Thank you for the clarification. Here is a **complete, professional checklist** to guide you through **what questions to ask** and **how to analyze use cases and data points** before preparing an **ADR (Architecture Decision Record)** or doing **system design**.

# ✅ Architecture & System Design Pre-Analysis Checklist

🎯 Purpose: To collect all necessary **business requirement document,technical, functional, and non-functional inputs** required before starting ADR writing or system-level design.

## 🟢 Section 1: Business Context Understanding

### ❓ Questions to Ask:

* What is the **primary business problem** we're solving?
* What is the **business goal or objective** of this initiative?
* What happens if we **don’t solve** this problem?
* What is the **timeline or deadline** for delivery?
* Who are the **end users or consumers** of this system?
* Are there any **compliance, regulatory, or industry standards** to follow?

### 🔍 What to Analyze:

* Align decision/design with business value (e.g., revenue, user satisfaction, cost-cutting).
* Identify critical deadlines that may limit technical choices.
* Understand stakeholders’ priorities (speed, quality, cost, innovation, etc.).

## 🟢 Section 2: Functional Requirements (Use Case Analysis)

### ❓ Questions to Ask:

* What are the **core business capabilities** or features to be delivered?
* Can you describe **key use cases or user journeys**?
* Are there any **system integrations** required?
* What are the **edge cases or failure scenarios**?
* Do we need **multi-tenant** or role-based features?
* What is the expected **user interaction flow**?

### 🔍 What to Analyze:

* Translate use cases into modules or services.
* Identify CRUD operations, orchestration, API contracts, and user events.
* Evaluate inter-service communication needs (sync/async).

## 🟢 Section 3: Non-Functional Requirements (Quality Attributes)

### ❓ Questions to Ask:

* What is the required **performance or response time**?
* What is the expected **concurrency or load** (users/sec, requests/min)?
* What are the **availability/Uptime** targets? (e.g., 99.9%)
* What are the **security** needs? (e.g., AuthN/AuthZ, encryption, data masking)
* Is there a need for **audit trails or traceability**?
* Are there any **backup, DR, or failover** expectations?
* What is the **expected latency or throughput**?

### 🔍 What to Analyze:

* SLOs, SLAs, security compliance (e.g., GDPR, HIPAA).
* Sizing infrastructure based on throughput.
* Need for caching, rate limiting, or QoS controls.

## 🟢 Section 4: Data Considerations

### ❓ Questions to Ask:

* What **type of data** is processed (structured/unstructured)?
* What is the **volume of data** per day/month?
* Do we need **real-time**, near-real-time, or batch processing?
* What are the **data retention and archival** policies?
* Is the data **sensitive, regulated, or multi-region**?
* How frequently is the data **read, written, or modified**?

### 🔍 What to Analyze:

* Data model design (domain-driven vs flat).
* Storage choices (SQL vs NoSQL, object stores).
* Performance optimizations (indexes, sharding, partitioning).
* Backup and disaster recovery needs.

## 🟢 Section 5: Existing Architecture or System Landscape

### ❓ Questions to Ask:

* Are we building **from scratch** or enhancing/replacing an existing system?
* What **existing systems** or APIs do we need to integrate with?
* What are the **current pain points** in the legacy system?
* What tools or cloud platforms are already in use?
* Are there any **tech stack preferences or restrictions**?

### 🔍 What to Analyze:

* Reuse vs rebuild decisions.
* Interoperability and migration complexity.
* Cloud-native or hybrid design decisions.

## 🟢 Section 6: Constraints and Assumptions

### ❓ Questions to Ask:

* What is the **budget limit or cost ceiling**?
* What are the **time constraints** (MVP timeline, launch window)?
* What are the **technology constraints** (only Azure, no Kafka, etc.)?
* What are the **team skill levels** and available bandwidth?
* Any **organizational policies** to be considered?

### 🔍 What to Analyze:

* Narrow down feasible options (e.g., PaaS vs container vs serverless).
* Reduce operational complexity by aligning with existing practices.
* Avoid high-risk tech if the team lacks experience.

## 🟢 Section 7: Stakeholders and Responsibilities

### ❓ Questions to Ask:

* Who are the **business owners** or sponsors?
* Who are the **primary consumers** (internal/external teams)?
* Who is responsible for **approving architecture**?
* Who will **support and monitor** the system in production?

### 🔍 What to Analyze:

* Communication flow for reviews, demos, validations.
* Roles and responsibilities for design, development, support.

## 🟢 Section 8: Success Metrics and Risks

### ❓ Questions to Ask:

* How will we **measure success** (KPIs)?
* What are the **biggest risks** or unknowns?
* What are the **fallbacks** if the decision doesn’t work?
* Are we allowed to do **spikes or PoCs** before finalizing design?

### 🔍 What to Analyze:

* Use this to justify decision trade-offs.
* Decide if reversible decisions are preferable for experimentation.

## ✅ Final Checklist Before Designing or Writing ADR

| **Area** | **Completed (✓)** | **Notes or Reference** |
| --- | --- | --- |
| Business objective understood |  |  |
| Use cases captured |  |  |
| Non-functional requirements listed |  |  |
| Data model and volume analyzed |  |  |
| Integration points identified |  |  |
| Constraints and assumptions known |  |  |
| Stakeholders mapped |  |  |
| Tools/platform preferences known |  |  |
| Success KPIs defined |  |  |
| Risks or spikes planned |  |  |

Would you like this as a downloadable **Excel/Google Sheet checklist** or a **Markdown template** for your architecture documents?

In a **large-scale system**, the number of **ADR (Architecture Decision Records)** depends on the complexity, lifecycle stage, and breadth of architectural concerns. However, there is a **strategic way** to plan them so they are **progressive, not overwhelming**, and reflect the **evolving architecture**.

## 🧠 General Guidelines

* A typical enterprise-scale system (e.g., 20+ microservices, multi-team) may have **25–100 ADRs** over its lifecycle.
* You don't write everything upfront — they evolve with the system.
* Focus on **significant**, **long-lived**, and **costly-to-reverse** decisions.

## ✅ What to Decide in the **First Few ADRs** (Foundation ADRs)

These set the architectural direction — they're foundational and help avoid chaos.

| **ADR #** | **Topic** | **Purpose** |
| --- | --- | --- |
| 1 | **Architecture Style** | Decide on microservices, monolith, DDD, serverless, etc. |
| 2 | **Cloud Platform** | e.g., Azure vs AWS vs hybrid |
| 3 | **Authentication & Authorization Strategy** | e.g., Azure AD B2C, JWT, OAuth2 |
| 4 | **API Gateway / Ingress Layer** | e.g., Azure APIM vs Kong vs NGINX |
| 5 | **CI/CD Tooling** | e.g., Azure DevOps vs GitHub Actions |
| 6 | **Infrastructure as Code (IaC)** | e.g., Terraform vs Bicep vs Pulumi |
| 7 | **Observability Stack** | e.g., Application Insights + Log Analytics + DCR setup |

These early ADRs give teams a shared foundation, helping avoid decision sprawl.

## 📈 Subsequent ADRs – Modular and Progressive

As the system grows, ADRs should be written **per domain, per integration**, or when major **new functionality** is added.

| **ADR Type** | **When to Write** |
| --- | --- |
| **Database per service or shared?** | When designing microservices' data ownership |
| **Message Broker** (e.g., Event Grid vs Kafka) | When enabling async communication |
| **Caching Strategy** | When adding performance layers |
| **Deployment Strategy** | Blue/green, rolling, or canary deployments |
| **Versioning Approach** | When APIs are exposed externally |
| **Multi-tenancy Model** | For SaaS or B2B platforms |
| **Data Residency** | If GDPR, HIPAA, or regional rules apply |
| **Retries & Circuit Breakers** | When handling resiliency and fault tolerance |
| **Frontend Hosting & Routing** | SPA routing, static hosting, CDN setup |
| **Service Mesh Decision** | When cross-service traffic observability becomes important |
| **Secrets Management** | e.g., Azure Key Vault vs environment config |

## 🧭 Good Approach to Managing ADRs

### ✅ 1. **Start with the Backbone**

Decide on fundamentals that will affect all teams: auth, API access, platform, CI/CD.

### ✅ 2. **One ADR = One Decision**

Keep it atomic. Don't club unrelated topics.

### ✅ 3. **Use Templates**

Stick to a format like [MADR](https://adr.github.io/madr/) or simple Markdown:

# Title

## Status

## Context

## Decision

## Consequences

## Alternatives Considered

### ✅ 4. **Track ADR Status**

Use Accepted, Rejected, Superseded, Proposed to manage ADR lifecycle.

### ✅ 5. **Keep It In Version Control**

Place ADRs in a /docs/architecture/adr/ folder in your codebase.

## 📘 Real-World Example Progression

Here’s a realistic sequence for a large Azure-based project:

1. ADR-001: Choose Microservices with Clean Architecture
2. ADR-002: Host all APIs in Azure App Services with Azure API Management
3. ADR-003: Use Azure AD B2C with OAuth 2.0 for authentication
4. ADR-004: Use Terraform for IaC
5. ADR-005: Implement GitHub Actions for CI/CD pipelines
6. ADR-006: Use Cosmos DB for unstructured data and SQL Server for relational
7. ADR-007: Adopt Azure Event Grid for event-driven communication
8. ADR-008: Service-to-service auth using managed identities
9. ADR-009: Use Redis cache for response acceleration
10. ADR-010: Multi-region deployment strategy using Traffic Manager

Would you like a **template repo structure** or **ADR naming/numbering convention** for organizing them in your project?

Certainly! Here's a **more detailed explanation** of how **Architecture Decision Records (ADRs)** and **System Design / High-Level Design (HLD)** relate, with step-by-step phases, examples, artifacts, and how they evolve together in a **real-world project** (like a healthcare or retail system).

## 🧭 **Overall Flow: System Design vs ADR**

| **Stage** | **What Happens** | **Outputs** | **Notes** |
| --- | --- | --- | --- |
| **1. Understand Business & Technical Context** | Business use cases, existing systems, constraints, users, SLA expectations | Business process flows, personas, functional & non-functional requirements | This is the input to all designs & decisions |
| **2. Draft Initial System Design** | Identify core modules, candidate architecture styles, major data flows | Context diagram, rough component breakdown, integration needs | No decisions yet—exploratory |
| **3. Identify Decision Points** | Highlight areas needing trade-offs (e.g., database type, communication protocol) | Decision matrix or architectural options table | This drives what ADRs you must write |
| **4. Write Initial ADRs** | Document your choice and rationale for each architectural decision | ADR #1, #2, etc. | Each ADR has: Title, Status, Context, Decision, Consequences |
| **5. Finalize High-Level Design (HLD)** | Solidify architecture based on ADRs; define boundaries, flows, NFRs | Final component diagrams, sequence diagrams, infra layout | Everything here should reflect the ADR decisions |
| **6. Low-Level Design (LLD) & Implementation** | Module-level class designs, contracts, repo structure | Class diagrams, OpenAPI specs, gRPC contracts, etc. | Based on HLD direction |

## 📜 **What Is an ADR and Where It Fits**

| **Element** | **Description** |
| --- | --- |
| **ADR Purpose** | Captures one major architectural decision with reasoning and consequences |
| **When to Write** | After exploring options and before locking in the decision (e.g., before infra or code setup) |
| **Typical ADR Examples** | Use of Azure Cosmos DB over Azure SQL, Use of Microservices over Monolith, Choosing Event-Driven Architecture, Using Azure API Management |
| **How Many** | In large systems: 10–30+ ADRs over time; start with 3–5 critical ones |
| **Tools** | Markdown files stored in Git repo, e.g., /docs/adr/0001-authentication.md |

## 📌 **Checklist Before Writing ADRs**

1. ✅ Have business use cases been analyzed?
2. ✅ Have technical constraints or NFRs (scalability, latency, data security) been defined?
3. ✅ Are there multiple options or patterns being considered?
4. ✅ Have PoCs or spikes been done for critical decisions?
5. ✅ Are stakeholders (architects, dev leads, infra) aligned?
6. ✅ Has a high-level sketch or flow been created to frame the decision?

## 🔁 **Iterative Nature**

* You **don’t need to wait** for all ADRs to be written before doing HLD.
* It's **incremental**:
  + Start with **Foundational ADRs** (auth, architecture style, data strategy)
  + Then proceed to **Subdomain ADRs** (e.g., async comm in claims module)
  + Update HLD as decisions evolve

## 🛠 Real Example: Healthcare Pre-Authorization System

| **Phase** | **Example Output** |
| --- | --- |
| Draft System Design | Patient, Provider, Intake API, Determination service identified |
| ADRs |  |

* ADR-001: Use Azure Cosmos DB for clinical notes
* ADR-002: Use Azure AD B2C for authentication
* ADR-003: Use Dapr for service-to-service communication | | Final HLD | Updated architecture diagram with Cosmos DB, Dapr sidecars, API Gateway, async event flows |

## 📘 Tips for Clean Process

* **Version ADRs** using a naming convention like 0001-authentication.md
* **Link ADRs from your HLD document**
* Use a **diagram per ADR** if needed (sequence, data flow)
* If ADR changes later, mark it as **"Superseded by ADR-008"** to preserve history

Would you like me to generate a **sample ADR** or a **HLD template with placeholders** that integrates with ADRs?

Great question. Here's a **clear and structured explanation** of **when to do system design** in a real-world software project lifecycle, including **system design**, **high-level architecture**, and **how it relates to ADRs, business requirements, and implementation**:

## 🧱 **When to Do System Design (High-Level Architecture)**

### ✅ **Ideal Time: After Requirements Are Gathered, Before Development Starts**

System Design should begin:

* After **business and technical requirements** are clear.
* Before you write actual production code or finalize infra.
* In parallel with or slightly **before writing key ADRs**.

## 🧭 **Full Timeline View**

| **Phase** | **What Happens** | **Do System Design Now?** |
| --- | --- | --- |
| 📋 **Requirement Analysis** | Understand business flows, users, data, and integrations | ❌ Not yet |
| 🔍 **Use Case Modeling** | Define functional & non-functional requirements | 🟡 Start identifying components |
| 🧠 **System Design Planning** | Identify modules, flows, data design, security concerns | ✅ Start high-level design |
| 📄 **Write Initial ADRs** | Choose key architectural strategies (e.g., event-driven, API gateway) | ✅ Use design to support ADRs |
| 🏗 **Finalize System Design** | Create detailed HLD artifacts (diagrams, contracts, flows) | ✅✅ |
| 🧰 **Low-Level Design (LLD)** | Component-level classes, data models, APIs | ❌ This is post-HLD |
| 🛠 **Development Starts** | Teams begin implementing services and infrastructure | ❌ Design is mostly complete |

## 🧩 What System Design Covers (in HLD phase)

| **Area** | **What’s Designed** |
| --- | --- |
| Modules / Bounded Contexts | Intake, Authorization Engine, Determination, Notifications |
| Communication | REST vs gRPC vs Event-Driven |
| Auth & Security | OAuth2 / Azure AD B2C, Token flows |
| Infrastructure | Cloud services used, scalability patterns |
| Data Strategy | DB choices (SQL, NoSQL), caching, retention |
| Integration | 3rd party APIs, EHR systems, payment systems |

## 🛎️ Why You Should Not Delay System Design

* **Without it, ADRs will be isolated** decisions.
* Dev teams may start building with **incorrect assumptions**.
* It helps align **domain knowledge, business priorities, and technical architecture**.

## 🎯 Summary: When to Do System Design

| **Situation** | **Do System Design?** |
| --- | --- |
| Project kickoff | ❌ Not yet |
| After requirement workshop | ✅ Start it |
| Before writing ADRs | ✅ Preferably before or in parallel |
| After ADRs | ❌ Too late for foundational decisions |
| Before dev sprint planning | ✅ Must be ready |

Let me know if you'd like:

* 📄 A **System Design Checklist**
* 🖼️ A **sample system design template**
* ✅ A **system design readiness self-assessment**

These will make your design more structured and reviewable.

# System Design vs High-Level Design (HLD): Detailed Comparison with Checklists

## 🔹 SYSTEM DESIGN: Concepts, Purpose, and Checklist

### 🎯 Purpose:

System design focuses on the **overall architecture and decisions** needed to meet business and technical requirements. It defines the **macro-level** structure of the solution.

### 🧱 Concepts Covered:

1. **Architecture Type** – Monolith, Microservices, Event-Driven, Serverless
2. **Deployment Strategy** – Cloud, Hybrid, On-Premises
3. **Security Architecture** – Identity, Access Control, Data Protection
4. **Scalability & Performance Plan** – Caching, Load Balancing, CDN
5. **Resiliency & Fault Tolerance** – Retry, Circuit Breaker, Failover
6. **Data Strategy** – Types of databases, Storage patterns
7. **Communication Patterns** – Sync (REST, gRPC), Async (queues, pub/sub)
8. **Observability** – Monitoring, Logging, Tracing
9. **Compliance & Audit** – HIPAA, GDPR, PCI-DSS
10. **Architecture Decision Records (ADRs)** – Justification for key decisions

### ✅ System Design Checklist:

* [ ] Have all business and technical requirements been understood?
* [ ] Have you chosen the appropriate architectural style?
* [ ] Have NFRs (performance, scalability, availability) been mapped?
* [ ] Is a deployment strategy documented?
* [ ] Are data flow and data storage solutions defined?
* [ ] Are external system integrations planned?
* [ ] Have security and compliance needs been considered?
* [ ] Are observability and monitoring tools planned?
* [ ] Have failure scenarios been modeled?
* [ ] Have ADRs been written for core technical decisions?

## 🔹 HIGH-LEVEL DESIGN (HLD): Concepts, Purpose, and Checklist

### 🎯 Purpose:

HLD breaks the system into **major components**, defines their interactions, and outlines **module-level responsibilities**. It is more implementation-aware.

### 🧩 Concepts Covered:

1. **Component Breakdown** – Logical grouping of services/modules
2. **Service Responsibilities** – What each module does
3. **Interface Contracts** – API definitions and payloads
4. **Deployment Topology** – Containers, clusters, regions, VPCs
5. **Data Flow Diagrams** – How data moves between components
6. **Authentication & Authorization Design** – Flow diagrams, token usage
7. **Integration Strategy** – Message brokers, APIs, events
8. **Error Handling Strategy** – Graceful error propagation
9. **High-Level Sequence Diagrams** – Use case-driven flow of control
10. **HLD Document or Architecture Workbook** – Deliverable for dev teams

### ✅ High-Level Design Checklist:

* [ ] Have all key services/modules been identified?
* [ ] Is a component diagram created with clear boundaries?
* [ ] Have all service-to-service interactions been mapped?
* [ ] Are interface contracts defined (request/response)?
* [ ] Have you drawn data flow diagrams?
* [ ] Are auth mechanisms documented in detail?
* [ ] Are external APIs or third-party integrations covered?
* [ ] Have failure points in component communication been addressed?
* [ ] Is deployment architecture covered at a logical level?
* [ ] Are diagrams version-controlled and shared with dev teams?

## 🔄 Summary of Key Differences:

| **🔍 Feature** | **🧠 System Design** | **🏗️ High-Level Design (HLD)** |
| --- | --- | --- |
| **Scope** | Entire system, architecture strategy | Subsystem/module level design |
| **Focus** | Tech stack, ADRs, NFRs, deployment | Component diagrams, APIs, data flow |
| **Timing** | Early phase (before coding starts) | After system design, before LLD |
| **Audience** | Architects, Leads, Product/Infra teams | Developers, Leads, QA, Architects |
| **Artifacts** | ADRs, architecture diagrams, strategy | HLD docs, component/interface diagrams |
| **Example Question** | Should we use CQRS with Kafka? | How does the PatientService expose API? |
| **Outputs** | Decision framework for tech choices | Dev-friendly structure and interfaces |

Let me know if you'd like to add **Low-Level Design (LLD)** to this comparison or generate checklists in Excel/Markdown format.

Here's a detailed comparison between **System Design** and **High-Level Design (HLD)**, including concepts and checklists. You can view and expand it in the document titled **“System Vs Hld Checklist”** in the side panel.

Would you like to add Low-Level Design (LLD) next, or convert this into an Excel download too?

To **know when to start System Design, HLD, and LLD**, here's a phase-wise guide for large/medium-scale project planning:

### 🚦**WHEN TO START WHAT** – Decision Matrix

| **Phase** | **Trigger Event or Milestone** | **Start This** | **Output/Deliverable** |
| --- | --- | --- | --- |
| ✅ **Before project kickoff / proposal** | Business need, RFP, feasibility study | ✔️ System Design | Architecture vision, ADRs, architectural decisions |
| ✅ **Right after functional requirements & constraints are known** | BRDs, PRDs, epic-level use cases ready | ✔️ System Design (detailed) | Architecture diagram, NFR mapping, tech stack choice |
| ✅ **After system architecture is decided** | ADRs written, tech stack selected | ✔️ High-Level Design (HLD) | Component diagrams, data flows, interfaces, HLD doc |
| ✅ **After HLD is approved** | Modules/teams are known, devs ready to start coding | ✔️ Low-Level Design (LLD) | Class diagrams, DB schema, method contracts, code stubs |
| 🔁 **Whenever key design decisions need change** | Performance issues, scaling changes | ✔️ New ADRs | Documented reasoning for pivot in architectural choices |

### 🧭 Detailed Step-by-Step Flow

#### Step 1: ✨ **Initiation Phase**

* Input: Business problem, product vision, constraints
* Output: **Initial System Design**
  + Identify architectural style
  + Create ADRs for key tech choices
  + Rough sketch of modules/services

#### Step 2: 🔍 **Analysis Phase**

* Input: Use cases, epics, NFRs
* Output: **Detailed System Design**
  + Decide cloud/on-prem strategy
  + Data strategy, resiliency, security
  + Integration points and system boundaries

#### Step 3: 🧱 **Design Phase**

* Input: Finalized system architecture
* Output: **High-Level Design**
  + Module responsibilities
  + API contracts, component diagrams
  + Data flows, auth flow, external systems

#### Step 4: 🧑‍💻 **Build Phase (Pre-Coding)**

* Input: Approved HLD
* Output: **Low-Level Design**
  + Class-level structure
  + DB schema, indexes
  + Code interface definitions
  + Pseudocode, test plan alignment

### 🕰️ Visual Timeline Summary

|--- Project Kickoff ---|--- System Design ---|--- HLD ---|--- LLD ---|--- Dev Begins ---|

| | ADRs, Infra | API, Data | Classes | CI/CD, coding |

Would you like this as a timeline Excel or visual flowchart too?

Here’s a **System Design Checklist** (based on the visual guide you provided) that you can use **before starting High-Level Design (HLD)**. It ensures all foundational decisions are made and core system behaviors are understood.

## ✅ **System Design Pre-HLD Checklist**

### 🧱 1. **Core Concepts of System Design**

| **Item** | **Question / Check** |
| --- | --- |
| ✅ Latency vs Throughput | Do we need low latency or high throughput? What's the priority? |
| ✅ CAP Theorem | Which two will you prioritize: Consistency, Availability, or Partition Tolerance? |
| ✅ Load Balancing | Will we distribute traffic across multiple nodes/servers? How? (e.g., NGINX) |
| ✅ Caching | Will we cache responses/data? Where (client/server/DB)? |
| ✅ Sharding | Do we need to split large datasets across DBs? |
| ✅ Replication | Do we need master-slave or multi-primary DB replication? |
| ✅ Consistency Models | Is Strong/Eventually/Causal consistency more important for your system? |
| ✅ Scaling Strategy | Will we scale vertically (beef up) or horizontally (add nodes)? |

### 🧩 2. **Key Components & Tools**

| **Item** | **Question / Check** |
| --- | --- |
| ✅ Load Balancers | Which load balancer will be used (NGINX, AWS ELB, HAProxy)? |
| ✅ Databases | What DB fits our need — SQL, NoSQL, document store? |
| ✅ Cache Systems | Will we use Redis/Memcached? For what data types? |
| ✅ CDNs | Will static assets be served via CDN (Cloudflare, Akamai)? |
| ✅ Object Storage | Is object storage (S3, GCS) required for files/images/logs? |
| ✅ Queue Systems | Will we use Kafka, RabbitMQ, or SQS for async processing? |
| ✅ Containerization | Will the system use Docker/Kubernetes? |
| ✅ Service Mesh | Are we adopting service mesh (Istio, Linkerd) for service communication? |

### 📘 3. **Important Terms & Behaviors**

| **Item** | **Question / Check** |
| --- | --- |
| ✅ Idempotency | Are APIs idempotent for repeated calls? |
| ✅ Rate Limiting | Are there per-user/system rate limits? |
| ✅ Throttling | Will we slow down high-frequency callers gracefully? |
| ✅ Backpressure | Will producers know when consumers are overwhelmed? |
| ✅ Heartbeat | Do services regularly check/report their health? |
| ✅ Cold Start | Are you considering latency when spinning up new services (e.g., Lambda)? |
| ✅ Sticky Sessions | Do we need session stickiness in load balancing (stateful systems)? |
| ✅ Session Management | Is session state stored server-side, client-side, or externally? |
| ✅ Service Discovery | Will we dynamically resolve services (Consul, Eureka)? |
| ✅ Leader Election | Does any node/service act as a master/leader? |

### 🔍 4. **Monitoring & Observability**

| **Item** | **Question / Check** |
| --- | --- |
| ✅ Logging | Are logs centralized and structured (e.g., via ELK stack)? |
| ✅ Metrics | Are CPU, memory, and custom metrics captured (Prometheus, Azure Monitor, etc.)? |
| ✅ Dashboards | Do we have Grafana, Kibana, or Azure Dashboards configured? |
| ✅ Alerting | Are alerts in place for failures, high CPU, etc.? |
| ✅ Tracing | Can we trace user requests across services (e.g., OpenTelemetry)? |

### ⚡ 5. **Reliability & Fault Tolerance**

| **Item** | **Question / Check** |
| --- | --- |
| ✅ Health Checks | Are liveness/readiness probes defined? |
| ✅ Retry & Timeout Logic | Do services have retries with backoff and fail-safe timeouts? |
| ✅ Failover Mechanisms | Will the system auto-switch to backup DBs/services on failure? |
| ✅ Auto-Scaling | Will infra scale automatically on load (CPU, queue size)? |
| ✅ Load Testing | Have you planned for load/stress testing before go-live? |

### 🧰 6. **Design Patterns Preparedness**

| **Item** | **Question / Check** |
| --- | --- |
| ✅ Microservices | Is your system split into independently deployable services? |
| ✅ Event-Driven Architecture | Are events used to decouple producers and consumers? |
| ✅ Circuit Breaker | Will dependent service failures be isolated? |
| ✅ Bulkhead Pattern | Do you isolate services/resources per tenant/type to avoid ripple failures? |
| ✅ Backend for Frontend (BFF) | Do different frontends need custom APIs (e.g., mobile vs web)? |

Would you like this as an **Excel checklist** or included in your HLD documentation as a preface section?