Cognixia

Foundations of System Design

By: Ahmad

1

Cognixia

Architecture Flow

www.cognixia.com

2

Architectural Workflow

Cognixia

Role / Concept	Primary Focus	Key Output / Constraint
Business Strategy / Needs	What is the overarching **business goal**?	**High-Level Requirements** and **Budget**
Solution Architect (SA)	**Solving the Business Problem.** Choosing the high-level technology and external integration points.	**Solution Blueprint** (Technology Selection (Vendors, Platforms), Project ROI/Cost Analysis.)
System Architect (SyA)	**Defining the Execution Environment.** Infrastructure, scaling, security, and integration across the ecosystem.	**System Design** (System Architecture Diagram, Infrastructure Plan, Network Topology, Security Model).
Software Architect (SWA)	**Designing the Internal Structure.** Structuring the custom application to meet System Design constraints and NFRs.	**Software Architecture** (Software Architecture Pattern, Component Diagrams, Coding Standards.
Software Designer / Developer	**Implementation Details.** Writing the specific code, adhering to the architecture.	**Working Code** that adheres to the established architecture and standards.
DevOps / Infrastructure Engineer	**Implementing the System Design & Deployment Pipelines.** Building the environments to host the application.	**Deployment Pipelines, Infrastructure-as-Code (IaC)**.
Operations / SRE	**Sustaining Production & Evolution.** Monitoring, patching, scaling, and managing deployments in a live environment.	**Operational Feedback** and **System Stability**.

www.cognixia.com

3

Architectural Workflow | Skills

Cognixia

Role / Concept	Required Skills	Specific Example Action
Business Strategy / Needs	Business Acumen, Financial Analysis, Stakeholder Management.	**Action:** Define the need to handle 1 million transactions with 200ms latency and set the Q4 launch date.
Solution Architect (SA)	Domain Modeling, Cloud Strategy, Vendor Evaluation, Cost Management.	**Action:** Decide to custom build on **On-premises** or **cloud** and integrate with the **Stripe** payment gateway and an external **KYC** service.
System Architect (SyA)	High-Availability Design, Data Flow Design. **Synthesis of**: Networking, Security, Platform, Data Management principles from specialists.	**Action:** Mandate the use of a **cross-region Kubernetes cluster** and the **Cassandra database** (for high-speed writes) to meet the availability and throughput NFRs.
Software Architect (SWA)	Design Patterns, Modularity, API Design, Trade-off Analysis, Technical Governance.	**Action:** Define the application as an **Event-Driven Microservices Architecture**, ensuring all core services (Payment Processor, Fraud Detection) are **stateless** and communicate via a message queue.
Software Designer / Developer	Programming Language Proficiency, Unit/Integration Testing, Data Structures & Algorithms, Clean Code Principles.	**Action:** Write the core processTransaction() code, ensuring it strictly uses the architect-defined **message broker interface** for outputs and avoids using local server memory for data storage.
DevOps / Infrastructure Engineer	Cloud Computing (GCP), Kubernetes/Docker, Terraform/Ansible, CI/CD Tools.	**Action:** Use **Terraform** to provision the cross-region Kubernetes cluster and configure the automated **CI/CD pipeline** to deploy the microservices into the cluster upon code approval.
Operations / SRE	Observability Tools (Prometheus/Grafana), Incident Response, Automation Scripting, SLO/SLA Definition.	**Action:** Configure **Prometheus** to monitor the critical 200ms latency SLO.

4

Cognixia

System Design

5

Cognixia

What is System Design

- **System Design** is the **process** of defining the **architecture**, **components**, **modules**, **interfaces**, **data flow**, and behaviors of a system to **meet specific business** or **technical requirements**.
- It is the **bridge between requirements** (*what the product should do*) **and implementation** (*how the code actually works*).
- It **focuses** on **how** to **build** a **system that** is:
  - **Scalable**
  - **Reliable**
  - **Secure**
  - **Maintainable**
  - **Efficient**
- It **involves translating business needs** into a **technical blueprint** that guides developers, architects, and engineers.
- **System Design answers the question** → "**How** do we **build** a **system that** can handle millions of users, stay available 99.99% of the time, respond in <200ms, and scale when traffic 10x overnight?

6

System Design | Key Elements

- **Architecture**
  - The conceptual model that defines the **overall structure** (*e.g., monolithic, microservices, client-server*).
- **Components/Modules**
  - The **individual building blocks** (*like databases, servers, load balancers, caches, and message queues*) and their specific responsibilities.
- **Interfaces**
  - How **components communicate** with each other (*e.g., APIs, protocols*).
- **Data Flow**
  - How **data is stored**, managed, and **travels** through the system.

www.cognixia.com

7

Pillars of Design

www.cognixia.com

8

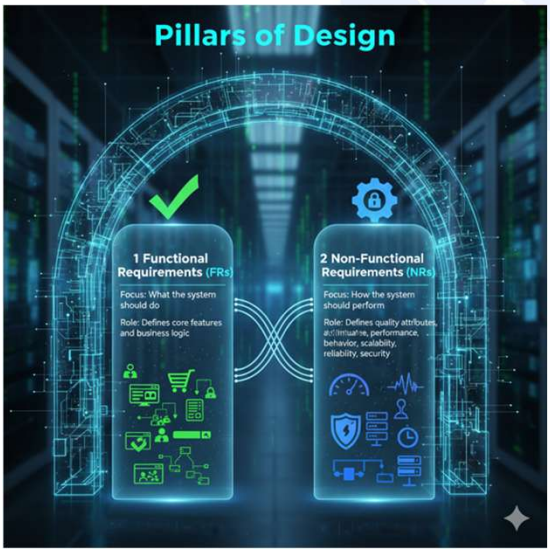
What is Pillars of Design

- **System design** is **built** upon **two distinct**, yet complementary, types of requirements:

  - **Functional Requirements (FRs)**
    - **Focus**: **What** the **system should do**
    - **Role**: **Defines core** features and **business logic**

  - **Non-Functional Requirements (NFRs)**
    - **Focus**: **How** the **system should perform**
    - **Role**: **Defines quality attributes**, **performance**, **behavior**, **scalability**, **reliability**, **security**

- **Together**, they **define** what a system must do and how well it must do it, **forming** the **complete specification** for the **architecture**.



www.cognixia.com

9

Functional Requirements (FR)

- These describe **what the system does** — specific actions, features, or behaviors the system must perform.

- They define **business logic, use cases, and user expectations**.

- **Examples:**
  - Banking App: **User** should be **able to transfer money** between accounts
  - E-commerce: System should **allow customers to add items** to a **cart**
  - WhatsApp: System should **deliver real-time messages**
  - Netflix: Should allow **users** to **search** and **stream movies**
  - Immigration Portal: **Users** should **upload documents** and track application status

- **Real-Life Analogy**
  - In a **restaurant**: Taking orders, cooking food, billing → These are **functional requirements** (core functions).

www.cognixia.com

10

Non-Functional Requirements (NFRs)

Cognixia

- Non-Functional Requirements **define** the **quality attributes**, **constraints**, and **operational characteristics** of the system.
- They **specify** the criteria used to judge the **operation of a system**, rather than specific behaviors.
  - **Focus**: **How** the **system should perform**.
  - **Nature**: Broad, often **involving trade-offs**, and critical to overall user satisfaction and business success.
  - **Goal**: To **satisfy** the **technical** and quality **needs** of the system owners and users.
- **Real-Life Analogy:**
  - In a **restaurant**:
    - How fast food is served
    - Cleanliness
    - Hygiene
    - Customer service quality
  - These do not affect *what* is done but affect *how well* it is done.

www.cognixia.com

11

NFRs | Components

Cognixia

NFR Category	Description	Example	Components / Tools to Use
Performance	**Speed** & responsiveness	Response < 2 sec	**Caching** (Redis, Memcached), **CDN** (CloudFront, Akamai), **Load Balancer**
Scalability	**Handle growth** in traffic/users	Handle 1M users	**Auto Scaling** (AWS ASG, Kubernetes HPA), Microservices, **Message Queue** (Kafka, RabbitMQ), **Database Sharding**, Stateless Services, Containerization (Docker, K8s)
Availability	**System uptime** (fault tolerance)	99.99% uptime	**Multi-Zone Deployment**, **Load Balancer** (ALB, NLB), Health Checks, Failover, Replication, Auto-Healing, **Backup & Recovery**
Reliability	**Works without failure**	No message loss	**Distributed Queues** (Kafka, SQS), Replication, Acknowledgement Systems, Idempotency, **Retry Policies**, **Circuit Breaker** (Hystrix)
Security	**Data protection** and secure access	Encryption, MFA	**IAM**, OAuth2, JWT, **WAF**, **Firewall**, HTTPS/**TLS**, **Data Encryption** (KMS), **Key Vault**
Maintainability	**Easy to modify**, fix, or enhance	Modular microservices	Microservices Architecture, **APIs**, **CI/CD**, Containerization, Version Control (Git), **Terraform** / IaC, Clean Code Principles
Usability	**Ease of use** and UI experience	Smooth checkout	Responsive UI/UX, UI Frameworks (React, Flutter), Accessibility (WCAG), Consistent Design, **A/B Testing**, UX Research
Compliance	Follow **legal** and **industry standards**	GDPR, HIPAA	Audit Logs, **Data Masking**, **Encryption**, IAM, Compliance Certifications, Legal Mapping, Consent Management
Interoperability	**Work with other systems**	API integration	**API Gateway**, REST/gRPC, Webhooks, Enterprise Integration Patterns

12

NFRs | Example

Cognixia

Requirement	Solution	Components
Fast product pages	Use caching	Redis, CDN
Handle festival sale traffic (10x users)	Auto-scale services	Kubernetes, Load balancer
No downtime during order processing	Fault tolerance	Active-active clusters, replication
Secure payments	Encrypt, MFA	TLS, OAuth2, PCI-DSS
Multiple devices (mobile, web, tablet)	Interoperability	API Gateway, REST

13

Cognixia

Technical Blueprint

14

Business Vision to Technical Blueprint

Cognixia

- In system design and software architecture, one of the most crucial roles is **bridging the gap between business goals and technical implementation**.
- This process ensures that **what the business wants** becomes **what the engineering team builds** — accurately, efficiently, and strategically.
- **What Is Business Vision**
  - **Business vision defines**:
    - What the company wants to achieve
    - Target customers
    - Value proposition (*why users will use the product*)
    - Business goals (*increase revenue, reduce cost, improve efficiency, etc.)*
  - **Example**: *"We want to build a scalable e-commerce platform that supports millions of users and provides 24/7 availability."*

15

Vision to Blueprint … continue

Cognixia

- **What Is a Technical Blueprint**
  - Technical Blueprint is the **system design and architecture plan** that translates business expectations into a **technical structure**.
  - It **includes**:
    - System architecture (*Monolith, Microservices, Serverless, Event-driven*)
    - Technology stack (*Databases, Cloud services, APIs, CDN*)
    - NFR mapping (*Scalability, Security, Performance*)
    - Data flow, integrations, components, deployment architecture
    - Governance and compliance (*GDPR, HIPAA, PCI-DSS*)

16

Online Grocery Delivery | Business Idea

Cognixia

Business Goal	Technical Translation (Blueprint)
Handle 1M daily users	Use CDN, Load Balancers, Auto-scaling, Microservices
Should be available 24/7	Use multi-region deployment, failover, 99.99% uptime
Should be secure	OAuth2, JWT, WAF, Encryption, MFA, API Gateway Security
Fast page load (<2 sec)	Use caching (Redis), compressed images, SSR, CDN
Support multiple payment methods	API integration with Razorpay, Stripe, PayPal
Business wants real-time order tracking	Use WebSockets, Kafka, GPS integration

17

Cognixia

HLD vs LLD

18

Design Approaches: HLD vs LLD

Cognixia

- System Design **involves two major stages**:
  - **High-Level Design (HLD)** — *Architectural blueprint*
  - **Low-Level Design (LLD)** — *Detailed technical implementation*
- **Both** are **essential** but **used at different stages** and by different roles (*architects, designers, developers*).
- **HLD** and **LLD** are **sequential** and **hierarchica**l:
  - **HLD is performed first:**
    - It **creates** the **container** and the **boundary** for the **entire system**.
    - The **output** of the **HLD** (*the system architecture and component list*) **becomes** the **input** for the **LLD**.
  - **LLD is performed next:**
    - The **LLD** then **specifies** the **internal workings** of **each individual component defined** in the **HLD**.
- **Workflow**:
  - Requirements → HLD → LLD → Development → Testing → Deployment

www.cognixia.com

19

HLD vs LLD | Comparison

Cognixia

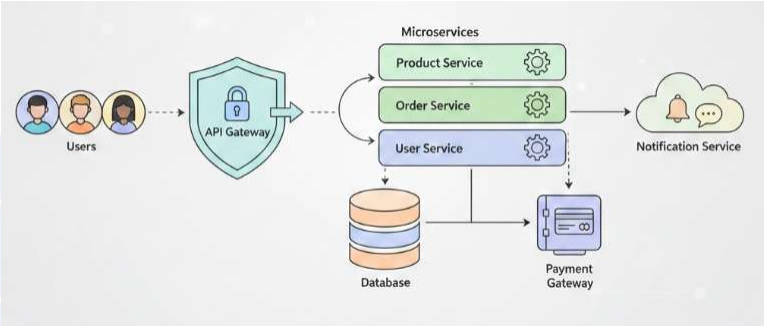
Feature	High-Level Design (HLD)	Low-Level Design (LLD)
Purpose	**System overview** & architecture	**Detailed internal design**
Focus	***What* the system will do**	***How* it will be implemented**
Scope	**Entire system** / components	**Individual modules** & classes
Audience	**Architects**, stakeholders, tech leads	**Developers**, testers
Abstract	**Conceptual,** abstract	**Detailed**, technical
Output	**Architecture diagrams**, tech stack, data flow	**Class diagrams**, APIs**, DB schema**, pseudocode
Includes	Microservices, load balancing, caching, databases, messaging	Class design, functions, data structures, logic
Flexibility	**Open for discussion**	**Mostly final** & implementable
Example	Use API Gateway, Redis caching	Define API endpoints, DB tables, class methods

www.cognixia.com

20

High-Level Design (HLD) | Architect's View

- **Objective:** Provide the **big-picture architecture** of the system.
- **HLD Example Diagram (E-commerce)**
  - Users → API Gateway → Microservices → Database → Payment Gateway → Notification Service



www.cognixia.com

21

Low-Level Design (LLD) | Engineer's View

- **Objective: Convert HLD** into **detailed implementation plans**.
- **LLD Example**:
  - **Login API Schema** (*HTTP POST request to an API endpoint used for user login*)

```
POST /api/login
{
  "email": "user@example.com",
  "password": "123456"
}
```

  - **Database User Table** (*Table stores information about users in a system*)

Field	Type	Description
user_id	int	Primary key
email	varchar	Unique email
password_hash	varchar	Encrypted password
last_login	datetime	Last login timestamp

www.cognixia.com

22

Cognixia

Architecture Reviews

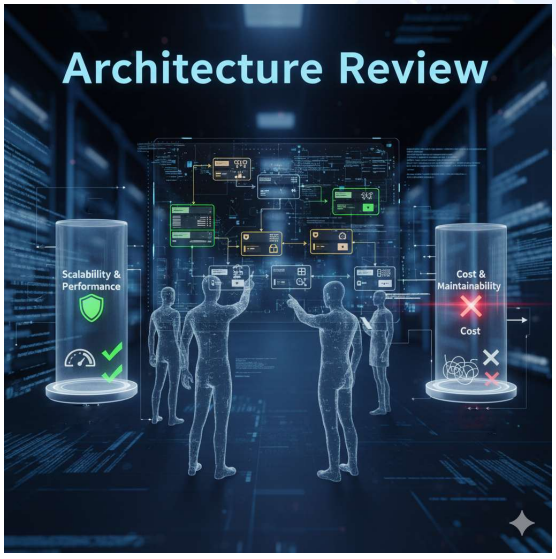
www.cognixia.com

23

Cognixia

Architecture Reviews

- An architecture review is a **systematic evaluation** of a **software system's architecture** to ensure it **meets functional** and **non-functional requirements**, adheres to best practices, and aligns with business goals.
- An effective review ensures that the **proposed architecture** is **fit for purpose**, feasible, and robust **before** significant **development** effort is invested.
- **Purpose**
  - **Identify risks**, gaps, and potential problems early.
  - Ensure the **architecture supports scalability**, **maintainability**, **performance**, **security**, and **cost-effectiveness**.
  - Promote knowledge sharing and **consensus among stakeholders**.



www.cognixia.com

24

Architecture Review | Steps

Step	Method	Description / Actions	Tips / Examples
1	**Define Objectives and Scope**	• **Identify why the review is needed** (*early design check, pre-release evaluation, post-implementation audit*). • **Define which parts of the architecture will be reviewed** (*components, interfaces, data flows, scalability, security, etc.).*	**Example**: Reviewing a new microservices system before production deployment.
2	**Prepare Documentation**	• **Collect all relevant artifacts**: architecture diagrams (*component, deployment, data flow*), design documents, functional & non-functional requirements, technology stack and decisions.	Well-prepared documents make the **review efficient and focused**.
3	**Assemble the Review Team**	• **Include** diverse **stakeholders**: software architects, developers/tech leads, QA/testing engineers, operations/DevOps, business representatives *(if needed)*.	**Goal**: Ensure all perspectives are considered (***technical, operational, business***).
4	**Conduct the Review**	• **Walkthrough** the architecture **step by step**. • **Ask critical questions:** scalability, performance, maintainability, security, cost, risk. • Use techniques like checklists, scenario-based analysis, or questionnaires.	**Consider alternative approaches** to **compare pros and cons.**
5	**Document Findings and Recommendations**	• **Record strengths, weaknesses/risks**, and **suggested improvements** or design changes.	Prioritize issues based on impact and urgency.
6	**Follow-Up**	• **Assign action items to responsible teams**. • Track progress and ensure recommended changes are implemented. • Schedule re-review if needed.	Especially important for critical systems.

25

Architecture Decision Records

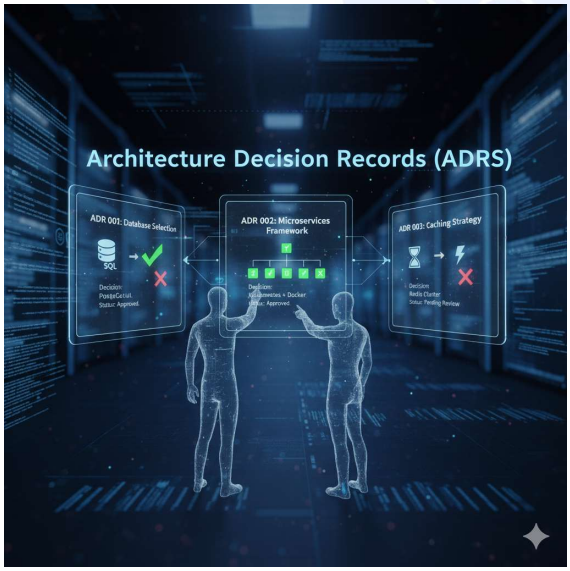
www.cognixia.com

26

Architecture Decision Records (ADRs)

Cognixia

- An Architecture Decision Record (ADR) is a **written document** that **captures important architectural decisions** made during a software project.
- It **records what decision was made**, **why it was made**, **what alternatives** were considered, and the consequences.
- **Purpose:**
  - **Provides** historical **context** for **architectural choices**.
  - **Ensures** transparency and **knowledge sharing among team members**.
  - **Helps new team members** understand **why** the **system** is **designed** a **certain way**.
  - **Supports future decision-making** and reduces repeated debates.



27

ADR | Components

Cognixia

- While **formats vary**, an effective ADR typically follows a consistent, **structured template** to ensure **all necessary context is captured**.

Component	Description	Purpose / Example
Title	A clear, concise **name of the decision**.	• Makes it easy to identify the decision quickly. • **Example:** • **Choose Database for E-commerce System**
Status	**Current state of the decision:** proposed, accepted, deprecated, or superseded.	• Helps track which decisions are active or outdated. • **Example:** • **Accepted**

Context	Background information and reasons **why the decision is needed**.	• Provides rationale and problem description.
		• **Example:**
		  ○ The **system needs to handle a high volume of transactions**, maintain inventory consistency, and support reporting queries.
		  ○ **Team has experience** with **SQL and NoSQL databases**.

www.cognixia.com

28

ADR | Components … continue

Cognixia

Component	Description	Purpose / Example
Decision	The **actual choice made**.	• Clearly states what was decided. • **Example:** ○ **Use PostgreSQL** as the primary database for transactional data and **Redis for caching** frequently accessed data.

Alternatives Considered	**Other options evaluated** and why they were rejected.	• Shows that multiple approaches were considered and why the chosen solution was selected.
		• **Example**
		  ○ **MySQL**: Pros – widely used, mature; Cons – less advanced features for complex queries.
		  ○ **MongoDB**: Pros – flexible schema; Cons – eventual consistency issues for critical transactions.
		  ○ **Cassandra**: Pros – highly scalable; Cons – complex setup and operational overhead.

www.cognixia.com

29

ADR | Components … continue

Cognixia

Component	Description	Purpose / Example

Consequences	**Positive and negative outcomes of the decision**, including impact on performance, cost, maintainability, and scalability.	• Helps assess risks, trade-offs, and future impact. **Example:**
		  ○ **Pros**: Strong ACID compliance for transactions, familiar tooling, good community support.
		  ○ **Cons**: Vertical scaling may be needed as load increases; caching logic must be implemented for high read performance.

Date	When the decision was made.	• Provides a timeline and helps in historical context. • **Example:** ○ 24-Nov-2025
Authors / Stakeholders	Who made or contributed to the decision.	• Ahmad Majeed Zahoory (Architect) • Tech Leads • DBA Team

www.cognixia.com

30

ADR | Example

Cognixia

Architecture Decision Record (ADR 007)

- **Title**: **Implement Load Balancer** Health Checks and Failover for API Gateway HA
- **Status**: **Accepted**
- **Context**:
  - The **API Gateway** is a **single point** of **failur**e and must **maintain high availability** (HA) to guarantee **service uptime**.
  - We need an HA solution that is **simple to implement**, **leverages standard cloud tooling**, and provides **near-instantaneous failover**. We are evaluating two options: a simple Load Balancer (LB) approach or a complex Active-Passive cluster.
- **Decision**:
  - We will **implement HA** for the **API Gateway** by **placing two identical instances** behind a **Load Balancer** configured **with health checks** and **automatic traffic routing** (Failover).
- **Alternatives Considered**:
  - **Primary/Secondary Active-Passive Cluster**
    - **Ruled out** due to significantly **higher complexity** in **setup**, configuration, and monitoring, as well as custom failover logic.

31

ADR | Example … continue

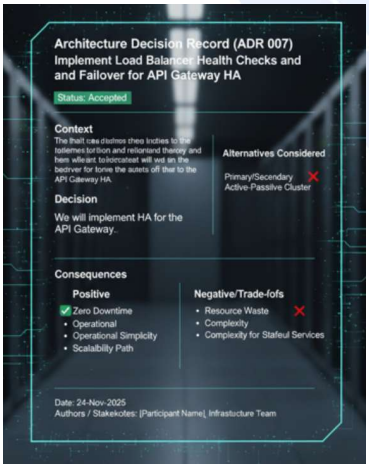
Cognixia

- **Consequences:**
  - **Positive:**
    1. **Zero Downtime on Failure: The LB automatically detects** a **failed instance** and **routes traffic** to the healthy one immediately, providing fast recovery time.
    2. **Operational Simplicity:** This method uses native, **managed cloud services** (Load Balancer), **reducing operational overhead** and custom scripting.
    3. **Scalability Path:** This architecture **easily supports scaling up from 2 to N instances** (Active-Active) when traffic demands increase.
  - **Negative/Trade-offs:**
    1. **Resource Waste:** In a **2-server setup**, **one server** is **technically idle during normal operation**, running up unnecessary compute costs compared to an Active-Active setup.
    2. **Complexity for Stateful Services: If** the **API Gateway** were **stateful** (it is currently stateless), this approach would **require sticky sessions**, adding complexity (but for stateless it's fine).
- **Date**: 24-Nov-2025
- **Authors / Stakeholders**: Ahmad Zahoory, Database Architects, Tech Leads

32

ADR | Example … continue

- **Sample**
  - The image is designed to look like a **digital document** or a well-structured **meeting output**, **emphasizing clarity** and organization



www.cognixia.com

33

34