Excellent. Here's the **complete, architect-level preparation guide** for:

## ✅ PART 1: System Design – All Concepts, Scenarios & Architect-Level Q&A

### 🔷 SECTION A: Core Concepts in System Design

| **Concept** | **Description** | **Example Scenario** |
| --- | --- | --- |
| **1. Load Balancing** | Distribute traffic across servers | Horizontal scaling of an e-commerce platform |
| **2. Caching** | Store frequently accessed data | Product catalog cache in Redis |
| **3. CDN (Content Delivery Network)** | Distribute content geographically | Image & video delivery for Instagram |
| **4. Database Scaling** | Vertical vs. horizontal, read replicas, sharding | Sharded MongoDB for user profiles |
| **5. Consistency Models** | Strong, eventual, causal | Eventual consistency for news feed |
| **6. CAP Theorem** | Pick 2 of Consistency, Availability, Partition Tolerance | DynamoDB prioritizes AP |
| **7. Data Partitioning** | Divide data across DBs/nodes | User ID-based partitioning |
| **8. Rate Limiting** | Prevent abuse | API rate limits using token bucket |
| **9. Throttling** | Degrade service under load | Limit image uploads per user |
| **10. Message Queues** | Async processing | Order dispatch with RabbitMQ |
| **11. Event-Driven Architecture** | Services publish/subscribe to events | Microservices communication using Kafka |
| **12. Polling vs Webhooks vs WebSockets** | Push-based vs pull-based models | Real-time stock ticker (WebSockets) |
| **13. Authentication vs Authorization** | Identity vs access control | OAuth 2.0 for login, roles for access |
| **14. Circuit Breaker** | Avoid repeated failures | Polly in .NET to isolate failing DB |
| **15. Observability** | Logs, metrics, tracing | Application Insights in Azure |
| **16. Failover and Redundancy** | Backup systems in place | Active-passive setup for DB |
| **17. Distributed Transactions** | Saga pattern, 2PC | Payment flow across services |
| **18. Idempotency** | Safe re-processing | Payment retry safety key |
| **19. Backpressure** | Prevent downstream overload | Reactive Streams |
| **20. API Gateway** | Entry point, routing, auth | Azure API Management / Kong |
| **21. Microservice Communication** | Sync (REST/gRPC) vs Async (MQ/Kafka) | Inventory ↔ Order service |
| **22. Horizontal vs Vertical Scaling** | Add machines vs upgrade machine | Scale web app by adding containers |
| **23. Logging & Monitoring** | Track and alert | Serilog + Seq + Grafana stack |
| **24. Blue-Green / Canary Deployment** | Deployment strategies | Staged rollout to 10% of users |

### 🔷 SECTION B: Common System Design Scenarios with Architect-Grade Design

#### 📌 1. **URL Shortener**

* Concepts: Hashing, DB sharding, base62 encoding, cache
* Storage: Redis for cache, SQL/NoSQL for DB
* Handling collisions and analytics

#### 📌 2. **Design Twitter Feed**

* Follower graph, fan-out on write vs read
* Caching hot tweets
* Real-time updates with WebSockets

#### 📌 3. **E-Commerce System**

* Services: Auth, product catalog, cart, order, payment
* Inventory sync across multiple warehouses
* CQRS for checkout logic

#### 📌 4. **Real-Time Chat App**

* Persistent WebSockets
* Message queue for delivery
* Store chat history in NoSQL (MongoDB)

#### 📌 5. **Design Ride Booking App (e.g., Uber)**

* Location indexing (GeoHash)
* Matchmaking system
* Surge pricing logic

#### 📌 6. **Payment System**

* Idempotent payment APIs
* Fraud detection with AI
* Reconciliation and settlements

#### 📌 7. **Rate Limiter**

* Sliding window, token bucket algorithms
* Redis as store
* API Gateway integration

## ✅ PART 2: Design Patterns – Full Coverage with Scenarios & C# Code

### 🟨 CREATIONAL PATTERNS

| **Pattern** | **Description** | **Use Case** |
| --- | --- | --- |
| **Singleton** | One shared instance | Logging, Configuration |
| **Factory Method** | Create objects at runtime | Payment processor |
| **Abstract Factory** | Grouped families of objects | UI components (Dark vs Light Theme) |
| **Builder** | Step-by-step object construction | Building complex Order object |
| **Prototype** | Clone existing objects | Object snapshots in memory (game states) |

#### ✅ Example: **Builder Pattern**

public class OrderBuilder

{

private Order \_order = new();

public OrderBuilder WithItem(string item) { \_order.Items.Add(item); return this; }

public OrderBuilder WithCustomer(string name) { \_order.Customer = name; return this; }

public Order Build() => \_order;

}

### 🟧 STRUCTURAL PATTERNS

| **Pattern** | **Description** | **Use Case** |
| --- | --- | --- |
| **Adapter** | Convert one interface to another | Integrate legacy APIs |
| **Bridge** | Decouple abstraction from implementation | Shape → Draw API (SVG/Canvas) |
| **Composite** | Tree structure of objects | File system, menu hierarchies |
| **Decorator** | Add responsibilities at runtime | Logging levels |
| **Facade** | Simplify subsystem interface | Simplified DB API |
| **Flyweight** | Memory sharing for similar objects | Character glyphs in a text editor |
| **Proxy** | Add access control or lazy load | Virtual proxy for expensive object |

#### ✅ Example: **Adapter Pattern**

public interface INewLogger { void LogInfo(string msg); }

public class LegacyLogger { public void Write(string message) => Console.WriteLine(message); }

public class LoggerAdapter : INewLogger

{

private LegacyLogger \_legacy = new();

public void LogInfo(string msg) => \_legacy.Write("[INFO] " + msg);

}

### 🟩 BEHAVIORAL PATTERNS

| **Pattern** | **Description** | **Use Case** |
| --- | --- | --- |
| **Strategy** | Switchable algorithms | Sorting strategies |
| **Observer** | Publish-subscribe model | Eventing (Order placed → Email/SMS) |
| **Command** | Encapsulate actions | Undo/Redo functionality |
| **Chain of Responsibility** | Multiple handlers for a request | Support ticket escalation |
| **Mediator** | Central controller for components | UI dialog box coordination |
| **Memento** | Save/restore state | Text editor history |
| **Interpreter** | Language grammar parsing | Calculator DSL |
| **State** | Object behavior based on state | Traffic light controller |
| **Template Method** | Common algorithm skeleton | Report generation |
| **Visitor** | Operate on different types | AST traversal |

#### ✅ Example: **Command Pattern**

public interface ICommand { void Execute(); }

public class SaveCommand : ICommand

{

public void Execute() => Console.WriteLine("Saved!");

}

public class Button

{

private ICommand \_command;

public Button(ICommand cmd) => \_command = cmd;

public void Click() => \_command.Execute();

}

## 🔁 BONUS: Design Patterns Grouped with Use Cases

| **Pattern Type** | **Patterns** |
| --- | --- |
| **Creation** | Singleton, Factory, Abstract Factory, Builder, Prototype |
| **Structural** | Adapter, Decorator, Composite, Facade, Bridge, Proxy, Flyweight |
| **Behavioral** | Observer, Strategy, Command, Mediator, Chain of Responsibility, State, Template, Visitor, Memento, Interpreter |

## 📦 NEXT STEPS (Let Me Know)

Would you like me to now:

1. 🔖 Generate **Flashcards (MOQ format)** for all design patterns + system design?
2. 🧪 Create **Mock Interview (QA style)** based on these?
3. 🧾 Export all this as a **single PDF**?

Just reply with what format you want next, and I’ll generate accordingly.

Perfect! Below is a **Mock Interview** for a **Technical Architect / Tech Lead** role, covering:

* System Design Concepts
* Real-World Scenarios
* All 23 Design Patterns (Creational, Structural, Behavioral)
* C# Code-based Deep Dives

# 🎯 Mock Interview – System Design + Design Patterns

**Role**: Technical Architect  
**Style**: Q&A with Scenarios  
**Depth**: Architect-level

## 🔹 SECTION 1: SYSTEM DESIGN INTERVIEW

### **Q1. How would you design a real-time chat application like WhatsApp or Slack?**

**Follow-up questions**:

* How would you handle message delivery guarantees?
* What protocol would you use for real-time communication?
* How would you scale WebSocket connections?

**Expected Answer**:

* Use WebSockets for real-time bi-directional communication.
* Store chat messages in NoSQL (MongoDB/CosmosDB) for flexibility.
* Use Redis pub/sub for message fan-out.
* Scale with load balancers supporting sticky sessions.
* Apply message queues (like Kafka) to buffer chat traffic.

### **Q2. How would you design a URL shortener system (like bit.ly)?**

**Follow-up**:

* How do you avoid hash collisions?
* How would you scale DB for billions of URLs?

**Answer Outline**:

* Base62 encoding for short URLs.
* Store mapping in a sharded NoSQL DB (e.g., DynamoDB).
* Use Redis for hot-cache of popular URLs.
* Rate limit for abusive shorteners.

### **Q3. Design a Rate Limiter for an API Gateway**

**Follow-up**:

* What algorithm would you choose?
* How to ensure it works in a distributed environment?

**Answer**:

* Use token bucket or leaky bucket algorithm.
* Store token counters in Redis for speed + atomic operations.
* Use Lua scripting in Redis for atomicity.
* Optionally integrate in Azure API Management for global throttling.

### **Q4. How would you ensure consistency in a distributed system?**

**Answer**:

* Choose consistency level based on use-case (strong/eventual).
* Use techniques like quorum reads/writes.
* Use distributed locks (e.g., RedLock with Redis).
* For workflows: use Saga pattern instead of 2PC.

## 🔹 SECTION 2: DESIGN PATTERN QUESTIONS

### 🔸 CREATIONAL PATTERNS

#### **Q1. Explain the Singleton Pattern. Where should you use and not use it?**

**Scenario**: Global configuration manager or logger.  
**C# Code**:

public sealed class AppConfig

{

private static readonly Lazy<AppConfig> \_instance = new(() => new AppConfig());

public static AppConfig Instance => \_instance.Value;

private AppConfig() { }

}

**Caution**: Avoid for testability and tightly coupled classes.

#### **Q2. Explain Factory Method Pattern with use case.**

**Scenario**: You have different payment types: PayPal, Stripe, UPI.

public interface IPayment { void Pay(); }

public class PayPal : IPayment { public void Pay() => Console.WriteLine("PayPal Paid"); }

public class PaymentFactory

{

public static IPayment GetPayment(string method) =>

method switch {

"PayPal" => new PayPal(),

\_ => throw new Exception("Unsupported")

};

}

### 🔸 STRUCTURAL PATTERNS

#### **Q3. Explain Decorator Pattern and where to apply it.**

**Scenario**: Add runtime behavior to Logger.

public interface ILogger { void Log(string msg); }

public class BasicLogger : ILogger { public void Log(string msg) => Console.WriteLine(msg); }

public class ErrorLogger : ILogger

{

private readonly ILogger \_inner;

public ErrorLogger(ILogger inner) => \_inner = inner;

public void Log(string msg) => \_inner.Log("[ERROR] " + msg);

}

**Use-case**: Logging, Notification pipelines, Encryption wrappers.

#### **Q4. What is Facade Pattern?**

**Scenario**: Simplify access to payment, shipping, inventory subsystems.

public class OrderFacade

{

private readonly Payment \_payment = new();

private readonly Inventory \_inventory = new();

public void PlaceOrder() { \_inventory.Check(); \_payment.Process(); }

}

### 🔸 BEHAVIORAL PATTERNS

#### **Q5. Explain Observer Pattern**

**Scenario**: Notify multiple systems when an order is placed.

public interface IObserver { void Update(string msg); }

public class EmailService : IObserver { public void Update(string msg) => Console.WriteLine("Email: " + msg); }

public class OrderPublisher

{

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

public void Subscribe(IObserver obs) => \_subscribers.Add(obs);

public void Notify(string msg) => \_subscribers.ForEach(o => o.Update(msg));

}

#### **Q6. Strategy Pattern Example**

**Scenario**: Different sorting strategies.

public interface ISort { void Sort(List<int> data); }

public class QuickSort : ISort { public void Sort(List<int> data) => Console.WriteLine("QuickSort"); }

public class SortContext

{

private ISort \_strategy;

public void Set(ISort strategy) => \_strategy = strategy;

public void Execute(List<int> data) => \_strategy.Sort(data);

}

#### **Q7. Chain of Responsibility Pattern Example**

**Scenario**: Request passes through multiple validations

public abstract class Handler

{

protected Handler Next;

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

public abstract void Handle(string req);

}

## 🔹 SECTION 3: ADVANCED PATTERNS (Architect Level)

#### **Q8. Explain CQRS and Event Sourcing with .NET example**

**CQRS**: Separate read/write models for performance and scalability.  
**Event Sourcing**: Persist state changes as events instead of current state.

**Use-case**: Order management system.

// Write model

public class CreateOrderCommand { public int Id; public string Product; }

// Read model

public class OrderSummaryDto { public int Id; public string Product; }

#### **Q9. How do you design for eventual consistency in microservices?**

* Use Outbox Pattern to save events alongside DB changes
* Publish to Kafka/RabbitMQ asynchronously
* Use retry & dead-letter queues
* Accept temporary inconsistency in read models

## 📌 End of Mock Interview Set 1

Would you like:

1. Mock Interview **Set 2** (More scenario-based system designs)?
2. Flashcards for revision?
3. A consolidated **PDF/Notion doc**?

Let me know — and I’ll generate the next format instantly.

Great — here’s **Mock Interview Set 2**, continuing with deep, scenario-driven **System Design** and **Design Pattern** interview questions tailored for a **Technical Architect** role.

# 🧠 Mock Interview – Set 2

**Theme**: Architecture Scenarios, Scaling, GoF Patterns in Real Projects  
**Focus**: Reasoning, Design Decisions, Trade-offs, Real-World Applications

## 🔹 SECTION 1: SYSTEM DESIGN SCENARIOS – ADVANCED

### **Q1. Design a Global E-Commerce Platform (Amazon-like)**

**Follow-up Questions**:

* How do you ensure availability during Black Friday?
* How do you handle product inventory across regions?
* How do you manage catalog search performance?

**Expected Answer**:

* **Traffic Handling**: Use CDN for static content, load balancers, autoscaling.
* **Search**: Use Elasticsearch for catalog indexing.
* **DB**: Sharded catalog DB, regional replicas.
* **Inventory**: Event-driven model for stock updates (Kafka).
* **Write Model**: Use CQRS for checkout.
* **Payments**: Use SAGA to coordinate transactions.
* **Globalization**: Geo-DNS routing, language localization layer.

### **Q2. Design a Notification System for Multiple Channels (Email, SMS, Push)**

**Key Concepts**: Observer pattern, retries, template rendering, queueing

**Expected Answer**:

* Notification Orchestrator → dispatch to channel workers
* Each channel has its own queue + retry logic
* Template engine for formatting
* Store notifications in CosmosDB or MongoDB
* Use exponential backoff for failed messages
* Monitor via dashboard + alerting

### **Q3. Design a Video Streaming Platform (like YouTube)**

**Key Points**:

* Use CDN for video delivery
* Transcoding pipeline using blob triggers and Azure Media Services
* Use separate read/write stores for metadata and view count
* View tracking via log collectors (Kafka → Aggregator → Storage)
* Comments section → paginated + sorted by like count

## 🔹 SECTION 2: SYSTEM DESIGN CONCEPT – DEEP DIVE QUESTIONS

### **Q4. What’s the difference between Horizontal and Vertical Scaling?**

**Answer**:

* Horizontal: Add more nodes (web servers, DB replicas)
* Vertical: Add more CPU/RAM to existing node
* Horizontally scalable systems are better for resilience and cloud-native patterns.

### **Q5. What are the different consistency models in distributed systems?**

| **Model** | **Description** | **Example Use-Case** |
| --- | --- | --- |
| Strong | Every read sees latest write | Banking |
| Eventual | Reads become consistent over time | Social Media Feed |
| Causal | Related operations maintain order | Collaborative Docs |
| Read-Your-Writes | A user sees their own writes immediately | Profile updates |

### **Q6. How would you handle long-running transactions across microservices?**

**Answer**:

* Use **SAGA pattern** with compensating transactions.
* Use state machines or orchestrators (e.g., Azure Durable Functions, Netflix Conductor).
* Avoid distributed locks and 2PC.

## 🔹 SECTION 3: DESIGN PATTERNS – ADVANCED (With C# Code)

### 🔸 CREATIONAL

#### **Q7. Abstract Factory Pattern – UI Component Factory**

**Scenario**: Dark Theme vs Light Theme Components

public interface IButton { void Render(); }

public class DarkButton : IButton { public void Render() => Console.WriteLine("Dark Button"); }

public interface IUIFactory

{

IButton CreateButton();

}

public class DarkUIFactory : IUIFactory

{

public IButton CreateButton() => new DarkButton();

}

#### **Q8. Prototype Pattern – Game Character Cloning**

public class GameCharacter : ICloneable

{

public int HP { get; set; }

public object Clone() => this.MemberwiseClone();

}

### 🔸 STRUCTURAL

#### **Q9. Composite Pattern – File Explorer Tree**

public interface IFileSystem

{

void Display();

}

public class File : IFileSystem

{

public string Name;

public File(string name) => Name = name;

public void Display() => Console.WriteLine("File: " + Name);

}

public class Folder : IFileSystem

{

public string Name;

public List<IFileSystem> Children = new();

public Folder(string name) => Name = name;

public void Add(IFileSystem item) => Children.Add(item);

public void Display()

{

Console.WriteLine("Folder: " + Name);

foreach (var child in Children) child.Display();

}

}

#### **Q10. Proxy Pattern – Lazy Image Loading**

public interface IImage { void Show(); }

public class RealImage : IImage

{

private string \_filename;

public RealImage(string filename) { \_filename = filename; Load(); }

private void Load() => Console.WriteLine("Loading " + \_filename);

public void Show() => Console.WriteLine("Showing " + \_filename);

}

public class ProxyImage : IImage

{

private RealImage \_realImage;

private string \_filename;

public ProxyImage(string filename) => \_filename = filename;

public void Show()

{

\_realImage ??= new RealImage(\_filename);

\_realImage.Show();

}

}

### 🔸 BEHAVIORAL

#### **Q11. Command Pattern – UI Button Trigger**

public interface ICommand { void Execute(); }

public class PrintCommand : ICommand { public void Execute() => Console.WriteLine("Printed!"); }

public class Button

{

private readonly ICommand \_command;

public Button(ICommand command) => \_command = command;

public void Click() => \_command.Execute();

}

#### **Q12. State Pattern – Traffic Light**

public interface ILightState { void Switch(); }

public class Red : ILightState

{

public void Switch() => Console.WriteLine("Switch to Green");

}

#### **Q13. Visitor Pattern – Report Generator**

public interface IElement { void Accept(IVisitor visitor); }

public class Order : IElement

{

public void Accept(IVisitor visitor) => visitor.Visit(this);

}

public interface IVisitor { void Visit(Order order); }

public class ExcelVisitor : IVisitor

{

public void Visit(Order order) => Console.WriteLine("Exporting to Excel");

}

## 🧪 Wrap-Up Suggestions

Would you like:

1. ✅ **Mock Interview Set 3** – Cloud Architecture, Microservice Deep Dives, Event Sourcing
2. 🔖 Flashcards (MOQ format) for quick revision?
3. 📘 A Notion/Markdown/PDF export of all mock interview content so far?

Let me know how you want to continue, and I’ll proceed accordingly.

Excellent. Here's your **Mock Interview Set 3**, focusing on:

✅ Cloud-Native & Microservice Architectures  
✅ Event-Driven Design  
✅ Advanced System Trade-offs  
✅ High-Level Enterprise-Scale Scenarios  
✅ Deep Behavioral + Structural Patterns Usage in Real Systems

# 🧠 Mock Interview – Set 3 (Architect-Level)

**Theme**: Cloud-Native Design, Distributed Systems, High-Scale Real-World Problems

## 🔷 SECTION 1: CLOUD SYSTEM DESIGN SCENARIOS

### **Q1. How would you design a Cloud-Native Microservices-Based Application for an Online Learning Platform?**

**Follow-ups**:

* What services would you isolate?
* How do you manage inter-service communication?
* How do you scale APIs?

**Expected Answer**:

* Services: Auth, User, Course, Payment, Notification, Progress Tracking
* Use Azure API Management or AWS API Gateway
* Communication: REST + gRPC (sync), Event Bus (async)
* Saga pattern for course purchase
* Use containers with orchestration (AKS / EKS)
* Circuit breaker (Polly), bulkheads, retries
* CI/CD using GitHub Actions or Azure DevOps

### **Q2. How would you design a File Storage and Sharing System (e.g., Dropbox, Google Drive)?**

**Expected Answer**:

* Metadata in SQL (PostgreSQL / CosmosDB)
* File chunks stored in Azure Blob / S3
* Upload pipeline: client → chunker → async processing → virus scan
* Use pre-signed URLs for secure file access
* Real-time collaboration via WebSockets
* Versioning and audit logs

### **Q3. Design a Real-Time Analytics Dashboard for Millions of IoT Devices**

**Follow-ups**:

* How do you ingest high-velocity data?
* What stores do you choose for real-time vs batch?

**Answer**:

* Ingestion: Azure Event Hubs / Kafka
* Stream processing: Azure Stream Analytics / Flink
* Hot path (real-time): InfluxDB or Redis TimeSeries
* Cold path: Azure Data Lake or S3
* Display: SignalR + React + WebSocket updates
* Scale out with partitioned consumers

## 🔷 SECTION 2: EVENT-DRIVEN ARCHITECTURE QUESTIONS

### **Q4. What are the benefits and pitfalls of Event-Driven Architecture?**

**Benefits**:

* Loose coupling, extensibility
* Asynchronous communication
* Better decoupling and resilience

**Challenges**:

* Event ordering, duplication, eventual consistency
* Debugging/tracing harder
* Requires schema evolution handling

### **Q5. How would you implement Event Sourcing + CQRS in a .NET App?**

**Answer**:

* Commands change the state → events are persisted (Event Store)
* Separate read models updated via projection
* Use MediatR for command dispatch
* Events published via Kafka
* Queries served via denormalized view (e.g., ElasticSearch, MongoDB)

## 🔷 SECTION 3: ENTERPRISE TRADE-OFF QUESTIONS

### **Q6. REST vs gRPC vs Message Queue – When to use what?**

| **Protocol** | **Use When** | **Trade-offs** |
| --- | --- | --- |
| REST | Client-server, easy integration | Verbose, not performant for large data |
| gRPC | Internal microservice calls, performance needed | Requires .proto, more setup |
| Message Queue | Decoupled, async communication | Eventual consistency, retry needed |

### **Q7. How do you handle configuration across microservices?**

**Answer**:

* Use centralized config system: Azure App Config / Spring Cloud Config
* Secrets in Azure Key Vault
* Inject via environment variables or mounted volumes
* Feature flags using LaunchDarkly or Azure Feature Manager

## 🔷 SECTION 4: ADVANCED DESIGN PATTERN USAGE

### 🔸 Behavioral Patterns

#### **Q8. Mediator Pattern in a UI Dialog System**

**Scenario**: Avoid direct communication between form controls (Textbox, Button)

public interface IMediator { void Notify(object sender, string ev); }

public class DialogMediator : IMediator

{

public void Notify(object sender, string ev)

{

if (ev == "text\_changed")

Console.WriteLine("Enable submit button");

}

}

#### **Q9. Interpreter Pattern – Evaluate Expressions**

**Scenario**: Calculator or DSL parser

public interface IExpression { int Interpret(); }

public class Number : IExpression

{

private readonly int \_value;

public Number(int value) => \_value = value;

public int Interpret() => \_value;

}

public class Add : IExpression

{

private readonly IExpression \_left, \_right;

public Add(IExpression l, IExpression r) { \_left = l; \_right = r; }

public int Interpret() => \_left.Interpret() + \_right.Interpret();

}

### 🔸 Structural Patterns

#### **Q10. Flyweight Pattern – Memory-Efficient Object Pool**

**Scenario**: Text Editor stores thousands of characters

public class Character

{

public char Symbol { get; }

public string Font { get; }

public Character(char symbol, string font)

{

Symbol = symbol;

Font = font;

}

}

public class CharacterFactory

{

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

public Character Get(char symbol, string font)

{

var key = symbol + font;

if (!\_cache.ContainsKey(key))

\_cache[key] = new Character(symbol, font);

return \_cache[key];

}

}

## 🔷 SECTION 5: ARCHITECT-LEVEL SYSTEM INTEGRATION QUESTIONS

### **Q11. How would you design for multitenancy in SaaS?**

**Options**:

* Shared DB, tenant ID column (cheaper, complex isolation)
* Separate schema per tenant (isolation + moderate cost)
* Separate DB per tenant (expensive, highly isolated)

**Auth**: Use claims-based token to resolve tenant context

### **Q12. How would you ensure observability in a cloud-native app?**

**Answer**:

* Logging: Serilog + Seq
* Metrics: Prometheus / Azure Monitor
* Tracing: OpenTelemetry, Jaeger
* Alerting: PagerDuty, Azure Alerts
* Correlation ID passed through services

## ✅ End of Set 3

### Next Options:

1. ✅ **Set 4** – High-Level Architecture Design Round: Trade-offs, Diagrams, ADRs
2. 🔁 **Flashcards (MOQ)** for all 23 design patterns and core system concepts
3. 📘 A PDF export or Notion-friendly format for full interview guide

Let me know what format or direction you want next.

Perfect. Here's your **Mock Interview – Set 4**, tailored for **High-Level Architecture Design Rounds** — exactly what you’ll face in a **Technical Architect** or **Principal Engineer** interview at top tech firms.

# 🏗️ Mock Interview – Set 4: High-Level Architecture, Trade-Offs, ADRs

**Focus Areas**:

* End-to-End System Architecture
* Trade-off Discussions
* Technology Stack Decisions
* Architecture Decision Records (ADR)
* Integration and Deployment Strategies
* Security, Observability, Resilience

## 🔷 SECTION 1: END-TO-END ARCHITECTURE SCENARIOS

### **Q1. Design a Scalable Online Food Delivery Platform (like Swiggy, Uber Eats)**

**Follow-up Questions**:

* How would you split the services?
* How do you match orders to delivery agents?
* How do you deal with high traffic during dinner time?

**Expected Architectural Components**:

* Microservices: Auth, Restaurant, Menu, Order, Payment, Delivery, Notification
* Load balancer → API Gateway → Service Mesh (Istio or Azure YARP)
* Async communication via Kafka for order status
* Geo-location using Redis GeoIndex
* Search: Elasticsearch for restaurant lookups
* Database: PostgreSQL for orders, MongoDB for restaurant data
* Caching: Redis for menu caching
* Frontend: SPA using React + Azure CDN
* Deployment: Azure Kubernetes Service + Azure DevOps pipelines

### **Q2. Design Architecture for a Global Enterprise HRMS SaaS (multi-tenant)**

**Follow-ups**:

* How do you isolate tenants?
* How do you design feature toggles?
* How do you manage data access and RBAC?

**Key Decisions**:

* Multitenancy model: **Separate schema per tenant**
* Auth: Azure AD B2C with OpenID Connect
* Feature toggles via Azure App Config / LaunchDarkly
* Role-based access control using claims-based auth
* DB: SQL for core HR, Blob Storage for documents
* Tenant onboarding via automated CI/CD flow
* Monitoring per tenant using tags in Application Insights

## 🔷 SECTION 2: ARCHITECTURAL DECISION RECORDS (ADRs)

### **Q3. What is an ADR and why is it important?**

**Answer**: ADR (Architecture Decision Record) documents:

* A key decision
* The context around it
* The alternatives considered
* The pros/cons
* The final outcome

It's critical for:  
✅ Auditability  
✅ Developer onboarding  
✅ Decision traceability  
✅ Organizational learning

### **Q4. Example ADR: API Gateway Choice**

**Title**: Use Azure API Management over custom NGINX proxy  
**Context**:

* Need for central security, versioning, analytics  
  **Options**:
* Azure APIM
* Custom NGINX + Lua
* Ocelot (open source .NET gateway)

**Decision**: Chose Azure APIM due to:

* Built-in analytics
* OAuth2 support
* Policy-based throttling and rate-limiting
* Enterprise SLA

## 🔷 SECTION 3: ARCHITECTURE TRADE-OFF QUESTIONS

### **Q5. How do you choose between Monolith, Microservices, or Modular Monolith?**

| **Option** | **When to Use** | **Trade-offs** |
| --- | --- | --- |
| Monolith | Startup, simple domain | Easy deploy, tight coupling |
| Microservices | Large org, independent domains | Scalability, complex ops |
| Modular Monolith | Transition phase | Code modularity + deploy as monolith |

### **Q6. How do you decide between SQL and NoSQL?**

| **Criteria** | **SQL (PostgreSQL)** | **NoSQL (CosmosDB, MongoDB)** |
| --- | --- | --- |
| Relations | Strong | Weak/none |
| Schema | Fixed | Flexible |
| Transactions | Strong (ACID) | Weak / Eventual |
| Use-case | Orders, Payments | Logs, Events, Profiles |

### **Q7. How do you handle secrets and configuration in cloud architecture?**

**Answer**:

* Store secrets in Azure Key Vault
* App configuration via Azure App Config
* Rotate keys regularly
* Inject config via environment variables in containers
* Use managed identity for services (no hardcoded credentials)

## 🔷 SECTION 4: SECURITY, OBSERVABILITY, DEPLOYMENT

### **Q8. How would you secure APIs in a multi-service architecture?**

**Answer**:

* Use Azure API Management or OAuth2 via IdentityServer
* Token validation middleware
* Role claims in JWT
* Rate limiting + IP allow list
* Encrypt communication via TLS
* Use service mesh for internal mTLS

### **Q9. What’s your CI/CD architecture for multi-service deployment?**

**Answer**:

* GitHub Actions / Azure DevOps → Build container → Run unit + integration tests
* Push to ACR (Azure Container Registry)
* Deploy via Helm / Kustomize to AKS
* Canary deployments using deployment slots or flags
* Use ArgoCD or Flux for GitOps model
* Monitor via Grafana dashboards

### **Q10. How do you implement observability in a distributed system?**

**Answer**:

* **Logs**: Serilog / ELK / Azure Monitor
* **Metrics**: Prometheus / Azure Monitor Metrics
* **Tracing**: OpenTelemetry + Zipkin / Jaeger
* Pass **Correlation ID** through each service call
* Dashboards + alerts configured per SLA/KPI

## 🔷 SECTION 5: ARCHITECTURAL INTEGRATION CHALLENGES

### **Q11. How do you approach integrating with third-party services (e.g., payment gateways, CRMs)?**

**Answer**:

* Use Adapter pattern to wrap APIs
* Apply Circuit Breaker (Polly) for resilience
* Store request in outbox before sending
* Ensure idempotency using unique tokens
* Retry with exponential backoff
* Secure using signed webhook verification

### **Q12. How do you handle schema evolution in Event-Driven Architecture?**

**Answer**:

* Use schema registry (e.g., Confluent)
* Always add, never remove fields
* Version events: OrderCreated\_v2
* Handle backward compatibility in consumers
* Avoid tight coupling between publisher and subscribers

## ✅ END OF MOCK INTERVIEW SET 4

### ✅ You’re Now Prepared to Discuss:

* System-wide architecture trade-offs
* Deployment, config, and secrets at scale
* ADRs and governance in enterprise projects
* Cloud-native design patterns
* Integration with real-world systems

### ▶️ What Next?

Would you like:

1. **Set 5**: Domain-Driven Design (DDD), Bounded Contexts, Integration Patterns
2. Flashcards for revision
3. A master document (PDF or Notion-ready) with all sets

Let me know and I’ll generate accordingly.

Excellent choice! Here's **Mock Interview – Set 5**, focused on **Domain-Driven Design (DDD)**, **Integration Patterns**, and **Enterprise Architecture Principles**, all tailored for **Technical Architect / Tech Lead** interviews.

# 🧠 Mock Interview – Set 5: Domain-Driven Design, Integration, and Enterprise Patterns

## 🔷 SECTION 1: DOMAIN-DRIVEN DESIGN (DDD) – ARCHITECT LEVEL

### **Q1. What is Domain-Driven Design and when should you apply it?**

**Answer**: DDD is a software design approach that focuses on modeling software based on real-world business domains.

Use it when:

* Domain complexity is high
* Multiple teams work on different parts of the system
* You want to avoid "big ball of mud" design

Key concepts:

* Ubiquitous Language
* Bounded Contexts
* Aggregates
* Value Objects
* Domain Events
* Repositories

### **Q2. How do you define a Bounded Context? Can you give an example?**

**Answer**: A Bounded Context defines a clear boundary where a specific domain model applies.

**Example**: In an e-commerce system:

* Order context: deals with placing and managing orders
* Inventory context: deals with stock and warehouse
* Both may have a "Product" entity but with different attributes and meanings

Each Bounded Context can be a microservice.

### **Q3. Explain Aggregate Root and why it's important.**

**Answer**: An **Aggregate** is a cluster of domain objects treated as a single unit. The **Aggregate Root** is the only entry point to access or modify its data.

**Example**: In an Order aggregate:

* Order is the root
* OrderItems are internal entities
* All changes go through Order

It enforces transactional consistency within the aggregate.

### **Q4. What’s the role of a Repository in DDD?**

**Answer**: A Repository provides abstraction over data persistence, allowing the domain layer to remain persistence-agnostic.

public interface IOrderRepository

{

Task<Order> GetByIdAsync(Guid id);

Task SaveAsync(Order order);

}

### **Q5. How would you implement Value Objects in .NET?**

**Example**: Money as a Value Object

public record Money(decimal Amount, string Currency);

* Value objects are immutable
* Compared by value, not reference
* No identity

## 🔷 SECTION 2: INTEGRATION PATTERNS

### **Q6. What is the Outbox Pattern and why is it needed?**

**Answer**: Used to ensure atomicity between database writes and event publishing.

**Problem**: DB write succeeds, but event publish fails (or vice versa)

**Solution**:

* Store event in DB "Outbox" table with status = "pending"
* Background processor reads and sends to event bus (Kafka, etc.)
* Retry on failure

### **Q7. What is the Anti-Corruption Layer (ACL) in DDD?**

**Answer**: ACL is a pattern that protects your domain model from influence or changes in an external system or legacy model.

**Use**:

* Translate external models into your own
* Use adapters or facades to interface

### **Q8. How would you manage schema evolution in a message-based system?**

**Answer**:

* Use versioned event types (OrderCreatedV1, OrderCreatedV2)
* Use Protobuf or Avro schemas with a registry
* Consumers should be backward compatible
* Never remove fields — only add

### **Q9. How do you orchestrate a business process across services (e.g., order placement)?**

**Answer**: Two patterns:

* **Orchestration**: Central orchestrator controls flow (e.g., Durable Functions, Conductor)
* **Choreography**: Services react to events independently (Event-driven)

**Choreography Example**:

1. OrderPlaced → triggers PaymentRequested
2. PaymentCompleted → triggers ShippingInitiated

Use SAGA for compensation (e.g., cancel payment if shipping fails)

## 🔷 SECTION 3: ENTERPRISE ARCHITECTURE QUESTIONS

### **Q10. What are Enterprise Integration Patterns (EIP)?**

**Answer**: Patterns for integrating enterprise applications and services.

Key patterns:

* Message Bus
* Publish/Subscribe
* Request/Reply
* Content-Based Router
* Message Filter
* Message Translator
* Aggregator

**Example**:

* Azure Service Bus → implements pub/sub, dead-letter queues
* NServiceBus → supports retry, sagas, routing

### **Q11. How do you define and manage Service Contracts in Microservices?**

**Answer**:

* Use OpenAPI / Swagger to define REST contracts
* Use gRPC and Protobuf for strongly typed contracts
* Use versioning: /v1/orders, /v2/orders
* Use contract testing tools like Pact

### **Q12. What’s a canonical data model? Should you use one?**

**Answer**: A canonical model is a standard format for data exchanged between services.

✅ Pros:

* Simplifies transformations
* Acts as a common language

❌ Cons:

* Adds overhead
* Hard to evolve as domains change
* Can lead to a shared "god object"

Better alternative: Let each bounded context own its model + use adapters.

## 🔷 SECTION 4: IMPLEMENTING DDD IN .NET

### **Q13. Show a basic DDD-based architecture structure in .NET**

/src

/Domain

- Entities

- ValueObjects

- Events

- Interfaces (IRepository)

/Application

- UseCases

- Commands/Queries (CQRS)

- DTOs

/Infrastructure

- EF Core Repositories

- Email Service Adapter

/WebAPI

- Controllers

- DI and Middleware

### **Q14. How does Clean Architecture relate to DDD?**

**Answer**:

* Clean Architecture enforces separation of concerns (inner vs outer layers)
* DDD fits into the domain layer (core business logic)
* Infrastructure (data, email, etc.) is in outer layers
* Application layer handles orchestration between use cases and domain

## ✅ END OF SET 5

### 🧠 You Now Master:

* Domain modeling with Bounded Contexts, Aggregates, and Value Objects
* Event-driven integration with real patterns (Outbox, SAGA, ACL)
* Strategic architecture decisions for SaaS, enterprise, and microservices
* DDD implementation patterns in .NET + Clean Architecture

### 📘 What's Next?

Would you like:

1. ✅ **Set 6**: AI/ML System Design, Vector DBs, Prompt Architectures, Gen AI APIs
2. 🔖 Flashcards (MOQ) for **all 5 Sets** to revise faster
3. 📦 Export all content to **PDF/Markdown/Notion** format?

Let me know and I’ll generate it accordingly.