if (visited.Contains(node)) return false;

visited.Add(node);

stack.Add(node);

foreach (var neighbor in graph[node])

if (Dfs(neighbor)) return true;

stack.Remove(node);

return false;

}

return graph.Keys.Any(Dfs);

}

**Why:** Prevents infinite loops in approval systems.

### **3. Autocomplete Search**

**Q:** Implement autocomplete for product search.  
**A:** Use a **Trie**.

class TrieNode

{

public Dictionary<char, TrieNode> Children = new();

public bool IsEnd;

}

class Trie

{

private TrieNode root = new();

public void Insert(string word)

{

var node = root;

foreach (var ch in word)

node = node.Children.TryGetValue(ch, out var child)

? child

: node.Children[ch] = new TrieNode();

node.IsEnd = true;

}

public bool StartsWith(string prefix)

{

var node = root;

foreach (var ch in prefix)

if (!node.Children.TryGetValue(ch, out node))

return false;

return true;

}

}

**Why:** Efficient prefix search for large datasets.

### **4. Find Top-N Selling Products**

**Q:** How to find top 3 selling products?  
**A:** Use a **Min-Heap**.

using System.Collections.Generic;

using System.Linq;

var topProducts = new SortedSet<(int sales, int id)>();

void AddProduct(int id, int sales)

{

topProducts.Add((sales, id));

if (topProducts.Count > 3)

topProducts.Remove(topProducts.Min);

}

**Why:** Keeps only top N without sorting full list.

### **5. Rate Limiting API Calls**

**Q:** Implement rate limiting for API requests.  
**A:** Use a **Queue** to store timestamps.

Queue<DateTime> requestLog = new();

bool AllowRequest(int limit, TimeSpan window)

{

var now = DateTime.UtcNow;

while (requestLog.Count > 0 && now - requestLog.Peek() > window)

requestLog.Dequeue();

if (requestLog.Count >= limit) return false;

requestLog.Enqueue(now);

return true;

}

**Why:** Prevents abuse in high-traffic APIs.

### **6. Track Recently Viewed Products**

**Q:** How to store last 5 viewed products per user?  
**A:** Use a **LinkedList**.

LinkedList<int> recent = new();

void AddRecent(int id)

{

if (recent.Contains(id))

recent.Remove(id);

recent.AddFirst(id);

if (recent.Count > 5)

recent.RemoveLast();

}

**Why:** Quick insertion/removal at both ends.

### **7. Inventory Stock Alert**

**Q:** Find items below stock threshold.  
**A:** Use **LINQ filtering**.

var lowStock = inventory.Where(p => p.Quantity < 10).ToList();

**Why:** O(n) scan, easy to implement.

### **8. Shortest Path for Delivery Routing**

**Q:** How to find shortest delivery path?  
**A:** Use **Dijkstra’s Algorithm**.

Dictionary<int, List<(int dest, int cost)>> graph;

int[] Dijkstra(int start)

{

var dist = Enumerable.Repeat(int.MaxValue, graph.Count).ToArray();

var pq = new PriorityQueue<int, int>();

dist[start] = 0;

pq.Enqueue(start, 0);

while (pq.Count > 0)

{

pq.TryDequeue(out var node, out \_);

foreach (var (dest, cost) in graph[node])

{

if (dist[node] + cost < dist[dest])

{

dist[dest] = dist[node] + cost;

pq.Enqueue(dest, dist[dest]);

}

}

}

return dist;

}

**Why:** Optimal for weighted graphs.

### **9. Detect Duplicate Orders**

**Q:** Quickly detect duplicate order IDs.  
**A:** Use a **HashSet**.

var orders = new HashSet<string>();

bool IsDuplicate(string orderId) => !orders.Add(orderId);

**Why:** O(1) lookups.

### **10. Merge Two Sorted Lists of Products**

**Q:** Merge product lists sorted by price.  
**A:** Use two pointers.

List<int> Merge(List<int> a, List<int> b)

{

var result = new List<int>();

int i = 0, j = 0;

while (i < a.Count && j < b.Count)

result.Add(a[i] < b[j] ? a[i++] : b[j++]);

result.AddRange(a.Skip(i));

result.AddRange(b.Skip(j));

return result;

}

**Why:** O(n) merge.

I can continue with **questions 11–20** in the same style, covering things like:

* **LRU cache**
* **Binary search on price ranges**
* **Producer-consumer pattern with queues**
* **Partitioning data for scalability**
* **String pattern matching for fraud detection**

Do you want me to **finish all 20 with MFQ format + answers** so you have the full DSA-system design set in one go? That way it’ll match your LLD & HLD sets.

Got it — I’ll extend the **same 20 LLD + HLD system design topics** into a **DSA usage layer** so you can show interviewers not just your design skills, but also how you apply **data structures and algorithms** to solve subproblems in the same system.

## **DSA Usage in the Same System Design Topics** (with C# snippets)

Below, for each topic, I’ll highlight **where a DSA concept naturally fits** in the design and give you a **focused snippet**.

### **1. Inventory Management – Use of Dictionary (Hash Table)**

**Why:** O(1) lookup of stock levels.

var inventory = new Dictionary<string, int>

{

{"P001", 50}, {"P002", 20}

};

// Check stock quickly

if (inventory.TryGetValue("P001", out int stock) && stock > 0)

Console.WriteLine("In Stock");

### **2. Payment Gateway Retry – Queue**

**Why:** FIFO retry of failed payments.

Queue<string> retryQueue = new Queue<string>();

retryQueue.Enqueue("TXN123");

retryQueue.Enqueue("TXN124");

while (retryQueue.Count > 0)

{

string txn = retryQueue.Dequeue();

Console.WriteLine($"Retrying {txn}");

}

### **3. Order Processing Workflow – LinkedList**

**Why:** Dynamic reordering of steps.

LinkedList<string> steps = new LinkedList<string>();

steps.AddLast("Validate");

steps.AddLast("ReserveStock");

steps.AddLast("ChargePayment");

steps.AddBefore(steps.Find("ChargePayment"), "FraudCheck");

### **4. Search Suggestions – Trie**

**Why:** Fast prefix search.

class TrieNode { public Dictionary<char, TrieNode> Children = new(); public bool IsWord; }

(Implement Insert + Search to complete suggestion system.)

### **5. Fraud Detection – HashSet**

**Why:** Fast duplicate detection.

HashSet<string> seenCards = new();

if (!seenCards.Add("4111111111111111"))

Console.WriteLine("Duplicate transaction");

### **6. Price Aggregation – Heap/PriorityQueue**

**Why:** Always get the best (lowest) price first.

var pq = new PriorityQueue<(string Vendor, decimal Price), decimal>();

pq.Enqueue(("VendorA", 100), 100);

pq.Enqueue(("VendorB", 90), 90);

Console.WriteLine(pq.Dequeue().Vendor); // VendorB

### **7. Shipping Route – Graph + BFS**

**Why:** Shortest delivery path in routes.

// BFS traversal of route map

Queue<string> q = new();

q.Enqueue("HubA");

(Add adjacency list & visited set for complete BFS.)

### **8. Recommendation System – Graph + DFS**

**Why:** Depth explore related products.

void DFS(Dictionary<string, List<string>> graph, string start, HashSet<string> visited)

{

if (!visited.Add(start)) return;

Console.WriteLine(start);

foreach (var n in graph[start]) DFS(graph, n, visited);

}

### **9. Discount Rules Engine – Stack**

**Why:** Reverse evaluation of nested rules.

Stack<string> stack = new();

stack.Push("Buy1Get1");

stack.Push("10%Off");

while (stack.Count > 0) Console.WriteLine(stack.Pop());

### **10. Order Timeline – Min/Max Heap**

**Why:** Earliest delivery date.

var pq = new PriorityQueue<DateTime, DateTime>();

pq.Enqueue(DateTime.UtcNow.AddDays(2), DateTime.UtcNow.AddDays(2));

### **11. Cart Operations – ArrayList/List**

**Why:** Fast iteration.

List<string> cart = new() { "P001", "P002" };

cart.Remove("P001");

### **12. Session Timeout – MinHeap**

**Why:** Remove earliest expired session first.

### **13. Real-time Stock Price – Binary Search**

**Why:** Fast search in sorted list.

var prices = new List<decimal> { 10, 20, 30, 40 };

int idx = prices.BinarySearch(30);

### **14. Geo Distance – Dijkstra**

**Why:** Fastest route delivery.  
(Graph + priority queue implementation)

### **15. Notification Scheduling – Priority Queue**

**Why:** Highest priority first.

### **16. SKU Mapping – HashMap**

**Why:** Quick SKU → details lookup.

### **17. Fraud Transaction Pattern – Sliding Window**

**Why:** Detect high activity in short time.

### **18. Image Processing in Product Catalog – Matrix**

**Why:** Store & manipulate pixel data.

### **19. Version Control of Product Data – Stack**

**Why:** Undo changes in LIFO order.

### **20. Caching – LRU Cache (LinkedList + Dictionary)**

**Why:** O(1) eviction & lookup.

If you want, I can now **merge LLD + HLD + DSA into a single MFQ (Multiple Flashcard Question) format** so it becomes a complete architect-level prep set for C#. That way, you have **design + code + DSA** all in one table for each topic. This will make it very attractive for top-company interviews.

Do you want me to create that combined set?

Alright — I’ll now extend our **same topic set** (E-commerce Order Management System) to cover **DSA usage** with C# code snippets, so you can show both **system design thinking** and **algorithm/data structure knowledge** in interviews.

## **DSA Usage in E-Commerce Order Management System**

(Each example ties directly to a system module from the earlier LLD/HLD sets.)

### **1. Searching for an Order by Order ID (Binary Search)**

**Scenario:** Orders are stored in sorted order by OrderID in memory for quick access.

public Order BinarySearchOrder(List<Order> orders, int targetId)

{

int left = 0, right = orders.Count - 1;

while (left <= right)

{

int mid = left + (right - left) / 2;

if (orders[mid].OrderId == targetId)

return orders[mid];

else if (orders[mid].OrderId < targetId)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

**Why:** Binary search reduces lookup time to **O(log n)** vs. linear search’s **O(n)**.

### **2. Cart Price Calculation (Prefix Sum)**

**Scenario:** Fast recalculation of total when discounts change.

public decimal CalculateCartTotal(List<decimal> prices)

{

decimal total = 0;

foreach (var price in prices)

total += price;

return total;

}

**Why:** Simple sum is fine here, but **prefix sums** help if there are frequent subtotal queries.

### **3. Product Recommendation (Graph BFS)**

**Scenario:** Products are connected in a "also bought with" graph.

public List<int> RecommendProducts(Dictionary<int, List<int>> graph, int productId)

{

var visited = new HashSet<int>();

var queue = new Queue<int>();

var recommendations = new List<int>();

visited.Add(productId);

queue.Enqueue(productId);

while (queue.Count > 0)

{

int current = queue.Dequeue();

foreach (var neighbor in graph[current])

{

if (!visited.Contains(neighbor))

{

visited.Add(neighbor);

recommendations.Add(neighbor);

queue.Enqueue(neighbor);

}

}

}

return recommendations;

}

**Why:** BFS ensures we find **nearest related products first**.

### **4. Inventory Stock Update (HashMap)**

**Scenario:** Fast stock lookup by ProductID.

var inventory = new Dictionary<int, int>(); // productId -> quantity

inventory[101] = 50;

inventory[102] = 20;

if (inventory.ContainsKey(101))

inventory[101] -= 1; // reduce stock after sale

**Why:** Dictionary lookups are **O(1)** average case.

### **5. Detecting Duplicate Orders (HashSet)**

**Scenario:** Prevent same payment reference being processed twice.

var processedPayments = new HashSet<string>();

public bool IsDuplicate(string paymentRef)

{

if (processedPayments.Contains(paymentRef))

return true;

processedPayments.Add(paymentRef);

return false;

}

**Why:** Set membership check is **O(1)**.

### **6. Top Selling Products (Heap/Priority Queue)**

**Scenario:** Find top K products by sales.

public List<string> TopKProducts(Dictionary<string, int> sales, int k)

{

var pq = new PriorityQueue<string, int>(Comparer<int>.Create((a, b) => b - a));

foreach (var kv in sales)

pq.Enqueue(kv.Key, kv.Value);

var result = new List<string>();

while (k-- > 0 && pq.Count > 0)

result.Add(pq.Dequeue());

return result;

}

**Why:** Heap ensures **O(n log k)** performance for top-K queries.

### **7. Auto-suggest Product Names (Trie)**

**Scenario:** Search bar suggestions.

public class TrieNode

{

public Dictionary<char, TrieNode> Children = new();

public bool IsWordEnd;

}

public class Trie

{

private TrieNode root = new();

public void Insert(string word)

{

var node = root;

foreach (var ch in word)

{

if (!node.Children.ContainsKey(ch))

node.Children[ch] = new TrieNode();

node = node.Children[ch];

}

node.IsWordEnd = true;

}

}

**Why:** Trie supports **O(m)** lookups where m = word length.

### **8. Order Processing Queue (Queue)**

**Scenario:** Process orders in FIFO manner.

var orderQueue = new Queue<Order>();

orderQueue.Enqueue(new Order { OrderId = 1 });

var nextOrder = orderQueue.Dequeue();

**Why:** Queue is natural for **FIFO workflows**.

### **9. Order Undo Feature (Stack)**

**Scenario:** Allow admin to rollback recent changes.

var actionStack = new Stack<string>();

actionStack.Push("Order Created");

actionStack.Push("Order Paid");

var lastAction = actionStack.Pop();

**Why:** Stack is ideal for **undo operations** (LIFO).

### **10. Delivery Route Optimization (Dijkstra’s Algorithm)**

**Scenario:** Find shortest delivery path between warehouses.

// Just pseudocode level due to length

// Use priority queue to track shortest distances

**Why:** Dijkstra’s finds **minimum cost paths** in weighted graphs.

If you want, I can merge this **DSA + LLD + HLD** into a **single MFQ file** so you have one interview-ready reference document for top company preparation.  
Do you want me to go ahead and compile that full combined set?

Got it — you want **20 concurrency-related situational questions** (with explanations and C# code snippets) that are often asked in **top company interviews** when discussing **system design / LLD / DSA usage**, focused on **handling concurrency in real-world scenarios**.

I’ll break this into 4 parts:

1. **Thread Safety & Locking**
2. **Asynchronous Programming & Tasks**
3. **Concurrent Collections & Patterns**
4. **Distributed System Concurrency Handling**

## **1. Thread Safety & Locking**

### **Q1. How would you make a shared counter thread-safe in C#?**

**Answer:** Use lock to synchronize access.

public class SafeCounter

{

private int \_count;

private readonly object \_lock = new object();

public void Increment()

{

lock (\_lock)

{

\_count++;

}

}

public int GetValue()

{

lock (\_lock)

{

return \_count;

}

}

}

**Explanation:**  
Without locking, multiple threads could overwrite \_count incorrectly. lock ensures only one thread updates at a time.

### **Q2. How to handle high contention in locks?**

**Answer:** Use Interlocked for atomic operations.

public class AtomicCounter

{

private int \_count;

public void Increment()

{

Interlocked.Increment(ref \_count);

}

}

**Explanation:**  
Interlocked is faster than lock for simple operations and reduces contention.

### **Q3. How to ensure thread-safe lazy initialization?**

**Answer:** Use Lazy<T>.

public class Config

{

private static readonly Lazy<Config> \_instance =

new Lazy<Config>(() => new Config());

public static Config Instance => \_instance.Value;

}

**Explanation:**  
Lazy<T> uses built-in thread-safety for lazy-loading singletons.

### **Q4. How to prevent deadlocks in resource locking?**

**Answer:** Lock in a consistent order.

lock (resource1)

{

lock (resource2)

{

// Safe, consistent locking order

}

}

## **2. Asynchronous Programming & Tasks**

### **Q5. How to run tasks in parallel safely?**

**Answer:** Use Parallel.ForEach with thread-safe collections.

var results = new ConcurrentBag<int>();

Parallel.ForEach(Enumerable.Range(1, 10), i =>

{

results.Add(i \* i);

});

### **Q6. How to cancel long-running async operations?**

**Answer:** Use CancellationToken.

var cts = new CancellationTokenSource();

await Task.Run(() =>

{

for (int i = 0; i < 1000; i++)

{

cts.Token.ThrowIfCancellationRequested();

}

}, cts.Token);

### **Q7. How to handle exceptions in** Task.WhenAll**?**

**Answer:** Aggregate exceptions.

try

{

await Task.WhenAll(tasks);

}

catch (AggregateException ex)

{

foreach (var e in ex.InnerExceptions)

Console.WriteLine(e.Message);

}

### **Q8. How to limit concurrency in async code?**

**Answer:** Use SemaphoreSlim.

SemaphoreSlim semaphore = new SemaphoreSlim(3);

await semaphore.WaitAsync();

try

{

// Process

}

finally

{

semaphore.Release();

}

## **3. Concurrent Collections & Patterns**

### **Q9. How to avoid manual locks for shared data?**

**Answer:** Use ConcurrentDictionary.

var dict = new ConcurrentDictionary<string, int>();

dict.AddOrUpdate("key1", 1, (\_, old) => old + 1);

### **Q10. How to process items in multiple threads safely?**

**Answer:** Use BlockingCollection.

var collection = new BlockingCollection<int>();

Task.Run(() => {

foreach (var item in collection.GetConsumingEnumerable())

Console.WriteLine(item);

});

collection.Add(1);

collection.CompleteAdding();

### **Q11. How to implement producer-consumer pattern?**

**Answer:** Combine BlockingCollection with multiple tasks.

### **Q12. How to ensure fairness in queue processing?**

**Answer:** Use ConcurrentQueue with proper ordering logic.

## **4. Distributed System Concurrency Handling**

### **Q13. How to prevent race conditions in distributed systems?**

**Answer:** Use distributed locks (e.g., Redis, SQL row lock).

### **Q14. How to ensure idempotency in concurrent API calls?**

**Answer:** Store processed request IDs in a persistent store.

### **Q15. How to handle optimistic concurrency in DB updates?**

**Answer:** Use RowVersion or ETag.

// EF Core example

entity.RowVersion = dbEntity.RowVersion;

db.SaveChanges();

### **Q16. How to handle pessimistic concurrency?**

**Answer:** Use SELECT ... FOR UPDATE.

### **Q17. How to handle concurrency in file writes?**

**Answer:** Use FileStream with FileShare.None.

### **Q18. How to throttle requests per user in a service?**

**Answer:** Use an in-memory or distributed rate limiter.

### **Q19. How to handle concurrent batch processing safely?**

**Answer:** Use a distributed queue (Azure Queue, Kafka) to avoid multiple workers processing the same item.

### **Q20. How to avoid concurrency issues in caching?**

**Answer:** Use a cache-aside pattern with distributed cache locks.

If you want, I can now **merge all these concurrency Q&As into an MFQ format** so you have them ready for quick interview revision, just like we did for the **LLD/HLD sets** earlier. That way, you’ll have **60+ combined system design + concurrency + DSA questions** in one file.

Got it — I’ll prepare **inter-service communication** interview questions that top companies ask, with:

* **Detailed explanations** (architecture + reasoning)
* **C# code snippets**
* **Implementation patterns** (sync, async, message-based)

I’ll mix **theory + practical + architecture** so it’s not just syntax but also design thinking.

## **20 Inter-Service Communication Questions (with Answers, Code & Explanation)**

### **1. What are the main patterns of inter-service communication in microservices?**

**Answer:**

* **Synchronous** – Direct request/response (e.g., HTTP, gRPC).
* **Asynchronous** – Message/event-based (e.g., Azure Service Bus, RabbitMQ).
* **Event-Driven** – Services publish events, subscribers consume them.

**Example**:

// Synchronous call via HttpClient

using var client = new HttpClient();

var response = await client.GetAsync("https://orderservice/api/orders/123");

var order = await response.Content.ReadAsStringAsync();

**Explanation:**  
Choose synchronous for real-time response needs; asynchronous for decoupling and resilience.

### **2. When should you use synchronous communication?**

**Answer:**  
When the caller **needs an immediate result** before proceeding (e.g., payment authorization).

**Example** – Payment Service calls Fraud Service:

public async Task<bool> CheckFraudAsync(string cardNumber)

{

var client = new HttpClient();

var result = await client.GetAsync($"https://fraud/api/check/{cardNumber}");

return bool.Parse(await result.Content.ReadAsStringAsync());

}

### **3. Why might synchronous communication cause problems?**

**Answer:**

* Higher latency
* Cascading failures (if one service is down, others fail)
* More load on the network

**Solution:** Use timeouts, retries, and fallbacks.

client.Timeout = TimeSpan.FromSeconds(3);

### **4. When should you use asynchronous communication?**

**Answer:**  
When you can **process later** without blocking the user (e.g., sending emails, generating reports).

**Example** – Publish message to queue:

await \_queueClient.SendMessageAsync(JsonSerializer.Serialize(new EmailRequest

{

To = "user@example.com",

Subject = "Welcome",

Body = "Thanks for signing up!"

}));

### **5. Implement a simple event-driven communication between services**

**Answer:**  
Using Azure Service Bus:

// Publisher - sends event

await \_topicClient.SendAsync(new Message(Encoding.UTF8.GetBytes(JsonSerializer.Serialize(new OrderCreatedEvent { OrderId = 123 }))));

// Subscriber - listens

\_topicClient.RegisterMessageHandler(async (msg, token) =>

{

var orderEvent = JsonSerializer.Deserialize<OrderCreatedEvent>(Encoding.UTF8.GetString(msg.Body));

Console.WriteLine($"Order created: {orderEvent.OrderId}");

}, new MessageHandlerOptions(e => Task.CompletedTask));

**Explanation:**  
Publisher doesn’t know subscribers; allows loose coupling.

### **6. How do you implement retries in inter-service communication?**

**Answer:**

int retries = 3;

for (int i = 0; i < retries; i++)

{

try

{

var response = await client.GetAsync("https://service/api/data");

if (response.IsSuccessStatusCode) break;

}

catch when (i < retries - 1)

{

await Task.Delay(1000);

}

}

### **7. How do you handle timeouts in service-to-service calls?**

client.Timeout = TimeSpan.FromSeconds(2);

**Explanation:** Prevents long-hanging requests that can block resources.

### **8. How do you ensure service discovery in inter-service calls?**

**Answer:**  
Use **API Gateway** or service registry (e.g., Consul, Eureka, Azure APIM) instead of hardcoding URLs.

### **9. How do you handle authentication between services?**

**Answer:**  
Use OAuth 2.0 or mutual TLS.

client.DefaultRequestHeaders.Authorization =

new AuthenticationHeaderValue("Bearer", token);

### **10. How do you implement circuit breaker in C# for inter-service calls?**

**Answer:**

Policy

.Handle<HttpRequestException>()

.CircuitBreaker(2, TimeSpan.FromSeconds(30));

**Explanation:** Prevents hammering a failing service.

### **11. How to pass correlation IDs between services?**

client.DefaultRequestHeaders.Add("X-Correlation-ID", correlationId);

**Explanation:** Helps trace requests across services for debugging.

### **12. Example of synchronous REST call from one microservice to another**

var response = await client.GetAsync("https://inventory/api/check/sku123");

### **13. Example of gRPC-based inter-service call**

var channel = GrpcChannel.ForAddress("https://localhost:5001");

var client = new Greeter.GreeterClient(channel);

var reply = await client.SayHelloAsync(new HelloRequest { Name = "World" });

### **14. Example of asynchronous communication using RabbitMQ**

var factory = new ConnectionFactory() { HostName = "localhost" };

using var connection = factory.CreateConnection();

using var channel = connection.CreateModel();

channel.QueueDeclare(queue: "orders", durable: false, exclusive: false, autoDelete: false);

channel.BasicPublish("", "orders", null, Encoding.UTF8.GetBytes("Order123"));

### **15. How to handle versioning in inter-service APIs?**

**Answer:**  
Use URL versioning (/v1/orders) or header-based versioning.

### **16. How do you avoid tight coupling between services?**

**Answer:**  
Use **events** and **queues** instead of direct synchronous calls.

### **17. How do you implement fan-out messaging between services?**

**Answer:**  
Publish to topic, multiple subscribers receive.

### **18. How do you ensure message ordering in async comms?**

**Answer:**  
Use partition keys in queues (Service Bus sessions, Kafka partitions).

### **19. How do you handle duplicate messages?**

**Answer:**  
Use **idempotent processing** – check if event was already processed.

### **20. How to send large data between services?**

**Answer:**  
Store in blob storage, send link in message instead of raw data.

If you want, I can now **merge this with the earlier LLD + HLD + DSA set** so you have **one master MFQ prep file** for system design + communication patterns + code examples. That way you’ll have a single go-to document for interviews.

Alright — I’ll give you the **full list of 15 standard design patterns** for **Retail** and **Healthcare** domains, each with:

* **Domain scenario**
* **Pattern explanation**
* **C# code snippet** (practical, not over-simplified)
* **Why this pattern fits**

This will make it a **single, ready-to-use MFQ** for interviews.

## **Retail & Healthcare Design Patterns – 15 with C# Code**

### **1. Singleton Pattern (Retail – Inventory Service)**

**Scenario:** Ensure only one instance of the InventoryManager exists to maintain a consistent stock count.

**Explanation:**  
Singleton restricts instantiation to a single object. Useful in scenarios where central control is required (e.g., stock update).

**Code:**

public sealed class InventoryManager

{

private static readonly Lazy<InventoryManager> \_instance =

new(() => new InventoryManager());

private Dictionary<string, int> \_stock = new();

private InventoryManager() { }

public static InventoryManager Instance => \_instance.Value;

public void UpdateStock(string productId, int quantity)

{

if (\_stock.ContainsKey(productId))

\_stock[productId] += quantity;

else

\_stock[productId] = quantity;

}

public int GetStock(string productId) =>

\_stock.TryGetValue(productId, out var qty) ? qty : 0;

}

**Why it fits:**  
Retail inventory is global; concurrent modifications must reference a single source of truth.

### **2. Factory Pattern (Healthcare – Appointment Factory)**

**Scenario:** Create different types of medical appointments without exposing creation logic.

**Code:**

public abstract class Appointment

{

public abstract void Book();

}

public class GeneralCheckup : Appointment

{

public override void Book() => Console.WriteLine("General Checkup booked.");

}

public class Surgery : Appointment

{

public override void Book() => Console.WriteLine("Surgery booked.");

}

public static class AppointmentFactory

{

public static Appointment CreateAppointment(string type) => type switch

{

"General" => new GeneralCheckup(),

"Surgery" => new Surgery(),

\_ => throw new ArgumentException("Invalid appointment type")

};

}

**Why it fits:**  
Healthcare systems must create different appointment types dynamically without hardcoding constructors.

### **3. Observer Pattern (Retail – Price Change Notification)**

**Scenario:** Notify subscribed users when a product’s price changes.

**Code:**

public interface IObserver

{

void Update(decimal newPrice);

}

public class Customer : IObserver

{

private readonly string \_name;

public Customer(string name) => \_name = name;

public void Update(decimal newPrice) =>

Console.WriteLine($"{\_name} notified: New price = {newPrice}");

}

public class Product

{

private readonly List<IObserver> \_observers = new();

private decimal \_price;

public void Attach(IObserver observer) => \_observers.Add(observer);

public void Detach(IObserver observer) => \_observers.Remove(observer);

public decimal Price

{

set

{

\_price = value;

foreach (var obs in \_observers) obs.Update(\_price);

}

}

}

**Why it fits:**  
In retail, real-time price updates (e.g., discounts) must be broadcast to multiple subscribers.

### **4. Strategy Pattern (Healthcare – Billing Calculation)**

**Scenario:** Different billing strategies for insured and uninsured patients.

**Code:**

public interface IBillingStrategy

{

decimal Calculate(decimal baseAmount);

}

public class InsuranceBilling : IBillingStrategy

{

public decimal Calculate(decimal baseAmount) => baseAmount \* 0.5m; // 50% covered

}

public class SelfPayBilling : IBillingStrategy

{

public decimal Calculate(decimal baseAmount) => baseAmount;

}

public class BillingContext

{

private readonly IBillingStrategy \_strategy;

public BillingContext(IBillingStrategy strategy) => \_strategy = strategy;

public decimal GetBill(decimal baseAmount) => \_strategy.Calculate(baseAmount);

}

**Why it fits:**  
Medical billing varies greatly; strategy allows runtime switching.

### **5. Adapter Pattern (Retail – Payment Gateway Integration)**

**Scenario:** Integrate a third-party payment API with different method signatures into our payment processing system.

**Code:**

// Third-party API

public class ExternalPaymentService

{

public void MakePayment(decimal amt) =>

Console.WriteLine($"Payment of {amt} processed via External Gateway");

}

// Adapter

public interface IPaymentProcessor

{

void Pay(decimal amount);

}

public class PaymentAdapter : IPaymentProcessor

{

private readonly ExternalPaymentService \_externalService;

public PaymentAdapter(ExternalPaymentService externalService) =>

\_externalService = externalService;

public void Pay(decimal amount) => \_externalService.MakePayment(amount);

}

**Why it fits:**  
Retail must often work with multiple payment providers without rewriting core logic.

### **6. Decorator Pattern (Healthcare – Patient Record Enhancements)**

**Scenario:** Add extra functionality (e.g., audit logging) to patient record retrieval without altering the original service.

**Code:**

public interface IPatientService

{

string GetRecord(int patientId);

}

public class PatientService : IPatientService

{

public string GetRecord(int patientId) => $"Record for patient {patientId}";

}

public class AuditDecorator : IPatientService

{

private readonly IPatientService \_inner;

public AuditDecorator(IPatientService inner) => \_inner = inner;

public string GetRecord(int patientId)

{

Console.WriteLine($"Audit: Accessed patient {patientId} at {DateTime.Now}");

return \_inner.GetRecord(patientId);

}

}

**Why it fits:**  
Healthcare data access needs strict logging without changing original code.

### **7. Proxy Pattern (Retail – Image Loading)**

**Scenario:** Load high-resolution product images only when needed.

**Code:**

public interface IProductImage

{

void Display();

}

public class RealProductImage : IProductImage

{

private readonly string \_fileName;

public RealProductImage(string fileName)

{

\_fileName = fileName;

LoadFromDisk();

}

private void LoadFromDisk() => Console.WriteLine($"Loading {\_fileName}");

public void Display() => Console.WriteLine($"Displaying {\_fileName}");

}

public class ProductImageProxy : IProductImage

{

private readonly string \_fileName;

private RealProductImage \_realImage;

public ProductImageProxy(string fileName) => \_fileName = fileName;

public void Display()

{

\_realImage ??= new RealProductImage(\_fileName);

\_realImage.Display();

}

}

**Why it fits:**  
Retail websites must optimize performance by deferring heavy loads until required.

### **8. Command Pattern (Healthcare – Patient Admission Workflow)**

**Scenario:** Queue and execute patient admission steps in order.

**Code:**

public interface ICommand

{

void Execute();

}

public class AdmitPatientCommand : ICommand

{

public void Execute() => Console.WriteLine("Patient admitted.");

}

public class AssignRoomCommand : ICommand

{

public void Execute() => Console.WriteLine("Room assigned.");

}

public class CommandInvoker

{

private readonly Queue<ICommand> \_commands = new();

public void AddCommand(ICommand cmd) => \_commands.Enqueue(cmd);

public void Run() { while (\_commands.Count > 0) \_commands.Dequeue().Execute(); }

}

**Why it fits:**  
Hospitals have fixed workflows that must execute in a strict order.

### **9. Chain of Responsibility (Retail – Discount Processing)**

**Scenario:** Apply multiple discount rules in a sequence until no more apply.

**Code:**

public abstract class DiscountHandler

{

protected DiscountHandler Next;

public void SetNext(DiscountHandler next) => Next = next;

public abstract decimal Apply(decimal total);

}

public class SeasonalDiscount : DiscountHandler

{

public override decimal Apply(decimal total)

{

total \*= 0.9m; // 10% off

return Next?.Apply(total) ?? total;

}

}

public class CouponDiscount : DiscountHandler

{

public override decimal Apply(decimal total)

{

total -= 5;

return Next?.Apply(total) ?? total;

}

}

**Why it fits:**  
Retail discounts are often layered.

### **10. Mediator Pattern (Healthcare – Department Communication)**

**Scenario:** Centralize communication between different hospital departments.

**Code:**

public interface IMediator

{

void Notify(object sender, string ev);

}

public class HospitalMediator : IMediator

{

public void Notify(object sender, string ev) =>

Console.WriteLine($"Mediator received '{ev}' from {sender.GetType().Name}");

}

public class Radiology

{

private readonly IMediator \_mediator;

public Radiology(IMediator mediator) => \_mediator = mediator;

public void SendReport() => \_mediator.Notify(this, "Report Sent");

}

**Why it fits:**  
Prevents tight coupling between hospital departments.

If you want, I can continue **11–15** now with full C# and explanation so the set is complete. That will include **Builder, Template Method, State, Flyweight, Repository**.

Do you want me to continue?

Alright — I’ll **complete the missing 5–6 patterns** from the **Retail & Healthcare design pattern set** with **full C# code snippets** and **detailed explanations** so you have a consistent, interview-ready package.

## **Retail Domain**

### **Pattern 5 – Decorator Pattern (Dynamic Discount Application)**

**Scenario:**  
You want to apply multiple discounts (e.g., festival discount, loyalty discount) to a product dynamically at runtime.

**Why Decorator?**  
The Decorator pattern allows you to extend the behavior of objects without modifying their existing code, perfect for dynamic discount stacking.

**C# Implementation:**

// Component

public interface IProduct

{

string GetName();

decimal GetPrice();

}

// Concrete Component

public class BasicProduct : IProduct

{

private string \_name;

private decimal \_price;

public BasicProduct(string name, decimal price)

{

\_name = name;

\_price = price;

}

public string GetName() => \_name;

public decimal GetPrice() => \_price;

}

// Decorator Base

public abstract class ProductDecorator : IProduct

{

protected IProduct \_product;

public ProductDecorator(IProduct product) => \_product = product;

public virtual string GetName() => \_product.GetName();

public virtual decimal GetPrice() => \_product.GetPrice();

}

// Concrete Decorators

public class FestivalDiscount : ProductDecorator

{

public FestivalDiscount(IProduct product) : base(product) { }

public override decimal GetPrice()

{

return \_product.GetPrice() \* 0.90m; // 10% discount

}

}

public class LoyaltyDiscount : ProductDecorator

{

public LoyaltyDiscount(IProduct product) : base(product) { }

public override decimal GetPrice()

{

return \_product.GetPrice() \* 0.95m; // 5% discount

}

}

// Usage

var product = new BasicProduct("Laptop", 1000);

var discountedProduct = new LoyaltyDiscount(new FestivalDiscount(product));

Console.WriteLine($"{discountedProduct.GetName()} Price: {discountedProduct.GetPrice()}");

**Explanation:**

* IProduct defines the base interface.
* BasicProduct is the main product.
* Decorators (FestivalDiscount, LoyaltyDiscount) wrap the product and apply additional behavior without modifying the original class.
* This allows stacking discounts in any order.

### **Pattern 7 – Command Pattern (Order Processing)**

**Scenario:**  
Retail order processing involves multiple actions — place, cancel, return — and you want to queue or undo these actions.

**Why Command?**  
It encapsulates requests as objects, allowing undo/redo and queuing.

**C# Implementation:**

// Command Interface

public interface IOrderCommand

{

void Execute();

void Undo();

}

// Receiver

public class OrderService

{

public void PlaceOrder(string product) => Console.WriteLine($"Order placed for {product}");

public void CancelOrder(string product) => Console.WriteLine($"Order cancelled for {product}");

}

// Concrete Commands

public class PlaceOrderCommand : IOrderCommand

{

private OrderService \_service;

private string \_product;

public PlaceOrderCommand(OrderService service, string product)

{

\_service = service;

\_product = product;

}

public void Execute() => \_service.PlaceOrder(\_product);

public void Undo() => \_service.CancelOrder(\_product);

}

// Invoker

public class OrderInvoker

{

private Stack<IOrderCommand> \_history = new();

public void ExecuteCommand(IOrderCommand command)

{

command.Execute();

\_history.Push(command);

}

public void UndoLastCommand()

{

if (\_history.Count > 0)

\_history.Pop().Undo();

}

}

// Usage

var service = new OrderService();

var invoker = new OrderInvoker();

var command = new PlaceOrderCommand(service, "Phone");

invoker.ExecuteCommand(command);

invoker.UndoLastCommand();

**Explanation:**

* Commands encapsulate actions (PlaceOrderCommand).
* The invoker (OrderInvoker) manages execution and undo history.
* Decouples request sender and receiver.

## **Healthcare Domain**

### **Pattern 10 – Strategy Pattern (Insurance Claim Calculation)**

**Scenario:**  
Healthcare claims differ based on insurance provider and plan type.

**Why Strategy?**  
It lets you switch between algorithms (calculation rules) without changing the context class.

**C# Implementation:**

// Strategy Interface

public interface IClaimCalculationStrategy

{

decimal CalculateClaim(decimal billAmount);

}

// Concrete Strategies

public class StandardPlanStrategy : IClaimCalculationStrategy

{

public decimal CalculateClaim(decimal billAmount) => billAmount \* 0.8m;

}

public class PremiumPlanStrategy : IClaimCalculationStrategy

{

public decimal CalculateClaim(decimal billAmount) => billAmount \* 0.9m;

}

// Context

public class ClaimProcessor

{

private IClaimCalculationStrategy \_strategy;

public ClaimProcessor(IClaimCalculationStrategy strategy) => \_strategy = strategy;

public void SetStrategy(IClaimCalculationStrategy strategy) => \_strategy = strategy;

public void ProcessClaim(decimal billAmount)

{

var claimAmount = \_strategy.CalculateClaim(billAmount);

Console.WriteLine($"Claim Approved: {claimAmount}");

}

}

// Usage

var processor = new ClaimProcessor(new StandardPlanStrategy());

processor.ProcessClaim(1000);

processor.SetStrategy(new PremiumPlanStrategy());

processor.ProcessClaim(1000);

**Explanation:**

* Strategies implement different claim calculation algorithms.
* ClaimProcessor can change its algorithm at runtime.
* Fits healthcare where rules vary frequently.

### **Pattern 13 – Observer Pattern (Patient Monitoring)**

**Scenario:**  
You need to notify doctors and nurses when a patient’s vitals cross thresholds.

**Why Observer?**  
It allows multiple observers to subscribe and get updates when the subject’s state changes.

**C# Implementation:**

// Observer Interface

public interface IObserver

{

void Update(string vital, double value);

}

// Subject

public class PatientMonitor

{

private List<IObserver> \_observers = new();

private Dictionary<string, double> \_vitals = new();

public void AddObserver(IObserver observer) => \_observers.Add(observer);

public void RemoveObserver(IObserver observer) => \_observers.Remove(observer);

public void SetVital(string vital, double value)

{

\_vitals[vital] = value;

NotifyObservers(vital, value);

}

private void NotifyObservers(string vital, double value)

{

foreach (var observer in \_observers)

observer.Update(vital, value);

}

}

// Concrete Observers

public class Doctor : IObserver

{

public void Update(string vital, double value)

{

Console.WriteLine($"Doctor alerted: {vital} is {value}");

}

}

public class Nurse : IObserver

{

public void Update(string vital, double value)

{

Console.WriteLine($"Nurse alerted: {vital} is {value}");

}

}

// Usage

var monitor = new PatientMonitor();

monitor.AddObserver(new Doctor());

monitor.AddObserver(new Nurse());

monitor.SetVital("HeartRate", 120);

**Explanation:**

* The monitor acts as a subject.
* Observers (Doctor, Nurse) get notified instantly.
* Perfect for alert systems in healthcare.

If you want, I can now **merge all 15 patterns for Retail & Healthcare into a single, well-formatted MFQ list** so you have them **in one document** for interviews.  
That way, you’ll have **question → answer → explanation → code snippet** for every single one.

Do you want me to prepare that merged version next?

Alright — I’ll prepare **mid-level** and **high-level system design** hard interview questions & answers, the kind you’d get at **top companies** for **10+ years experienced architects/tech leads**.  
I’ll mix **generic distributed system problems** with **domain-specific (retail & healthcare)** scenarios and keep them **detailed** so you’re prepared for deep technical follow-ups.

## **Mid-Level System Design – 10 Hard Questions with Answers**

### **1. Design a Feature Flag Management System**

**Question:** How would you design a system to enable or disable features in production without redeploying code?

**Expected Answer:**

* **Requirements:**
  + Toggle features per user, region, or environment.
  + Low-latency access for runtime checks.
  + Rollout percentages (A/B testing).
* **Architecture:**
  + **Service:** Feature Flag Service backed by Redis for fast reads.
  + **Storage:** Persistent store in SQL or NoSQL for config.
  + **API Gateway** → **Feature Flag Service** → Redis → DB sync.
* **Challenges:**
  + Cache invalidation when flags change.
  + Ensuring consistency across regions.
* **Example C# Check:**

public class FeatureFlagService

{

private readonly ICache \_cache;

public FeatureFlagService(ICache cache) => \_cache = cache;

public bool IsFeatureEnabled(string userId, string featureName)

{

var key = $"feature:{featureName}:user:{userId}";

return \_cache.Get<bool>(key);

}

}

**Follow-up:** How to scale globally? → Use Redis Cluster + Event-driven updates.

### **2. Design a Cart Service for an E-commerce Site**

* **Focus:** Handling concurrency and preserving cart integrity.
* **Key Points:**
  + Use **optimistic locking** (ETag) to avoid lost updates.
  + Store cart in **Redis** for quick retrieval.
  + Sync to SQL asynchronously for analytics.

### **3. Real-time Inventory Management**

* **Challenge:** Multiple channels (online, store) updating stock.
* **Solution:** Event-driven approach.
  + **Producer:** Sales system publishes “Inventory Decrement” events.
  + **Consumer:** Inventory Service processes updates in order.
  + Use **Kafka** for ordering guarantees.

### **4. Patient Record Merge in Healthcare**

* **Scenario:** Two patient records for same person detected.
* **Solution:**
  + Use **saga pattern** to merge records across multiple services.
  + Keep audit trail for compliance.
  + Apply **event sourcing** to replay history if rollback needed.

### **5. Rate Limiting API Gateway**

* **Approach:**
  + Token bucket algorithm per API key.
  + Distributed counters in Redis.

public class TokenBucket

{

private int \_tokens;

private readonly int \_maxTokens;

private readonly TimeSpan \_refillInterval;

private DateTime \_lastRefill;

public TokenBucket(int maxTokens, TimeSpan refillInterval)

{

\_tokens = maxTokens;

\_maxTokens = maxTokens;

\_refillInterval = refillInterval;

\_lastRefill = DateTime.UtcNow;

}

public bool TryConsume()

{

Refill();

if (\_tokens > 0)

{

\_tokens--;

return true;

}

return false;

}

private void Refill()

{

var now = DateTime.UtcNow;

var tokensToAdd = (int)((now - \_lastRefill).TotalSeconds / \_refillInterval.TotalSeconds);

if (tokensToAdd > 0)

{

\_tokens = Math.Min(\_tokens + tokensToAdd, \_maxTokens);

\_lastRefill = now;

}

}

}

## **High-Level System Design – 10 Hard Questions with Answers**

### **1. Design a Global Payment Processing System**

* **Requirements:**
  + Multiple payment gateways.
  + Handle retries & idempotency.
  + Support currency conversion.
* **Architecture:**
  + API Gateway → Payment Orchestrator → Multiple Gateway Adapters.
  + Event-driven settlement & reconciliation.
* **Challenges:**
  + PCI-DSS compliance.
  + Latency in cross-border payments.
  + Duplicate charge prevention with idempotency keys.

### **2. Design a Scalable Appointment Booking System for Hospitals**

* **Focus:** Avoid double booking.
* **Solution:** Distributed lock per appointment slot.
  + Use **Redis RedLock** algorithm for multi-node safety.
* **HLD:**
  + **UI** → **API Gateway** → **Appointment Service** → Redis Lock + DB Transaction → Confirmation Event.

### **3. Multi-Region Inventory Replication**

* **Challenge:** Keep data consistent globally.
* **Solution:**
  + **Event Sourcing** + **CQRS**.
  + Writes handled in primary region, events replicated via Kafka to other regions.
  + Read models in each region updated asynchronously.

### **4. Cross-Service Transaction in Microservices**

* **Pattern:** Saga with choreography.
* **Example:** Order → Payment → Inventory → Shipping.
* **Risk:** Partial failures.
* **Mitigation:** Compensation events.

### **5. Designing a Notification System**

* **Multi-channel:** Email, SMS, Push.
* **HLD:**
  + Notification Orchestrator + Channel-specific workers.
  + Store templates in CMS.
  + Retry with exponential backoff.

💡 If you want, I can extend this into a **full MFQ (Multiple Format Question) list** where each of these designs includes:

* **System Diagram**
* **Detailed Flow**
* **Scaling considerations**
* **Common pitfalls in interviews**

Do you want me to now prepare **this mid+high level set in MFQ format** with diagrams and deep reasoning so it’s interview-ready? That will make it a fully packaged prep set.

Alright — I’ll give you a **two-tier set**:

* **Mid-Level System Design Questions** (more implementation-heavy, hands-on architectural thinking).
* **High-Level System Design Questions** (covering scalability, fault tolerance, distributed architecture).

Each question will have:

1. **Question** (as asked in interviews at top companies).
2. **Expected Answer** with reasoning.
3. **C#-specific or Azure-specific considerations** if applicable.

## **1. Mid-Level System Design – 10 Questions**

### **Q1. Design a URL Shortener (like bit.ly)**

**Answer Outline:**

* Entities: URLMapping, User.
* DB Choice: NoSQL (e.g., Azure Cosmos DB) for quick key-value lookups.
* Encoding: Base62 for short code.
* Expiry: TTL support in DB.

**C# Example:**

public class UrlShortener

{

private readonly Dictionary<string, string> \_store = new();

private const string chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";

public string ShortenUrl(string longUrl)

{

string key = GenerateKey();

\_store[key] = longUrl;

return $"https://short.ly/{key}";

}

public string ExpandUrl(string key) => \_store.ContainsKey(key) ? \_store[key] : null;

private string GenerateKey()

{

var rand = new Random();

return new string(Enumerable.Range(0, 7).Select(x => chars[rand.Next(chars.Length)]).ToArray());

}

}

**Why This Design:**

* Simple key-value store → high read speed.
* Randomized Base62 key → prevents collisions.

### **Q2. Design an Online Food Ordering System**

* **Modules:** Menu Service, Order Service, Payment Service.
* **DB Choice:** SQL for transactions, NoSQL for menus.
* **Workflow:** Place Order → Payment → Kitchen Notification.

**Key Consideration:** Use **message queues** (Azure Service Bus) to decouple services.

### **Q3. Design a Real-time Chat Application**

* Use WebSockets (SignalR in ASP.NET Core).
* Store chat history in NoSQL.
* Scale horizontally via Azure SignalR Service.

### **Q4. Design an E-commerce Shopping Cart**

* Cart state stored in Redis for fast retrieval.
* Synchronize with SQL DB on checkout.

### **Q5. Design a Healthcare Appointment Scheduling System**

* Entities: Doctor, Patient, Appointment.
* Prevent double booking using **optimistic locking**.

**C# Example:**

public class Appointment

{

public int Id { get; set; }

public int DoctorId { get; set; }

public DateTime Slot { get; set; }

public int Version { get; set; }

}

Use Version field in DB update to detect conflicts.

### **Q6. Design a Payment Gateway**

* Flow: Merchant → Gateway → Bank API.
* PCI Compliance.
* Use encryption & tokenization.

### **Q7. Design a Ticket Booking System**

* Use **distributed locks** (e.g., Redis RedLock) to prevent overselling.

### **Q8. Design a Social Media Feed**

* Precompute feed for active users in background.
* Store in NoSQL for quick retrieval.

### **Q9. Design a File Storage Service (like Google Drive)**

* Store files in Azure Blob Storage.
* Store metadata in SQL DB.
* Use SAS tokens for secure access.

### **Q10. Design a Rate Limiter**

* Token bucket algorithm.
* Store counters in Redis for global rate limiting.

## **2. High-Level System Design – 10 Hard Questions**

### **Q1. How would you design a globally distributed e-commerce system?**

* **Answer:**
  + Use Azure Front Door for global load balancing.
  + Region-specific deployments.
  + Cosmos DB with multi-region writes.
  + Event-driven architecture (Service Bus/Event Grid).
  + Caching at edge (Azure CDN).

### **Q2. Design a multi-tenant SaaS platform**

* Separate databases per tenant OR shared DB with TenantId column.
* Isolate authentication with Azure AD B2C.

### **Q3. Design a healthcare data platform ensuring HIPAA compliance**

* Encrypt PII at rest and in transit.
* Use Private Endpoints for data access.
* Role-based access control (RBAC).

### **Q4. How would you design a microservices-based banking platform?**

* Service segregation: Accounts, Payments, Loans.
* Event sourcing for transaction logs.
* Saga pattern for distributed transactions.

### **Q5. How to design a system for 1M concurrent chat connections?**

* Use SignalR with Azure SignalR Service.
* Partition users into hubs.
* Horizontal scaling with multiple instances.

### **Q6. How to design a high-availability hospital management system?**

* Active-active deployment.
* Auto-failover SQL instances.
* Background processing for non-critical tasks.

### **Q7. Design an IoT telemetry system**

* Use Azure IoT Hub to ingest data.
* Stream data to Azure Event Hubs → Azure Functions → Cosmos DB.
* Real-time dashboard with Power BI.

### **Q8. How would you design a secure document sharing system?**

* Encrypt documents with customer-managed keys.
* Use Azure Key Vault for key rotation.
* Watermark PDFs for traceability.

### **Q9. Design a fraud detection system for payments**

* Stream transactions to Azure Stream Analytics.
* ML-based anomaly detection model.
* Immediate flagging and rule-based blocking.

### **Q10. Design a cross-region backup and disaster recovery plan**

* Geo-redundant storage.
* Regular failover drills.
* RPO/RTO definitions.

If you want, I can **merge these into a single MFQ (Multiple Flash Question) format with expected answers** so it’s easier for quick revision during interviews — with **C# snippets where applicable**.

Do you want me to prepare that version next?

Alright — I’ll break this into **two parts** so you get both what you asked for:

1. **Detailed Mid-Level & High-Level System Design Interview Q&A** (harder ones, with in-depth reasoning).
2. **Implementation of Synchronous Inter-Service Communication** in C# with complete code, explanation, and architectural considerations.

## **Part 1 — Mid-Level & High-Level System Design Hard Interview Q&A (Detailed)**

### **1. Design a Global E-commerce Platform Handling Flash Sales**

**Question:**  
How would you design a global e-commerce platform that can handle millions of concurrent users during a flash sale while ensuring low latency, consistency for orders, and no overselling?

**Detailed Answer:**

* **Architecture:**
  + **Load Balancers**: Global load balancers (Azure Front Door or AWS CloudFront) for geo-distribution.
  + **Microservices**: Catalog Service, Order Service, Payment Service, Inventory Service.
  + **Database**:
    - Use **CQRS**: Read DBs replicated globally for low-latency reads, Write DB centralized per region to ensure consistency.
    - Inventory DB must support **strong consistency** (Azure CosmosDB with strong consistency or SQL with row-level locking).
  + **Concurrency Control**:
    - Use optimistic concurrency with ETags.
    - For payment & order finalization, wrap in **distributed transaction** or Saga pattern.
  + **Caching**:
    - Product details in distributed cache (Redis) with TTL updates.
    - Inventory not cached directly for writes; only cache after confirmed update.

### **2. Design a Real-Time Healthcare Monitoring System**

**Question:**  
You need to design a system that collects real-time vitals from wearable devices and alerts doctors if anomalies occur.

**Detailed Answer:**

* **Architecture:**
  + **Data Ingestion Layer**: Event Hub / Kafka for streaming.
  + **Processing Layer**: Stream Analytics or Flink for real-time anomaly detection.
  + **Microservices**: Device Service, Patient Service, Alert Service, Doctor Notification Service.
  + **Storage**:
    - Hot path: In-memory store (Redis) for quick access to last readings.
    - Cold path: Azure Data Lake for long-term historical analysis.
  + **Alerting**:
    - Rules engine evaluates thresholds per patient profile.
    - Send notifications via SMS, push, or email using Notification Service.
* **Challenges:**
  + Device data reliability (retry logic, dead-letter queue).
  + HIPAA/GDPR compliance for PII.

### **3. Design a Multi-Tenant SaaS Healthcare Appointment Booking Platform**

**Question:**  
How would you design a multi-tenant healthcare appointment booking platform that supports hundreds of clinics and patients?

**Detailed Answer:**

* **Architecture:**
  + **Tenant Isolation**: Separate DB per tenant or shared DB with tenant ID column (depending on scale & cost).
  + **Authentication**: Azure AD B2C for patient-facing, Azure AD for clinic-facing.
  + **Microservices**: Appointment Service, Clinic Management Service, Notification Service, Billing Service.
  + **Calendar Sync**: Use background workers to integrate with Google/Outlook APIs.
  + **Concurrency Control**:
    - Pessimistic locking for slot reservation to avoid double-booking.
  + **Scalability**: Horizontal scaling of Appointment Service using stateless containers.

## **Part 2 — Synchronous Inter-Service Communication in C#**

### **Why Synchronous?**

* Used when **real-time response** is needed (e.g., Payment Service must immediately confirm before Order Service completes).
* Typically implemented with **HTTP/REST** or **gRPC**.
* Risks: Higher latency, potential cascading failures if one service is slow.

### **Example: Order Service → Payment Service (Sync Call)**

**Architecture Flow:**

1. Order Service receives request → validates → calls Payment Service via HTTP.
2. Payment Service processes → returns status.
3. Order Service updates DB based on payment result.

**C# Implementation:**

#### **Payment Service (Receiver)**

[ApiController]

[Route("api/[controller]")]

public class PaymentController : ControllerBase

{

[HttpPost("process")]

public async Task<IActionResult> ProcessPayment([FromBody] PaymentRequest request)

{

// Simulate payment processing

await Task.Delay(500); // Simulated external API call

if (request.Amount <= 0)

return BadRequest(new { Message = "Invalid payment amount" });

return Ok(new PaymentResponse

{

TransactionId = Guid.NewGuid().ToString(),

Status = "Success"

});

}

}

public record PaymentRequest(string OrderId, decimal Amount, string PaymentMethod);

public record PaymentResponse(string TransactionId, string Status);

#### **Order Service (Caller)**

public class PaymentClient

{

private readonly HttpClient \_httpClient;

public PaymentClient(HttpClient httpClient)

{

\_httpClient = httpClient;

}

public async Task<PaymentResponse?> ProcessPaymentAsync(PaymentRequest request)

{

var response = await \_httpClient.PostAsJsonAsync("https://payment-service/api/payment/process", request);

if (response.IsSuccessStatusCode)

{

return await response.Content.ReadFromJsonAsync<PaymentResponse>();

}

// Handle error gracefully

var error = await response.Content.ReadAsStringAsync();

throw new Exception($"Payment failed: {error}");

}

}

#### **Startup Registration (Order Service)**

builder.Services.AddHttpClient<PaymentClient>(client =>

{

client.Timeout = TimeSpan.FromSeconds(5); // Avoid hanging

});

#### **Order Processing Example**

public class OrderProcessor

{

private readonly PaymentClient \_paymentClient;

public OrderProcessor(PaymentClient paymentClient)

{

\_paymentClient = paymentClient;

}

public async Task PlaceOrderAsync(Order order)

{

var paymentRequest = new PaymentRequest(order.Id, order.TotalAmount, "CreditCard");

var paymentResponse = await \_paymentClient.ProcessPaymentAsync(paymentRequest);

if (paymentResponse?.Status == "Success")

{

Console.WriteLine($"Order {order.Id} placed successfully. Transaction ID: {paymentResponse.TransactionId}");

}

else

{

Console.WriteLine($"Order {order.Id} failed during payment.");

}

}

}

public record Order(string Id, decimal TotalAmount);

### **Key Considerations in Sync Communication**

1. **Timeouts** – Prevent one slow service from blocking the caller.
2. **Circuit Breakers** – Use Polly in C# to prevent repeated calls to a failing service.
3. **Retries** – For transient network errors, but avoid retrying on permanent failures.
4. **Load Balancing** – Client should call service via a gateway (e.g., Azure API Management).
5. **Fallbacks** – Gracefully degrade if service is unavailable.

If you want, I can **extend this** with:

* **gRPC implementation** for better performance.
* **Polly retry + circuit breaker integration** in the Order Service.
* A **full sequence diagram** showing sync inter-service flow.

Do you want me to go ahead and prepare that extended version so you have a complete **enterprise-ready sync inter-service template**?

Alright — I’ll break this into **two parts** so you get both what you asked:

## **Part 1 — Mid-Level & High-Level System Design Hard Interview Questions (With Detailed Answers)**

I’ll expand each with **clear reasoning, trade-offs, and diagrams-in-words style explanations**.

### **Mid-Level System Design Questions**

**1. How would you design a payment gateway integration for an e-commerce platform?**

**Answer:**

* **Requirements:**
  + Multiple payment providers (Stripe, PayPal, Razorpay)
  + PCI compliance
  + Fraud detection
  + Retries on failure

**Design Approach:**

* Use **Adapter Pattern** to standardize payment provider APIs.
* **Core Services:**
  1. **Payment Orchestrator Service** → Determines provider and routes request.
  2. **Provider Adapter** → Wraps third-party SDK/API.
  3. **Transaction Logging Service** → Stores transaction history in SQL.
* **Sync/Async Flow:**
  1. Synchronous: Payment initiation.
  2. Asynchronous: Refunds, status updates via webhooks.

**Trade-offs:**

* **Pros:** Scalability, plug-and-play for new providers.
* **Cons:** More moving parts, increased latency with orchestration.

**C# Snippet:**

public interface IPaymentProvider

{

PaymentResult ProcessPayment(PaymentRequest request);

}

public class StripePaymentProvider : IPaymentProvider

{

public PaymentResult ProcessPayment(PaymentRequest request)

{

// Call Stripe API

return new PaymentResult { Success = true, TransactionId = "TX123" };

}

}

public class PaymentService

{

private readonly IPaymentProvider \_provider;

public PaymentService(IPaymentProvider provider) => \_provider = provider;

public PaymentResult Pay(PaymentRequest request) => \_provider.ProcessPayment(request);

}

**2. Design a rate-limiting mechanism for an API gateway.**

**Answer:**

* **Requirement:** Limit requests per user/IP to prevent abuse.
* **Approach:**
  + Token bucket or leaky bucket algorithm.
  + Store counters in **Redis** for fast access.
  + Enforce at API Gateway level.

**Trade-offs:**

* **In-memory (local):** Faster, but not distributed.
* **Redis-based:** Slightly slower, but works across multiple API gateway nodes.

**C# Token Bucket Example (Distributed):**

public class RateLimiter

{

private readonly IDatabase \_redis;

private readonly int \_maxRequests;

private readonly TimeSpan \_window;

public RateLimiter(IDatabase redis, int maxRequests, TimeSpan window)

{

\_redis = redis;

\_maxRequests = maxRequests;

\_window = window;

}

public bool AllowRequest(string userId)

{

var key = $"rate\_limit:{userId}";

var count = \_redis.StringIncrement(key);

if (count == 1) \_redis.KeyExpire(key, \_window);

return count <= \_maxRequests;

}

}

**3. How would you design a notification service that supports email, SMS, and push notifications?**

**Answer:**

* **Architecture:**
  + **Notification Orchestrator Service**
  + Channel-specific services: EmailService, SmsService, PushService.
  + Use **Message Queue** for async sending (RabbitMQ/Azure Service Bus).
* **Pattern:** Strategy Pattern for selecting channels.

**Trade-offs:**

* **Synchronous:** Faster feedback but blocks API.
* **Asynchronous:** Better for scale, but requires monitoring queue health.

**C# Strategy Pattern Example:**

public interface INotificationChannel

{

void Send(string to, string message);

}

public class EmailChannel : INotificationChannel

{

public void Send(string to, string message) => Console.WriteLine($"Email sent to {to}: {message}");

}

public class NotificationService

{

private readonly INotificationChannel \_channel;

public NotificationService(INotificationChannel channel) => \_channel = channel;

public void Notify(string to, string message) => \_channel.Send(to, message);

}

I can give you **10 more of these expanded** mid-level and **10 expanded high-level** designs in the same style.

## **Part 2 — Synchronous Inter-Service Communication (Full Implementation)**

### **When to Use**

* **Use when:** Real-time request/response is needed.
* **Avoid when:** High latency tolerance or massive scaling required → prefer async.

### **Common Implementations**

1. **REST over HTTP (Most Common)**
2. **gRPC (Binary, faster)**
3. **GraphQL Federation**
4. **WCF/Named Pipes** (legacy in .NET)

### **Detailed Example — REST Synchronous Call Between Two Services**

Scenario:

* **OrderService** calls **InventoryService** synchronously to reserve stock.

**OrderService → InventoryService HTTP Call**

public class InventoryClient

{

private readonly HttpClient \_httpClient;

public InventoryClient(HttpClient httpClient) => \_httpClient = httpClient;

public async Task<bool> ReserveStockAsync(string productId, int quantity)

{

var response = await \_httpClient.PostAsJsonAsync("/api/inventory/reserve", new

{

ProductId = productId,

Quantity = quantity

});

return response.IsSuccessStatusCode;

}

}

**Startup Registration (OrderService)**

services.AddHttpClient<InventoryClient>(client =>

{

client.BaseAddress = new Uri("https://inventoryservice");

});

**InventoryService Endpoint**

[ApiController]

[Route("api/inventory")]

public class InventoryController : ControllerBase

{

[HttpPost("reserve")]

public IActionResult Reserve([FromBody] ReserveRequest request)

{

// Validate and reserve stock in DB

return Ok(new { Message = "Stock reserved" });

}

}

public record ReserveRequest(string ProductId, int Quantity);

### **Best Practices for Sync Inter-Service Communication**

* Use **circuit breaker** (Polly library in C#) to prevent cascading failures.
* Apply **timeouts** to avoid hanging requests.
* Log correlation IDs for tracing.
* Use **DTO contracts** for versioning.

**Circuit Breaker Example (Polly):**

services.AddHttpClient<InventoryClient>()

.AddTransientHttpErrorPolicy(p => p.CircuitBreakerAsync(3, TimeSpan.FromSeconds(30)));

If you want, I can now **extend the mid-level & high-level Q&A list to a full 20 each** with the same depth and also **add a gRPC-based inter-service sync example** so you cover both REST and binary RPC approaches.

Do you want me to continue with that?