Cognixia®

Design Concepts

By: Ahmad

1

Cognixia®

Intelligent Traffic Management

www.cognixia.com

2

Intelligent Traffic Management (ITM)

Cognixia

- Intelligent Traffic Management **refers** to the **automated distribution** and **routing** of **network traffic** (requests, users, or data) **across multiple servers**, **services**, or **locations** to **optimize performance**, **availability**, and **reliability**.
- It's "**intelligent**" because it **can make decisions dynamically** based on **real-time metrics** like **server load**, **latency**, **location**, or **availability**.

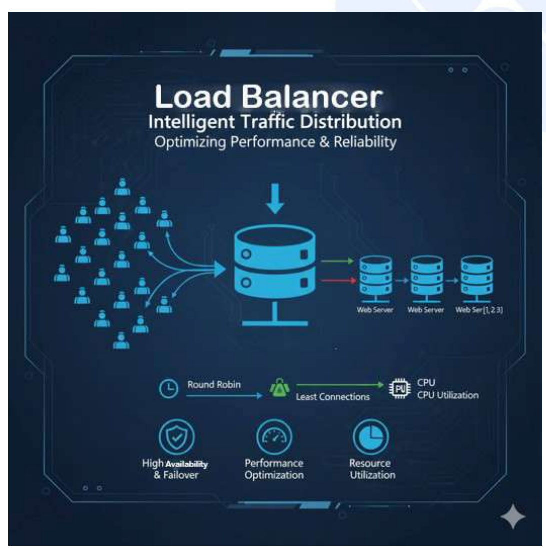
www.cognixia.com

3

What is Load Balancer

Cognixia

- Load balancing is a mechanism to **distribute incoming network traffic** across **multiple servers** or resources to **prevent overload** on **any single server** and ensure **high availability** and **reliability**.
- **Key Functions**
  - Distribute traffic
  - Monitor server health
  - Provide fault tolerance
  - Improve performance and user experience
- **Types of Load Balancers**
  - **Hardware Load Balancer**
    - Dedicated appliance (e.g., F5, Citrix NetScaler).
  - **Software Load Balancer**
    - Runs as software (e.g., NGINX, HAProxy).
  - **Cloud Load Balancer**
    - Managed service (e.g., AWS ELB, Azure Load Balancer).



www.cognixia.com

4

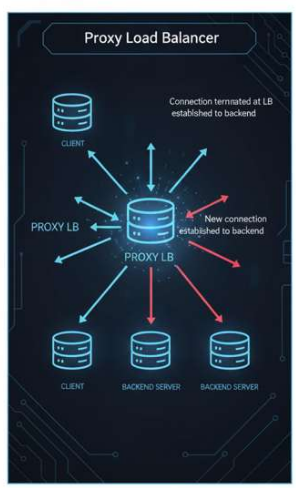
Proxy Mode and Pass-Through Mode

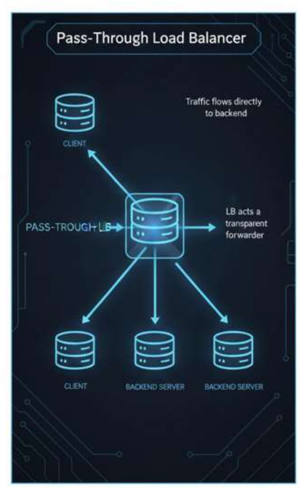
www.cognixia.com

5

Proxy vs. Pass-Through Modes

- The **operational mode dictates** how the **load balancer handles** the **client connection** and **network packets**.
- This is fundamentally **tied** to the **OSI layer** the LB operates on: **Layer 4 (L4)** or **Layer 7 (L7)**.
- **Types**
    - Proxy Load Balancer Modes
    - Pass-Through Load Balancer Modes
- The **choice between** Proxy Mode (*typically Layer 7*) and Pass-Through Mode (*typically Layer 4*) in load balancing depends entirely on the **features required** by **your application** and the performance demanded by your traffic.





www.cognixia.com

6

Cognixia

Load Balancer Strategies

www.cognixia.com

7

Cognixia

Load Balancing Strategies

- Load balancing **strategies** can be broadly **categorized** as **Static** (rule-based) or **Dynamic** (real-time and adaptive).
- ITM solutions typically **rely** heavily on **dynamic methods**, often **incorporating** Artificial Intelligence (**AI**) and Machine Learning (**ML**).

www.cognixia.com

8

Static Load Balancing Algorithms

Cognixia

- **Static algorithms** use **pre-set rules** and do not account for the current health or load of a server, though some can be "weighted" to account for capacity differences.
  - Round Robin
    - **Requests** are **distributed sequentially** to **each server** in the pool.
    - It's simple but fails to account for varying server capacities or current workloads.
  - Weighted Round Robin (WRR)
    - **Servers** are **assigned weights based** on their processing power or capacity.
    - Servers with **higher weights receive** a proportionally **greater number** of **requests**.
  - IP Hash (Source IP Hash)
    - The **load balancer performs** a **calculation** (hash) on the **client's IP address**.
    - This result **consistently m**aps that **client's requests** to the **same server**.
    - This is vital for **session persistence** where a user must **remain connected** to the **same server**.

www.cognixia.com

9

Dynamic Load Balancing Algorithms

Cognixia

- Dynamic algorithms use **real-time monitoring** of **server health** and **performance metrics** to make routing decisions on the fly, making them foundational to Intelligent Traffic Management.
  - Least Connection
    - Directs **new requests** to the **server** that c**urrently** has the **fewest active connections**.
    - This is highly effective for distributing uneven workloads.
  - Weighted Least Connection
    - An **extension** of **Least Connection** where the **server** with the **lowest ratio** of **active connections** to its **assigned capacity weight** receives the new request.
  - Least Response Time (or Least Time)
    - **Routes traffic** to the **server** that is **currently showing** the **fastest response time** (*often measured by combining active connections and response time to a health check*).
    - This is a strong strategy for optimizing user experience and minimizing latency.
  - Resource-Based (Adaptive)
    - This advanced method **uses specialized software** agents on **each server** to report **real-time metrics** like ***CPU usage***, ***memory consumption***, or ***available bandwidth***.
    - The load balancer then **routes traffic** to the **server** with the most **available free resources**.

www.cognixia.com

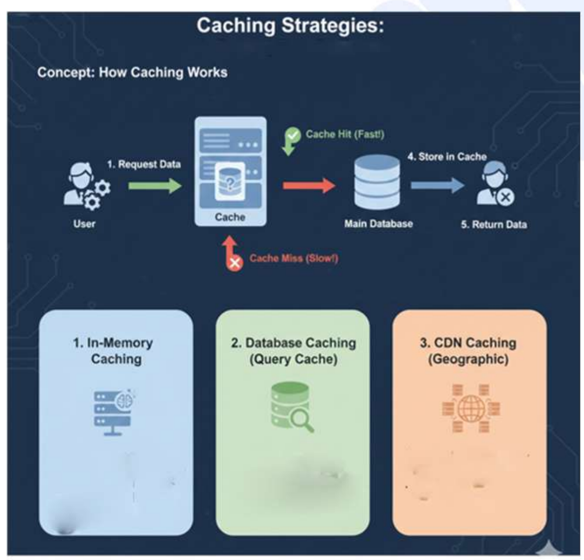
10

Improving Performance

What is Caching

- **Caching** is the process of **storing frequently accessed data in a faster, temporary storage location** (*cache*) so that future requests for the same data can be served faster—**without repeatedly fetching it from the original slow source** (database, disk, API, backend service).

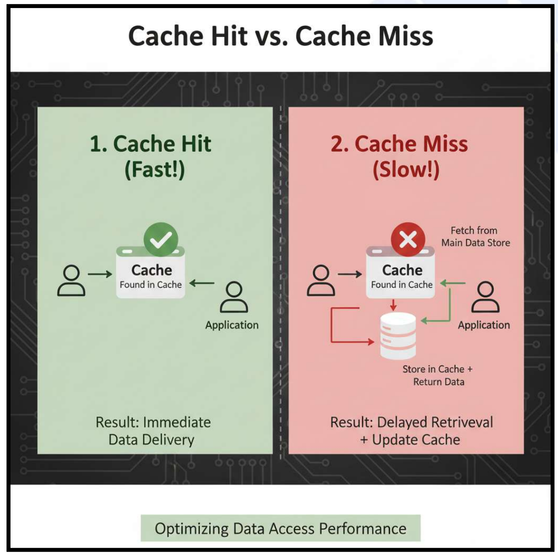
- **Why Use Caching**
  - **Faster Response**: **Reduces time** to **serve data** (*microseconds instead of seconds*)
  - **Reduces Load on Backend**: **Cuts** down database / API **calls**
  - **Cost Optimization**: **Fewer compute** and **database costs**
  - **Scalability**: **Handles more traffic** without increasing resources
  - **Better User Experience**: **Faster loading pages**, smoother apps



Caching … continue

- **How Caching Works**
  - Client Request → Cache? →
    - ✓ **Hit** → Return Cached Data (Fast)
    - ✗ **Miss** → Fetch from DB/API → Store in Cache → Return to Client

- **Service Providers**
  - **CDN**: Akamai, Cloudflare, Fastly, Edgio, StackPath, etc.
  - **In-Memory**: Redis, Memcached, Hazelcast, etc.



www.cognixia.com

13

Caching Types

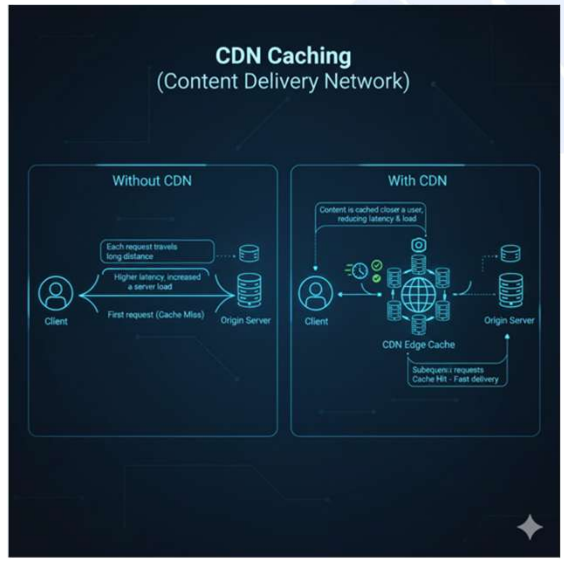
www.cognixia.com

14

Cache | Types … continue

Cognixia®

- Caches are **classified based** on where they **physically reside** within the system architecture:
  - CDN Caching (Content Delivery Network)
    - **Data stored** on **geographically distributed servers** (PoPs).
    - It **reduces** the **physical distance** content must travel, serving high-volume, global traffic.
    - When a **user requests** an **asset**, the **request** is **routed** to the **nearest PoP**.
    - If the **asset** is in the **PoP's cache** (***cache hit***), it's **served immediately**.
    - **If not** (***cache miss***), the **PoP fetches** it from the **original web server** (the origin) and **caches** it for **future requests**.

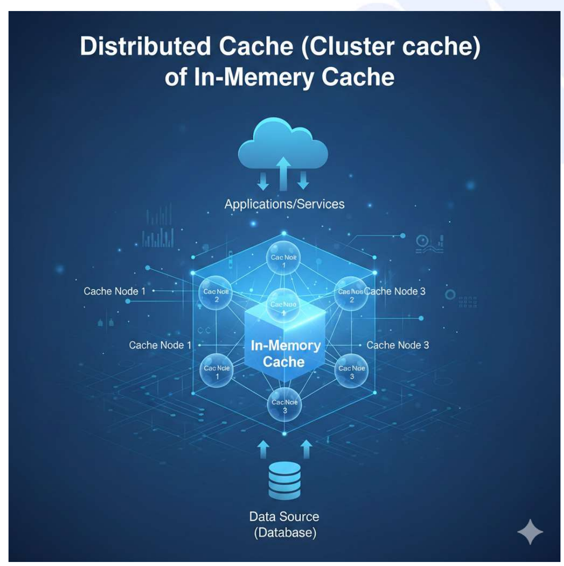


www.cognixia.com

Cache | Types

Cognixia®

- **Distributed Cache (Cluster cache)**
  - A **separate**, **dedicated cluster** of high-speed **servers** (*e.g., Redis, Memcached*) **accessible** over the **network.**
  - **In-Memory Cache**
    - ✓ The term In-Memory Cache **refers** to a **system** where **data** is **stored** entirely in **RAM** of **dedicated servers**.
    - ✓ **Data stored** in **RAM** is **accessed** orders of magnitude **faster than data stored** on **disks** (even SSDs).
  - Dynamic application data such as **user session data**, **database query results**, and computed **API responses**.
  - Unlike databases that store complex rows and tables, in-memory **caches** usually **store data** in **simple key-value pairs**, allowing for extremely **fast lookups** by key.



www.cognixia.com

Caching Strategies

17

Caching Strategies

- The strategy you choose **dictates** when **data** is **written** to the **cache** and how it is **updated**.
  - Cache-Aside (Lazy Loading)
  - Write-Through

www.cognixia.com

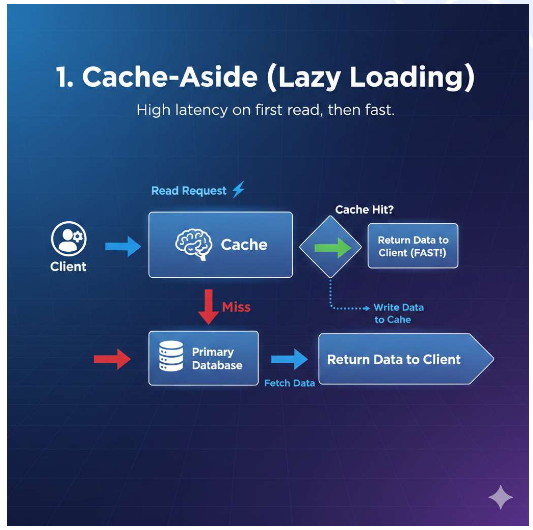
18

Caching Strategies

Cognixia

- The strategy you choose **dictates** when **data** is **written** to the **cache** and how it is **updated**.

- Cache-Aside (Lazy Loading)
  - Mechanism**:** The **application** or **client** is **responsible** for **checking** the **cache first**.
    - ✓ Cache Hit
      - ❖ If the **data** is found **returned immediately**.
    - ✓ Cache Miss
      - ❖ If the **data** is **not** in the **cache**, the **application fetches** it from the **database**, **returns** the data to the **client**, and then **writes** a **copy** of the data into the **cache** for future requests.
  - Pros
    - ✓ **Only requested data** is **cached**, preventing the cache from being filled with unused items.
  - Cons
    - ✓ **Higher latency** on the **first request** (*the cache miss*).
    - ✓ **Data** can **become stale** until a subsequent request updates it.
  - Best For
    - ✓ **Read-heavy workloads** where data access patterns are unpredictable.



www.cognixia.com

19

Caching Strategies … continue

Cognixia

- Write-Through
  - ✓ Mechanism
    - ❖ **Data** is **simultaneously written** to **both** the **cache** and the primary **database** *before* the operation is considered complete.
  - ✓ Pros
    - ❖ **Ensures data consistency** between the **cache** and the **database** at all times.
    - ❖ **Reduces latency** for future **reads**.
  - ✓ Cons
    - ❖ **Higher latency** on **write operations**, as the **application** must wait for both the **cache** and the **database writes** to **complete**.
  - ✓ Best For
    - ❖ Applications where **data consistency** and **immediate read availability after** a **write** are critical.

www.cognixia.com

20

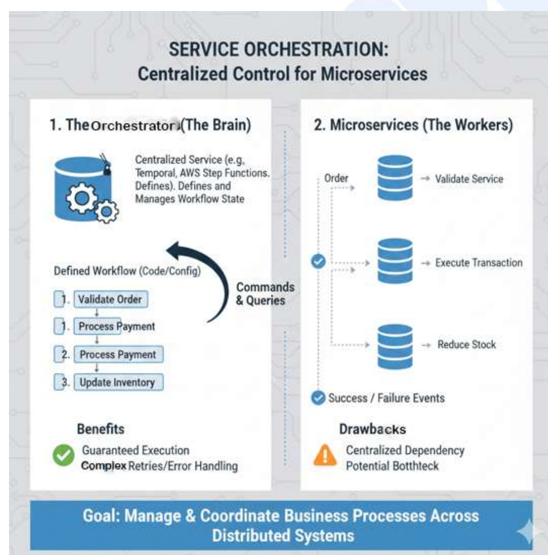
Service Orchestration

www.cognixia.com

21

What is Service Orchestration

- Service Orchestration is a **method** of **arranging** and **coordinating multiple independent services** (*often microservices*) to **execute** a complex business **process**es or **workflow**.
- It **dictates** the **flow of control** and the **sequence of actions** that must **occur across** various **services** to achieve a final goal.
- In essence, an **orchestrator** (*a dedicated service or component*) **takes** the **lead** and ***tells*** **each participant service exactly what to do** and **when to do** it.



www.cognixia.com

22

API Gateway

www.cognixia.com

23

API Gateway

- An **API Gateway** is a **central component** in modern software architectures (*especially microservices*) that **acts** as a **single point** of **entry** for all **external client requests**.
- It functions as a **reverse proxy** that **routes requests** to the **appropriate backend services**, while **simultaneously handling** common cross-cutting concerns like **securit**y, **caching,** and **traffic management**.
- **API Gateway** is the **single entry point** for all **client requests** into **backend services** (*web services, microservices, databases, authentication, etc.*) in modern architectures.

www.cognixia.com

24

API Gateway … continue

- It acts as:
  - Reverse Proxy
  - Traffic Router
  - Security Gate
  - Orchestrator of service calls

- **Supported By**: Kong Gateway, Tyk API Gateway, Apigee Gateway, etc.

- **Purpose:**
  - The **primary purpose** of an API Gateway is to **decouple clients** (*like web browsers or mobile apps*) from the complex, evolving architecture of the backend services.

Purpose	Detailed Usage
Request Routing	**Directs incoming requests** to the correct **internal microservice** based on the URL path, host, or other criteria.
Security Enforcement	Handles centralized tasks like **authentication** (*verifying API keys or tokens*) and **authorization**, shielding backend services from external threats.
Traffic Management	Implements **rate limiting**, throttling, and load balancing across internal service instances.
Request Aggregation	Combines responses from multiple backend services into a single, cohesive response for the client (*known as **API Composition** or **Backend For Frontend - BFF***).
Quality of Service	**Handles** service discovery, **logging**, **monitoring**, and applying Quality of Service (*QoS*) controls.

25

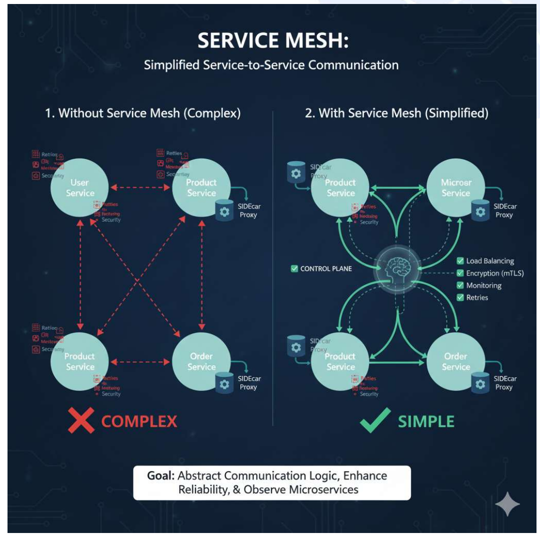
Service Mesh

www.cognixia.com

26

What is Service Mesh

- A **Service Mesh** is an **infrastructure layer** that manages **service-to-service communication** in a **microservices architecture**.
- Instead of **embedding networking logic** (*like retries, security, observability*) **inside each service**, the **mesh handles** it transparently via **sidecar proxies** (*e.g., Envoy*).
- Think of it as the **"traffic control system" inside your cluster**, ensuring **services talk** to **each other** reliably, securely, and observably.
- **Feature**
  - **Secure communication**: **Encrypts service-to-service traffic** using mTLS (Mutual TLS)
  - **Traffic management**: **Load balancing, retries, timeouts, circuit breakers**
  - **Observability**: Tracks **metrics, logs, traces** of each request
  - **Policy management**: **Controls access rules, rate-limiting, quotas**
  - **Resilience**: Ensures **services keep running even during failures**



www.cognixia.com

27

Service Mesh … continue

- The **Service Mesh** architecture **consists** of **two main parts**:
  - **Data Plane**
    - The actual network of **proxies that intercepts** and **handles** all **service traffic**.
    - **Mechanism**
      - ✓ **Every instance** of a microservice is **paired** with a **Sidecar Proxy**.
      - ✓ All **incoming** and **outgoing network traffic** for that service instance **goes through** its dedicated **sidecar proxy**.
    - **Examples**: Envoy (*the most popular open-source sidecar proxy*).
  - **Control Plane**
    - The **central brain** that **manages**, **configures**, and **monitors** all the **sidecar proxies** in the **Data Plane**.
    - **Mechanism**
      - ✓ It **converts high-level policies** (*defined by the operator*) **into** concrete **configurations** (*like routing rules and mTLS certificates*) and **pushes them** out to the **sidecars**.
    - **Examples**: Istio, Linkerd, Consul Connect.

www.cognixia.com

28

Cognixia

Session Management

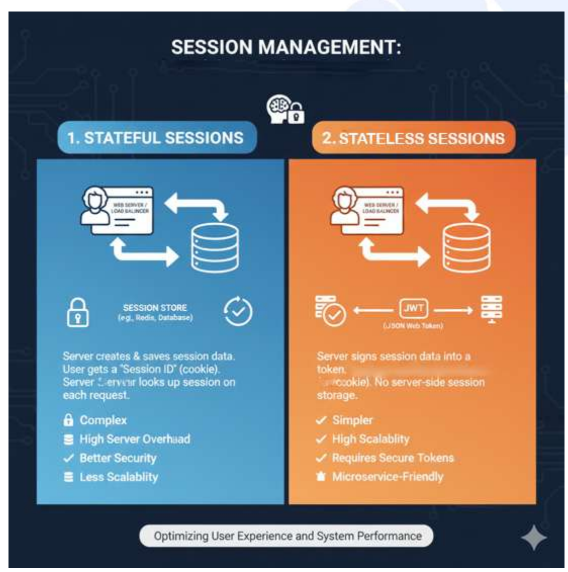
www.cognixia.com

29

Cognixia

What is Session Management

- User Session Management is the **process** of **tracking** a **user's activity** and **status** (*their "state"*) across multiple, non-contiguous requests to a server.
- Since **HTTP** is **inherently stateless** (*each request is treated as brand new*), the **system must have** a **strategy** to **preserve** the **user's context** (*e.g., login status, shopping cart, preferences, role, authentication, etc.*).
- The **two primary architectural patterns** for doing this are **Stateful** and **Stateless** sessions.



www.cognixia.com

30

Stateful Session

31

Stateful Session Management

- In a **Stateful system**, the **server** (*or an external store managed by the server*) is **responsible** for **retaining** all **information** about the **client session**.

- **How it Works**
  1. **Login**
     - The **user logs** in to the server.
  2. **Server Creates State**
     - The **server generates** a **unique**, unguessable **Session ID** (*e.g., a random string like aB7c4D9e*).
     - It **stores** all **session data** (*User ID, Role, Cart items*) locally in its **own memory**, a **database**, or a **dedicated session store** (*like Redis*).
  3. **Client Stores ID**
     - The **server sends** this **Session ID back** to the **client**, usually **embedded** in a **Cookie**.
  4. **Subsequent Request**
     - The **client sends** the **Cookie** (*containing the Session ID*) with **every subsequent request**.
  5. **Server Lookup**
     - The **server receives** the **ID** and must **perform** a **lookup** in its **session store** to **retrieve** the **user's full state**.

32

Stateless Session

33

Stateless Session Management

- In a **Stateless system**, the **client holds** the **state information**, which is **encoded** and **cryptographically** signed by the server. The **server retains no session record**.

- **How it Works (Using JWT)**
  - The most **common implementation** uses **JSON Web Tokens** (JWT).
    1. **Login**
       - The **user logs** in to the server.
    2. **Server Creates Token**
       - The **server creates a JWT**, **embedding essential user claims** (*e.g., user_id, role, expiry_time*) into the **token's payload**.
       - The **server** then **signs** the **token** with a **secret key**.
    3. **Client Stores Token**
       - The **server sends** the complete, self-contained **JWT back** to the **client** (*usually in local storage or an HTTP header*).
    4. **Subsequent Request**
       - The **client sends** the **JWT** with **every subsequent request**.
    5. **Server Verification**
       - The **server receives** the **JWT** and simply **verifies** the **cryptographic signature** using its secret key.
       - If the **signature** is **valid**, the data inside the token is trusted, and the **server** can **process** the **request** without looking up any database.

34

Cognixia

Fair Usage

www.cognixia.com

35

Cognixia

What is Fair Usage

- Fair usage ensures that **no single user, system, or service consumes more than their allowed share of resources**, preventing misuse, overloading, and ensuring availability, performance, and cost control for all users.
- This is commonly enforced using:
  - **Rate Limiting**
  - **Throttling**

www.cognixia.com

36

Rate Limiting

Rate Limiting

- It **sets a strict limit** on the number of requests a user or system is allowed to make **within a time period** (per second, minute, hour, or day).
- If the limit is crossed, the system **blocks additional requests immediately**, usually returning **HTTP 429**
- **Focus**: Enforcement of a **hard quota** or **cap** on **usage**.
- **Goal**: **Protect** the **service** from **abuse**, such as **DDoS attacks**, brute-force login attempts, and unauthorized data scraping, by denying requests outright.
- **Purpose**
  - **Prevent Abuse** → **Stops API spamming**, bots, or DDoS attacks.
  - **Ensure Fair Usage** → **Divides API usage fairly** among all users.
  - **Control Costs** → **Protects backend resources** and cloud cost overruns.

Cognixia

Throttling

www.cognixia.com

39

Cognixia

Throttling

- Throttling **slows down the request processing rate**, instead of blocking instantly.
- It **queues or delays** extra requests instead of rejecting them immediately.
- **Focus**: **Regulation** of the **traffic flow** to **match** the **server's processing capacity**.
- **Goal**: **Manage traffic surges** gracefully, maintain stable service performance during peak loads, and provide a better user experience by avoiding abrupt rejections.
- **Purpose:**
  - Avoid server overload → Smooths out request spikes instead of blocking users
  - Maintain performance → Ensures backend stays stable during peak load
  - Graceful Degradation → Allows requests but at slower speed

www.cognixia.com

40

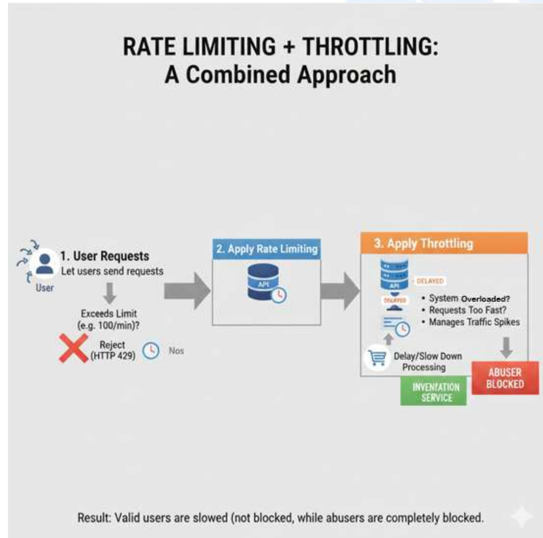
How to Use Both Together

- Rate Limiting and Throttling can **both be used together** in a single program or system.

- **High-Level Logic**:
  1. Let the user send requests
  2. First apply **Rate Limiting**:
     - If **user exceeds allowed limit** (*e.g., more than 100 requests/minute*) → **reject (429)**

  3. If **user is within limit**, but **requests are coming too fast or system is overloaded** → Apply **Throttling** → **Delay** or **slow down processing**

  Result: **Valid users** are **slowed** (not blocked), while **abusers** are **completely blocked**.



www.cognixia.com

41

Asynchronous Communication

www.cognixia.com

42

What is Asynchronous Communication

Cognixia

- Asynchronous communication is a **system design approach** where a **sender sends** a **message** or a request **without waiting** for an **immediate response**.
- The **sender continues its work**, and the **receiver processes the message independently** at its own pace.
- Asynchronous communication means systems or components **do not wait for each other to respond**.
- They **send a message and continue their work**, allowing **high performance, loose coupling, scalability, and resilience**.
- Instead of calling services directly and waiting (synchronous), they communicate **via messages or events**.
- **Analogy**
  - **WhatsApp Message** → You **send a message** and **continue working**. The **receiver reads it later**.
  - **Food Delivery App** → You **order food**, and **restaurant prepares** it independently — **you don't wait** on the phone.
  - **Courier Service** → You **give a package to courier**; they **deliver it asynchronously**.

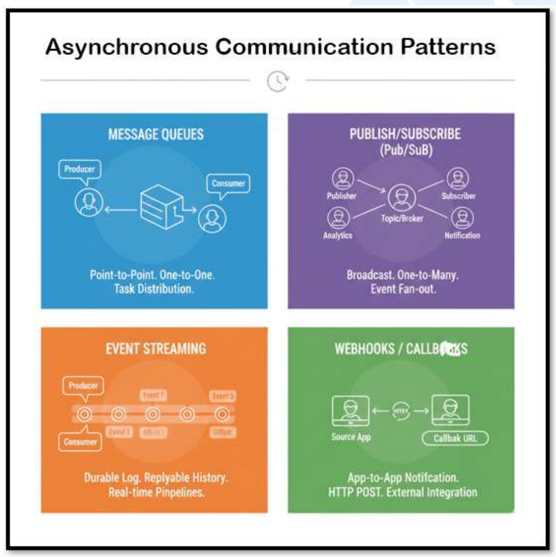
www.cognixia.com

43

Asynchronous Communication | Types

Cognixia

- The **type**s are primarily **categorized** by the **communication model** used—how many receivers get the message and how the message is handled.
  - **Message Queues**
    - (Point-to-Point)
  - **Publish/Subscribe (Pub/Sub)**
    - (One-to-Many)
  - **Event Streaming**
    - (Durable Log)
  - **Webhook / Callback**
    - (HTTP Push)



www.cognixia.com

44

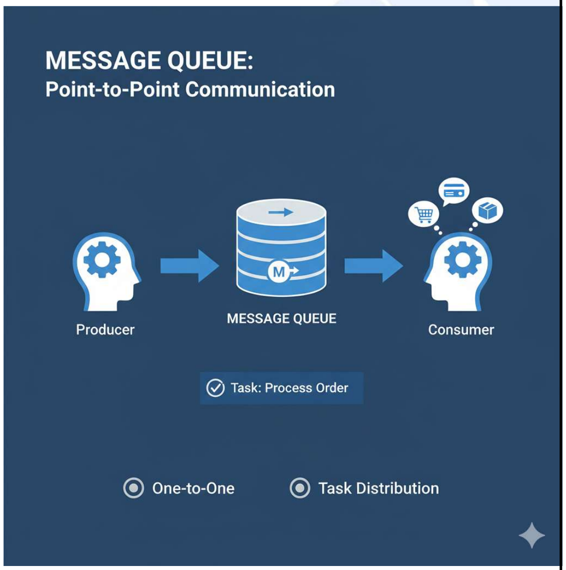
Message Queue

45

Message Queues

- A **Message Queue (MQ)** is a form of asynchronous, point-to-point communication where a sender (**Producer**) places a message onto a **queue**, and a single recipient (**Consumer**) retrieves and processes it.

- **Key Characteristics**
  - **Communication Model**
    - **One-to-One (Point-to-Point)**.
  - **Message Consumption**
    - A **message** is **consumed** by **exactly one** of the **consumers listening** to that **queue** and is then typically removed from the queue.
  - **Ordering**
    - Often maintains a **FIFO (First-In, First-Out)** order of messages for a single consumer.



46

Message Queues … continue

- **Technologies**: **RabbitMQ**, **IBM MQ**, **IronMQ**, **Apache ActiveMQ**, Beanstalkd, Gearman
- **Use Case**
  - **Order Processing in E-commerce: Order** service **puts orders** into a **queue**; **Inventory**, Payment **service reads it** one by one
  - **Resize Images in background: User uploads image → queued → worker resizes** asynchronously
  - **Email/SMS Notifications: Message goes to queue** and **notification service processes** it
- **Example**
  - **E-commerce Order Processing** (e.g., using **Rabbit MQ** or **IBM MQ**).
    - When a **user clicks** "**Place Order,**" the web **service immediately puts** an **OrderPlaced message** into a **queue** and r**eturns** a **confirmation** to the user.
    - A separate **fulfillment service picks up** the **message** from **the queue**, **processes** the **inventory check**, **payment**, and **shipping label generation**, without blocking the user interface.

Event-Driven Architecture (EDA)

What is Event-Driven Architecture (EDA)

Cognixia

- Event-Driven Architecture (EDA) is a **software design pattern** where the entire system's **behavior** is **centered** around the production, detection, consumption, and **reaction** to **events**.
- **Instead** of **services making direct**, **synchronous calls** to one **another** and **waiting** for a **response** (*the traditional request-response model*), they simply **announce** that a significant **change of state** (*an event*) has **occurred**.
- **Other services that** are **interested** in **that change** then **react asynchronously** and **independently**.
- The core goal of EDA is to achieve **loose coupling**, **high scalability**, and **real-time responsiveness**.
- **Core Models of EDA** (Communication Style)
  - **Publish/Subscribe (Pub/Sub)**
  - **Event Streaming**

www.cognixia.com

49

Cognixia

Publish/Subscribe (Pub/Sub)

www.cognixia.com

50

Publish/Subscribe (Pub/Sub)

Cognixia

- **Publish/Subscribe (Pub/Sub)** is a **broadcast model** where a sender (**Publisher**) sends a **message** (*called an* ***event***) to a named **channel** or **Topic**, and all interested recipients (**Subscribers**) simultaneously receive a copy of that event.
- Unlike queues, **message is not consumed once**, it's delivered to **all subscribers**.
- **Key Characteristics**
  - **Communication Model**
    - **One-to-Many (Broadcast)**.
  - **Message Consumption**
    - A **message** is **delivered** to **all** services **subscribed** to that **topic**.
  - **Decoupling**
    - **Publishers** and **Subscribers** have **no knowledge of each other**, communicating only through the central **broker/topic**.

www.cognixia.com

51

Cognixia

Event Streaming

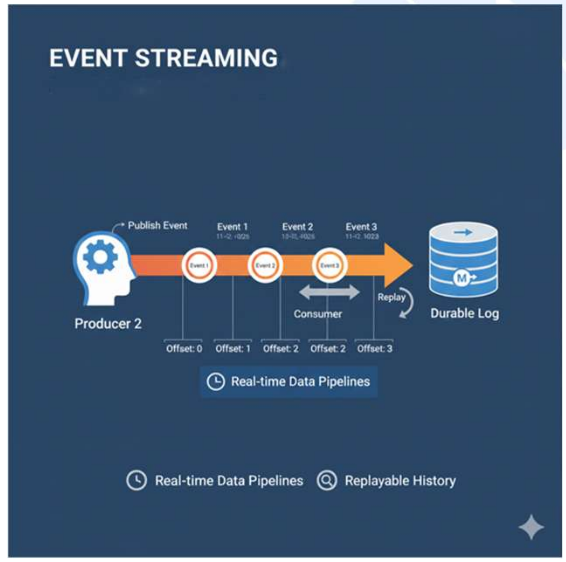
www.cognixia.com

52

Cognixia

Event Streaming

- **Event Streaming** is an advanced, powerful approach to data management that treats data as a **continuous, real-time, ordered, and durable stream of events**. It is one of the primary types of Event-Driven Architecture (EDA).
- **Event Streaming is the continuous collection, processing, and delivery of real-time events (data) as they happen.**
- **Instead** of **storing** and **sending data later**, systems **stream events instantly** to **consumers** that want to **react immediately**.
- It enables **Real-time processing**, **Live monitoring**, **Analytics**, and **Event-driven systems**.



www.cognixia.com

53

Cognixia

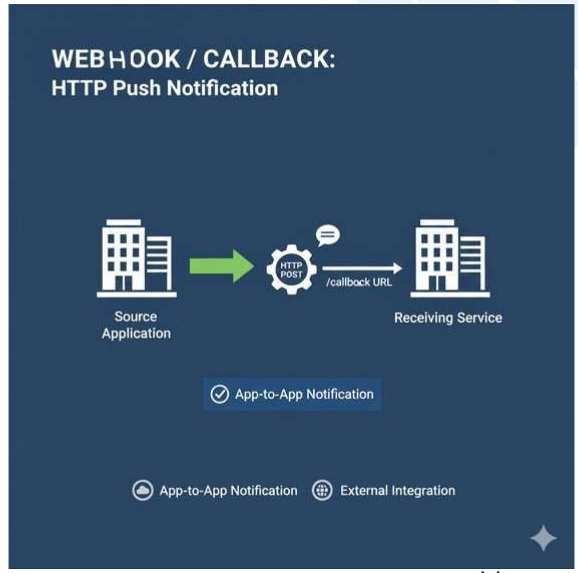
Webhook

www.cognixia.com

54

What is Webhook

- A **Webhook is a web-based callback**, where a system sends an **HTTP request (usually POST)** to your URL when **a specific event happens**.
- **What is a Callback**
  - A **callback** is a mechanism where **you give your function (URL/code) to another system**, and that system will **call you back later when something happens**.
- **Features**
  - It is **event-driven, one-way, push-based communication.**
  - Unlike API polling, you **do NOT repeatedly call API to check updates**.
  - Instead, the provider **pushes data automatically** to your endpoint.



55

Comparison

56

Comparison

Feature / Category	Message Queue	Pub/Sub	Event Streaming	Webhook
Purpose	Reliable **task processing** with guaranteed delivery	Real-time **event broadcasting** to multiple subscribers	**Continuous real-time stream processing** & analytics	**Notify external system** when an event happens
Communication	Async, Push or Pull	Async, Push	Async, Push or Pull	Async, Push
Message Delivery	One-to-One	One-to-Many	One-to-Many (Replayable)	One-to-One or One-to-Many
Real-Time	Near real-time	Real-time	Real-time (continuous)	Real-time
Use Case	**Background job processing**	**Notifications & broadcasting**	**Analytics, tracking, IoT, logging**	**Payment updates, CRM integration**
Industry Example	**E-commerce Order Processing**	**WhatsApp broadcast, stock alerts**	**Fraud detection, GPS tracking**	**Stripe payment callback**

www.cognixia.com

57

Database Design

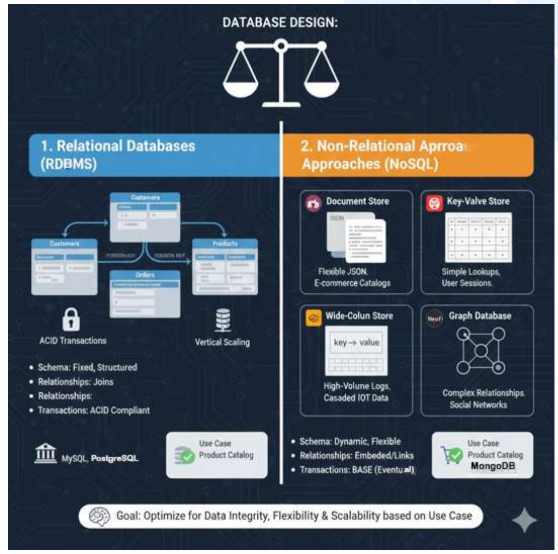
www.cognixia.com

58

What is Database Design

Cognixia

- **Database Design** is the **process** of **structuring** the **data** and **defining the relationships** between **different data elements** to support the **business needs** of an application.
- The goal is to **produce** a **model** that ensures data **integrity**, minimizes **redundancy**, and optimizes for **performance** (**speed of queries**).
- The fundamental decision in database design is choosing the underlying data model, primarily falling into **two major categories**:
    - Relational
    - Non-Relational (NoSQL)



www.cognixia.com

59

Cognixia

Relational Databases (RDBMS)

www.cognixia.com

60

Relational Databases (RDBMS)

Cognixia

- Relational databases **store data** in **tables** with **fixed schemas** (columns) and **rows**.
- They are based on the **relational model** and rely heavily on structured **relationships defined** by **keys** (*primary and foreign*).
- **Key Characteristics**
  - **Schema**
    - **Fixed** and **Structured** (*predefined columns and data types*).
    - Changes require migrations.
  - **Relationships**
    - **Defined** through **Foreign Keys** and **Joins** (*e.g., a customer is linked to their orders*).
  - **Transactions**
    - **Adheres** to **ACID properties** (*Atomicity, Consistency, Isolation, Durability*), making them **ideal** for **financial transactions** and systems **requiring high data integrity**.
  - **Scaling**
    - **Primarily scales vertically** (*adding more CPU/RAM to a single server*), though techniques like **sharding allow** for **horizontal scaling**.

www.cognixia.com

61

RDBMS … continue

Cognixia

- **When to Choose RDBMS**

Condition	Choose SQL?
High data consistency required	**Yes**
Complex relationships	**Yes**
Transaction rollback needed	**Yes**
Scalability needed	**Hard** via vertical scaling
Unstructured data	**No**

- **Popular Databases:** MySQL, PostgreSQL, Oracle, SQL Server, MariaDB

www.cognixia.com

62

Non-Relational (NoSQL)

63

Non-Relational Approaches (NoSQL)

- NoSQL databases encompass various **data models that deviate** from the **traditional table structure**.
- NoSQL databases store **unstructured, semi-structured, or high-volume data** using flexible data models.
- Focuses on **speed, scalability** and **schema flexibility**.
  - No fixed schema
  - Horizontally scalable (distributed)

64

NoSQL | Types

Type	Data Structure	Best Use Case	Industry Example
Document Store	Stores **data** as flexible, **semi-structured JSON documents**.	**Content Management, Catalogs, Profiles:** Data structure changes frequently, and data is read/written as a single unit.	**MongoDB, Couchbase** used for storing e-commerce product catalogs.
Key-Value Store	Simple **map** of a **unique key to a value** (*which can be any object*).	**Caching, Session Management, Real-Time Data:** Requires extremely fast, non-complex lookups.	**Redis, Memcached** used for storing user session tokens.
Wide-Column Store	**Stores data in tables**, but **columns can vary from row to row**; optimized for high availability and big data.	**Time-Series Data, Big Data Analytics, High-Volume Logging.**	**Cassandra, HBase** used for logging data from millions of IoT devices.
Graph Database	**Stores data as nodes** (entities) and **edges** (relationships).	**Social Networks, Recommendation Engines, Fraud Detection:** Relationships are the most important part of the data.	**Neo4j** used for storing a social graph (who follows whom).

www.cognixia.com

65

NoSQL | Horizontal Scaling

- In NoSQL databases, **horizontal scaling** is **achieved** primarily through **sharding** and **replication**.
  - **Sharding (Data Partitioning)**
    - Sharding is the **process** of **splitting** the **entire dataset into smaller, independent chunks called shards** or **partitions**.
    - Mechanism
      - **Each shard** is **placed** on a **separate server** (*node*).
      - When a **new server** is **added** to the cluster, the **system automatically redistributes** some **data** from the **existing nodes** to the **new node**.
    - Result
      - A **single query** only **needs** to **hit** the **server holding** the **relevant partition**, drastically improving performance under high load.
  - **Replication (High Availability)**
    - **Replication ensures** that **multiple copies** of the **data exist across different nodes**.
    - Mechanism
      - Even though the **data** is **sharded**, **each piece** of **data** is typically **copied** to **several** other **nodes**.
    - Result
      - If a **node fails**, the system can immediately **retrieve** the **data** from a **replica** on another active node, ensuring high availability and fault tolerance.
  - **Distributed Operations**
    - When an **application** needs to **read or write** data, the **request goes** to the **cluster**, and the system's **distribution coordinator** knows exactly **which nodes hold** the required **data** (*or its replica*) and **directs** the **request** accordingly.

www.cognixia.com

66

NoSQL | Consistency

Cognixia

- **Eventual Consistency**
  - In a distributed NoSQL system, a **write operation** must be **replicated across multiple nodes** (servers) to ensure high availability and durability.
  - The **consistency issue arises** because the **system allows clients** to **read data before** all **replicas** have been **successfully updated**.
  - **The Issue**
    - **Stale Reads**
      - **After** a **successful write** to **one node**, a **client reading** the **data** from a **different**, yet-to-be-updated **node** will **receive** the **old** (*stale*) **data**.
    - **Conflict**
      - **If multiple clients write** to the **same data** on **different nodes simultaneously**, the **system** has a **conflict** and **must determine which write wins**.

www.cognixia.com

67

Cognixia

Distributed Relational Database

www.cognixia.com

68

What is Distributed Relational Database

Cognixia

- **Distributed Relational Database** (*DRDB*), often referred to as **Distributed SQL**, is a database system that **combines** the **relational data model** (*tables, schemas, SQL*) and **ACID transactional guarantees** of a traditional RDBMS with the horizontal scalability, fault tolerance, and global distribution of distributed systems.
- Unlike a **traditional database** that runs on a **single server**, a **DRDB spreads** (*or shards*) and **replicates** its **data across multiple independent servers** (*nodes*) connected by a network.
- This makes the system appear as a single, logical database to the application, while the distributed architecture handles the complexities of data placement, synchronization, and query execution across the cluster.

www.cognixia.com

69

Working of Distributed Relational Database

Cognixia

- **Key mechanisms** that enable a DRDB to function:
  - **Data Sharding (Partitioning)**
    - The **database automatically breaks up large tables into smaller**, **manageable chunks** (*shards*) and **distributes** them **across different nodes** in the cluster.
    - This **allows** for **parallel processing of queries**, improving performance for massive datasets.
  - **Replication**
    - **Copies** of **data** (*replicas*) are **stored on multiple nodes**.
    - If **one node fails**, the system **automatically redirects traffic** to a **replica**, ensuring high availability and fault tolerance.
  - **Distributed ACID Transactions**
    - DRDBs **use sophisticated consensus algorithms** (*like Raft or Paxos*) to ensure that **transactions maintain ACID** (*Atomicity, Consistency, Isolation, Durability*) **properties**, even when a single transaction involves data spread across several nodes.
  - **Horizontal Scalability**
    - Performance and **capacity can be increased** simply by **adding more nodes** to the cluster, rather than upgrading the hardware of a single server (*vertical scaling*).

www.cognixia.com

70

DNS

www.cognixia.com

71

What is DNS

- The **Domain Name System (DNS)** is a core internet service that translates human-readable domain names into computer-readable IP addresses.
- Essentially, it acts as the **"phonebook of the internet,"** allowing users to connect to websites and services without memorizing complex numerical addresses.
- DNS is not just about **translating web addresses**; it is a critical component for modern, scalable, and resilient application design.
- **Connection Translation (The Core Function)**
  - Problem
    - ✓ Servers are addressed using IP addresses (*e.g., IPv4 or IPv6*) which can change, especially in cloud environments.
    - ✓ **Users only remember domain names**.
  - DNS Solution
    - ✓ It provides the mechanism for a client (*web browser, mobile app, etc.)* to query the IP address of the target server.
    - ✓ This decouples the service's location from its name.

www.cognixia.com

72

DNS Routing

- DNS routing, often simply called **DNS-based traffic steering** or **load balancing**, is the practice of **using** the **Domain Name System** (DNS) to intelligently **direct a user's connection** request to a **specific server**, **IP address**, or **endpoint** based on criteria **beyond simple name-to-address translation**.
- It's a crucial **technique** in modern **system architecture** for **achieving high availability, fault tolerance, low latency, and geographic compliance**.

Strategy	Routing Logic	Primary Result & Goal	Use Case Example
Geo-based Routing	Analyzes the **source IP address** to determine the user's geographical location.	Returns the IP address of the server or CDN endpoint that is **geographically closest** to the user. **Goal: Low Latency & Compliance.**	Directing users from Asia to the Singapore data center for faster load times.
Latency-based Routing	Constantly measures the **actual network latency** between the user's region and various server endpoints.	Returns the IP address of the endpoint that currently offers the **lowest measured latency**, regardless of physical distance. **Goal: Optimal Performance.**	Sending a user to a slightly further but less congested server that offers a quicker response time.
Failover Routing	Continuously runs **health checks** against the application servers to monitor their operational status.	If the primary server fails a health check, its IP is removed, and the IP of the designated **secondary (failover)** server is returned. **Goal: High Availability & Resilience.**	Automatically switching all traffic from Data Center A to Data Center B when A suffers a power outage.
Weighted Routing	Assigns a numerical **weight** (e.g., 80/20, 50/50) to each available IP address based on desired traffic distribution.	Distributes traffic proportionally according to the defined weights. **Goal: Controlled Deployment & Load Balancing.**	Sending 10% of users to a new version of the application (Canary Deployment) before rolling it out widely.

73

Storage

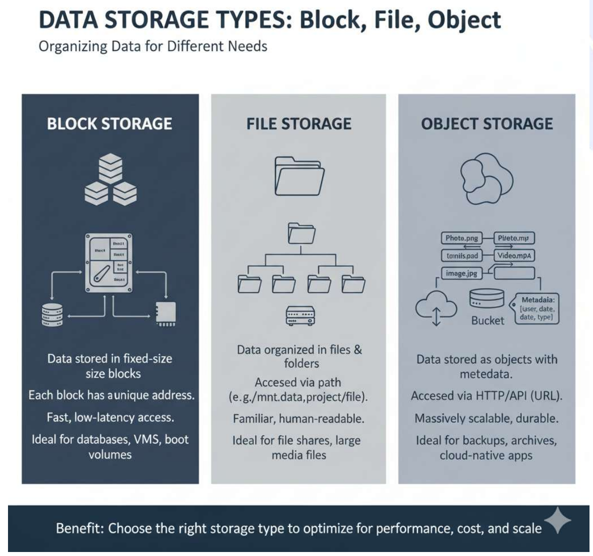
www.cognixia.com

74

What is Storage

Cognixia

- Storage in system architecture refers to the components and technologies responsible for digitally **recording and retaining data** for immediate or future use.
- It is a fundamental element of any computing system, determining how much data can be kept, how quickly it can be accessed, and how reliably it is protected.
- The architect's view focuses on three main dimensions: **Performance/Access**, **Sharing/Networking**, and **Durability/Cost**.



www.cognixia.com

75

Performance and Access Types

Cognixia

- **Select storage based** on the **required speed** and the fundamental way **data** is **accessed**.

Access Type	Description	Key Metric	Typical Technology	Use Case Example
Block Storage	• **Data is stored** in **fixed-size blocks**, each with a unique address, allowing for highly efficient, direct read/write access. • The operating system sees it as a raw disk.	**IOPS** (I/O Operations Per Second) and **Latency**	**SAN, Local SSD/NVMe**	Databases, Virtual Machines (VMs), boot volumes.
File Storage	• **Data is stored** as **files** and **directories**, complete with metadata (permissions, last modified date). • Access is hierarchical.	**Throughput** and **Latency**	**NAS (NFS/SMB)**	User home directories, centralized file shares, content repositories.
Object Storage	• **Data is stored** as self-contained **"objects"** (files plus all their metadata) in a **flat**, massive pool, accessed via HTTP APIs. • Highly scalable but slower.	**Scalability** and **Durability**	**Cloud Storage (S3, GCS)**	Data lakes, backups, static web content, media libraries.

76

77