Cognixia®

System Architecture

By: Ahmad

1

Cognixia®

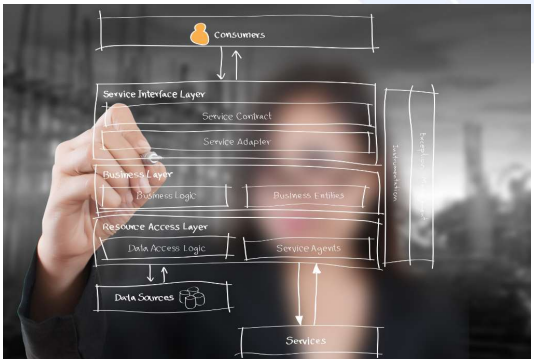
Software Architect

www.cognixia.com

2

What is Software Architecture

- Software Architecture is the **high-level structure** of a **software system** that **defines** how **components interact**, how **data flows**, and how the **system meets** its **requirements**.
- It's like the **blueprint** of a **building** but for **software**, **showing** the **components**, their **relationships**, and **design principles**.
- Software Architecture **determines** how a **system is organized**, how **modules communicate**, and how it can **meet** system's **Quality Attributes** (*non-functional requirements*) like **performance**, **security**, and **scalability**.
- **Key Points**:
  - **Focuses** on **structural design**, not implementation details.
  - **Balances technical** and **business requirements**.
  - **Helps** in **maintaining**, **scaling**, and **evolving software efficiently**.



www.cognixia.com

3

System Architect

www.cognixia.com

4

What is System Architecture

Cognixia

- System Architecture is the high-level design of any software system, where we decide:
  - What the parts or components of the system will be
    - Identify major building blocks (services, databases, UI, API, etc.)
  - How those components will interact and connect with each other
    - Explain communication: API, messaging, queues, protocols, gateways
  - And how the entire system will function as a whole
    - Show full flow from user action to backend processing to output
- The main purpose of System Architecture is to convert a complex system into an understandable structure, so that it can be designed, developed, and maintained easily.

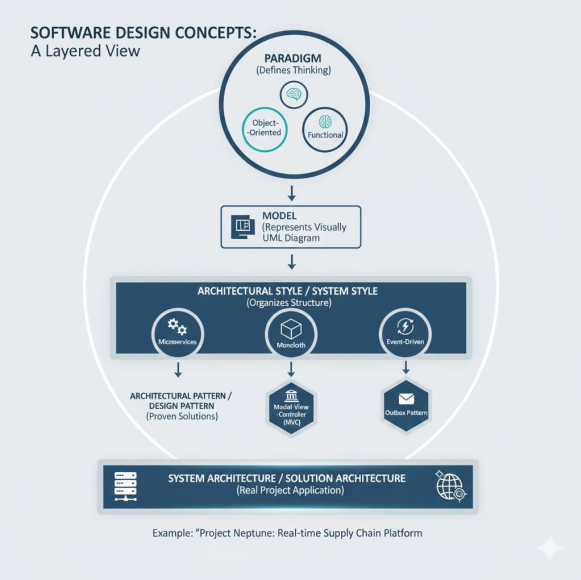
www.cognixia.com

5

Architecture | Components

Cognixia

- When we design software systems, we often hear terms like:
  - **Paradigm** (defines thinking)
  - **Model** (represents it visually)
  - **Architectural Style** (organizes the structure)
  - **Architectural Pattern/Design Pattern** (apply proven solutions)
  - **Solution Architecture** (apply everything in a real project)
- They are connected, sometimes used interchangeably.



6

Architecture | Example

- **Example: Online Food Delivery System (Swiggy / Zomato / Uber Eats).**

Concept	Simple Explanation Using Food Delivery Example
Paradigm (*How we think and write code?*)	We think • First **login** → then **select food** → then **pay** → then **track delivery**. • In terms of **Customer**, **Restaurant**, **Order**, **Delivery Boy** • When **OrderPlaced**, **system sends confirmation**. • When **FoodPickedUp**, **tracking starts**.
Model (*How we represent the system visually?*)	Draw diagrams like • **User interacts with restaurant → places order → tracking** Or • **database model:** Customer table, Restaurant table, Order table.
Architectural Style (*What is the general structure of the system?*)	It follows **Client-Server**: • App/mobile (client) talks to backend server Or • **Microservices**: separate services for Order, Payment, Delivery, Notification.
Architectural Pattern / Design Pattern (*Reusable small solution inside architecture*)	• Use **Observer** to **send order notifications** • **Circuit Breaker** for **failed payment gateway** • **Saga** to manage **distributed order-payment transaction**.
Solution Architecture (*Complete blueprint of solution*)	• Apply everything • Decide how **services connect** (API, Kafka), **technology** (Java, React, AWS), **database** choices, **security**, **scaling**, **fault tolerance**, **deployment model** (Docker/Kubernetes).

7

Paradigm

www.cognixia.com

8

What is Paradigm

- A paradigm is a **way of thinking** or a **philosophy** used to **solve a problem**.
- It defines how we perceive and approach designing software or systems.
- Example:
  - "**Divide big problems into smaller parts**" → Modular Paradigm
  - "**Make small independent services**" → Distributed Paradigm
- **Types:**
  - **Procedural Paradigm** (Solve problems step by step)
  - **Object-Oriented Paradigm** (Group data and behavior into objects)
  - **Functional Paradigm** (Use pure functions, avoid side effects)
  - **Event-Driven Paradigm** (React to events like clicks, messages)
  - **Declarative Paradigm** (Focus on what to do, not how)

www.cognixia.com

9

Paradigm Types

Paradigm	Easy Explanation	When to Use (Simple)	Real-World Example (Simple)
Procedural Paradigm	**You write steps in order** (do step 1 → step 2 → step 3). Focus is on *how* things happen.	When work is predictable and follows a fixed sequence.	Daily log processing script, ETL pipeline (read → clean → load).
Object-Oriented Paradigm (OOP)	**You create objects** (like Customer, Order) that have data + functions inside them.	When the system is big and has many related entities.	Banking app (Customer, Account, Loan), ERP systems.
Functional Paradigm	You use small **functions that don't change data**. No side effects. Best **for parallel tasks**.	When processing a lot of data or doing analytics/ML.	Fraud detection using map/filter/reduce on transactions.
Event-Driven Paradigm	System **works based on events**. Something happens → **system reacts**.	When you need real-time, asynchronous workflows.	E-commerce order workflow (OrderPlaced → PaymentSuccess → ShipOrder).
Declarative Paradigm	**You tell *what you want*, not *how to do it***. System does the work.	For configuration, queries, and describing end-state.	SQL queries, Terraform, Kubernetes YAML files.
Reactive Paradigm	You build systems that **respond to data streams**. It's non-blocking and scales easily.	When you have massive user traffic and need high responsiveness.	Netflix's recommendation feed or live stock market price updates.

10

Paradigm Comparison … continue

Cognixia

Situation / Need	Best Paradigm	Why?
Quick **scripting**, **automation**, hardware-level tasks	**Procedural**	Keeps it simple; just a list of steps to execute.
Enterprise systems, maintainable large apps	**OOP**	Maps software to real-world entities (Users, Accounts).
Data analytics, ML, finance, concurrency	**Functional**	Perfect for heavy math and parallel processing.
Real-time notifications, UI, IoT devices	**Event-Driven**	Allows components to talk without being tightly glued together.
Configuration, cloud infrastructure, queries	**Declarative**	You define the "Goal" and let the system handle the "How."
High-concurrency, streaming data, resilient UI	**Reactive**	Handles thousands of updates per second without crashing.

11

Use Case | Hospital Management System

Cognixia

Component	Best Paradigm	Why?
Patient Vital Monitoring (live readings)	**Event-Driven**	**Triggers alerts** based on **heart rate**, **oxygen drop**
Doctor & **Patient Management** Portal	**OOP**	**Doctor**, **Patient**, **Appointment objects**
Disease Prediction & AI Analytics	**Functional**	Uses **pure functions**, data processing, ML
Database **Querying** (Patient Records)	**Declarative**	**SQL** for **querying data**
Device Controlling (Temperature Sensor, Alarm)	**Procedural**	**Step-by-step commands** to **control devices**
Real-time Heart Rate Stream (Live Waveform)	**Reactive**	**Handles** a **continuous flow of data points** smoothly without blocking the UI.

12

Cognixia

Model

www.cognixia.com

13

Cognixia

What is Model

- **Model** serves as an **abstract representation** of a **system**, a **problem**, or a **process**.
- A model is a **simplified blueprint** or a **working concept** used for analysis, design, communication, and prediction *before* the actual system is fully built.
- It **does not show every detail** but **captures important components**, **relationships**, *behavior, or structure*.
- **A Model is like a blueprint, diagram, or map** that helps architects and stakeholders understand how the system is designed.
- **Types of Models**
  - Models are often **categorized based** on what they are **trying to represent** or simplify.
    - Domain Models (Conceptual/Semantic Models)
    - Structural Models (Static Models)
    - Behavioral Models (Dynamic Models)
    - Process Models (Development/Lifecycle Models)

www.cognixia.com

14

Models | Types

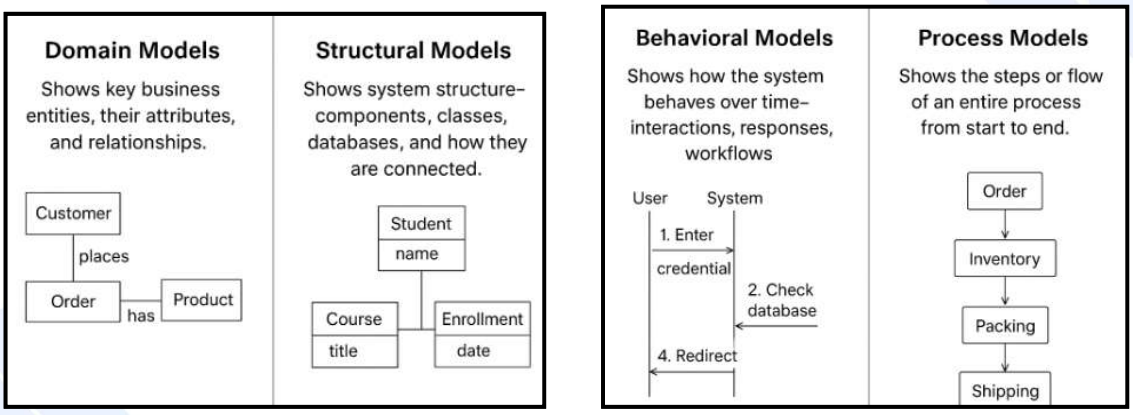
Cognixia

Model Type	Easy Explanation	When to Use	Real-World Example
Domain Models	Shows the **key business entities**, their **attributes**, and **relationships**. Focus = *what* exists in the business.	When **understanding the business domain** before designing the system.	E-commerce: Customer, Order, Product, Payment and how they relate.
Structural Models	Shows the **system structure—components**, **classes**, **databases**, and how they are connected. Focus = *system architecture*.	When **designing the system layout**, **database structure**, or class design.	Class diagrams for a hospital system (Patient, Doctor, Appointment).
Behavioral Models	Shows **how the system behaves over time**—interactions, responses, workflows. Focus = *actions and reactions*.	When **designing workflows**, **user interactions**, or system reactions.	Sequence: User places order → System validates → Payment → Confirmation.
Process Models	Shows the **steps or flow of an entire process** from start to end. Focus = *end-to-end workflow*.	When **documenting business processes**, **automation workflows**, or system processes.	Order-to-delivery process: Order → Inventory → Packing → Shipping.

15

Models | Types

Cognixia



www.cognixia.com

16

Cognixia

Architectural Style

www.cognixia.com

17

Cognixia

What is Architectural Style

- **Architectural Style** is a **high-level approach** *or* **way of structuring a system**.
- It tells you the **overall shape** and **interaction approach**, **not** the **full design**.
- **Scope**: The **entire system structure** and b**ehavior**. A **10,000-foot view**.
- **Purpose**: To define **system-wide principles** and **constraints**, helping to **achieve non-functional requirements** (*e.g., scalability, modifiability*).
- **Focus**: How **major components** are **structured** and **interact**.
- **Purpose of Architectural Style**
  - Provide a basic structure idea → Shapes system at high level (*Monolithic, Layered, Microservices*)
  - Help us communicate easily → "We are using Microservice style" – **gives quick clarity**
  - Sets direction for architecture → But doesn't give complete details

www.cognixia.com

18

Architectural Style Types

Cognixia

Style	Explanation
Layered (n-tier)	• The system is **structured in layers** like **Presentation**, **Business Logic**, and **Data**. • Each layer has a dedicated responsibility and communicates only with adjacent layers. • Common in traditional enterprise and web applications.
Client-Server	• The **server stores** and **manages data** or **services**, while **clients request** and **consume** them **remotely**. • This **model separates user interface** from **backend logic** and supports multiple clients like browsers, mobile apps, etc.
Microservices	• **Application** is **split into small**, **independent services**, each responsible for a specific business capability. • These **services communicate via APIs** or events and can be deployed, scaled, and maintained separately.
Event-Driven	• **Components communicate using events** instead of direct calls; one component produces an event and others react to it if interested. • Ideal for real-time, loosely coupled, scalable systems such as IoT or streaming platforms.
Peer-to-Peer (P2P)	• In this architecture where **every node (computer) acts as both a Client and a Server**. • There is **no central server** — all participants **share data** and resources directly **with each other**.

19

Cognixia

Design Pattern

20

What is Architectural/Design Pattern

Cognixia

- An Architectural Pattern **provides** a **reusable**, **proven solution** to a **recurring**, **complex problem** that **arises** when **implementing** an **architectural style**.
- **Scope**: **High-level component interactions**, but often more focused than the entire system.
- **Purpose**: To **solve** a **common architectural challenge** (*like managing distributed transactions or routing requests*) using **best practices**.
- **Focus**: Provides **a structured solution** for **implementing** a **style**.
- **Goal**: **Patterns help** you **write code** that is **more flexible**, **understandable**, and **easier** to **maintain**.
- The **pattern** is **generalized enough** to be **applied across many different projects** and contexts.
- **Patterns** are **not invented**; they are discovered and **documented** from **systems** that **already work well**.

www.cognixia.com

21

Patterns | Key Difference

Cognixia

Aspect	Architectural Pattern	Design Pattern
Level	**High-level** (System / Architecture)	**Low/Mid-level** (Class/Object / Code)
Purpose	To define the **fundamental structure** of the system, addressing system-wide non-functional requirements like **scalability, performance, and maintainability**.	To solve **specific, recurring design problems** during the implementation of a component.
Scope	**Entire system** or major subsystems	**Small part of the system** (one module or feature)
Stakeholders	**Solution Architect**, System Architect, Technical Lead	**Software Developer**, Programmer
Examples	**Layered**, **Microservices**, **Event-Driven**, **Client-Server**, **MVC**, **CQRS**	Singleton, Saga, CQRS
Testing Focus	**System-level testing** (integration, performance, scalability)	**Unit testing**, component testing
Abstraction	**High-Level.** Defines the overall structure, organization, and **interaction between *major components***.	**Low-Level.** Defines the detailed implementation and **interaction between *classes and objects***.
Impact	**Hard to Change.** A change requires a costly, system-wide overhaul.	**Easier to Change.** A change is localized to a few components or classes.

www.cognixia.com

22

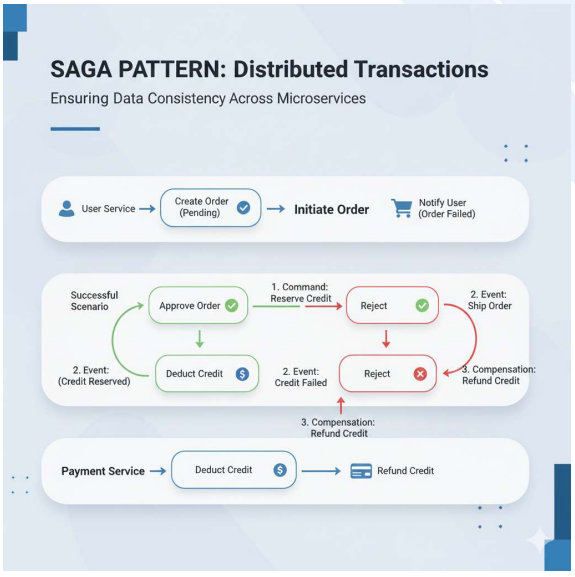
Architectural Pattern | Types

Architectural Pattern	Purpose	Common Use Case / Industry	Related Architectural Style
Layered (N-Tier)	Structure system into layers for separation of concerns	Banking apps, ERP, CRM, Insurance systems	**Layered Style**
MVC (Model-View-Controller)	Separates UI, logic, and data for flexibility and maintainability	Web apps, e-commerce sites, mobile apps	**Layered / Component-Based Style**
Client-Server	Centralized server providing resources to multiple clients	Email, web portal, SAP, authentication	**Client-Server Style**
Microservices	Break system into small services that can be developed, deployed, and scaled independently	Netflix, Amazon, FinTech, eCommerce	**Service-Oriented Style (SOA)**
Event-Driven	Communicate through asynchronous events for real-time response	IoT, stock trading, logistics tracking	**Event-Driven Style**
CQRS (Command Query Responsibility Segregation)	Separate read and write models for performance and scalability	High-traffic systems, banking, eCommerce	**Event-Driven Style**
Saga Pattern	Manage long-running or distributed transactions across microservices	Payment, travel booking, order workflows	**Microservices / Event-Driven Style**
Peer-to-Peer	All nodes are equal and share resources without central control	Blockchain, BitTorrent, Crypto apps	**Peer-to-Peer Style**
Serverless Pattern	Run small functions without managing servers (pay-per-use)	Chatbots, image processing, form submit	**Serverless Style**
API Gateway Pattern	Acts as single entry point for multiple services	Microservices, mobile apps, API management	**Service-Oriented Style**

23

Saga Pattern

- The **Saga Pattern** is a **design pattern used in microservices architecture** to handle **long-running or multi-step transactions** in a safe way.
- Unlike traditional database transactions, which are centralized and atomic, Sagas **break a transaction into multiple steps**, each handled by a different microservice.
- If any step fails, **compensating actions** are **triggered** to **undo the previous steps** to **maintain consistency**.



www.cognixia.com

24

Saga Pattern … continue

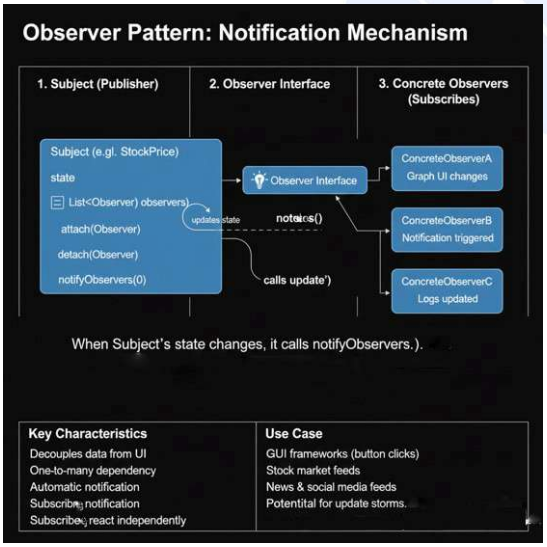
- **Online Order Process:**

Step	Service	Action	If Step Fails?
1	Order Service	Place Order	Nothing yet
2	Payment Service	Deduct Payment	Refund Payment
3	Inventory Service	Reserve Stock	Release Stock
4	Delivery Service	Schedule Delivery	Cancel Delivery

- **If** a **step fails** (*e.g., delivery cannot be scheduled*), **compensating transactions reverse** the **previous steps** (*payment refunded, stock released, order canceled*).
- **Example: E-commerce, Travel Booking, Insurance, Banking**

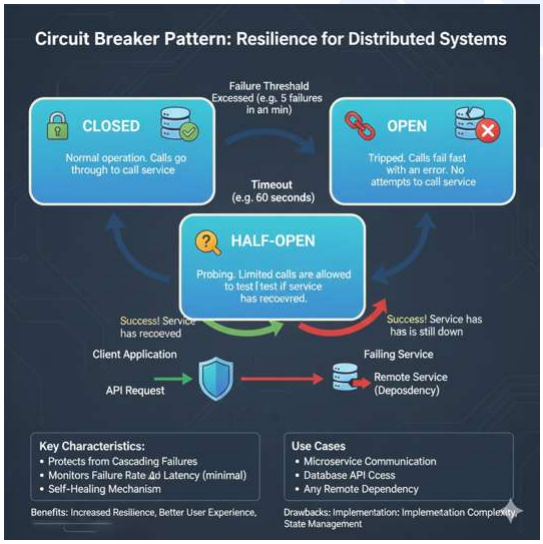
Observer Pattern

- Observer Pattern is a **design pattern** where **one object (*called Subject*)** maintains a list of other **objects** (***called Observers***), and **when something changes in the Subject, all Observers automatically get updated**.
- In simple words: **One changes → Many get notified automatically**.
- **Where is this used?**
  - This **pattern is used when**:
    - You have **one main source of data**,
    - And **multiple places need to react/know** when that **data changes**.
- **Example:**
  - **Stock Price Updates**:
    - **If** the **stock price changes, all observers** (*App, SMS, Emails*) automatically **receive updated prices**.
  - **YouTube Update**:
    - When a **YouTuber uploads** a **new video, all subscribers get a notification**.
    - **Subscribers don't need to check** daily — they just get **auto updates**.



Circuit-Breaker Pattern

- Circuit Breaker Pattern is used in **microservices and distributed systems** to **stop calling a failed or slow service** temporarily so that the **entire system does not crash or slow down**.
- Simple Meaning: **If a service keeps failing, stop calling it for some time and protect the system**.
- **Circuit Breaker States**
  - Closed: Everything is normal → Requests allowed
  - Open: Service is failing → Stop all requests
  - Half-Open: Service might be recovering → Allow limited test request



Circuit-Breaker Pattern … continue

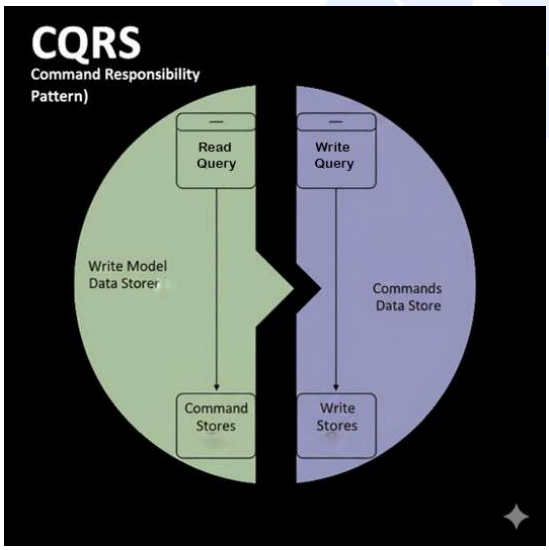
- **Example**: Payment API Failure
  - Imagine an e-commerce application calling the Payment Service.

Condition	What Circuit Breaker Does
Payment service is working	Calls allowed normally
Payment service is slow or failing repeatedly	Circuit Breaker trips – stops sending requests
Wait for a while	Circuit breaker waits (cooling period)
Service recovering	Send limited test requests
If successful, resume	Normal calls begin again

  - This prevents **system overload**, avoids **slow user experience**, and protects other services.

CQRS Pattern

- **CQRS** is an **architectural pattern** that separates the data **modification logic (Commands)** from the data **retrieval logic (Queries)**, often using different data models or databases.
- Uses an **optimized data structure for writing/integrity** and a **separate**, optimized structure **for reading/display**.
- **Components**
  - **Write Model/DB**: Optimized for writes and transactional integrity.
  - **Read Model/DB**: Optimized for reads and display performance.
- **Example** (**E-commerce**):
  - **Query Side handles millions** of **product searches** and **page views** (**Read-Heavy**).
  - The **Command Side handles** far fewer, but critical, actions like **PlaceOrderCommand** (**Write-Heavy**).

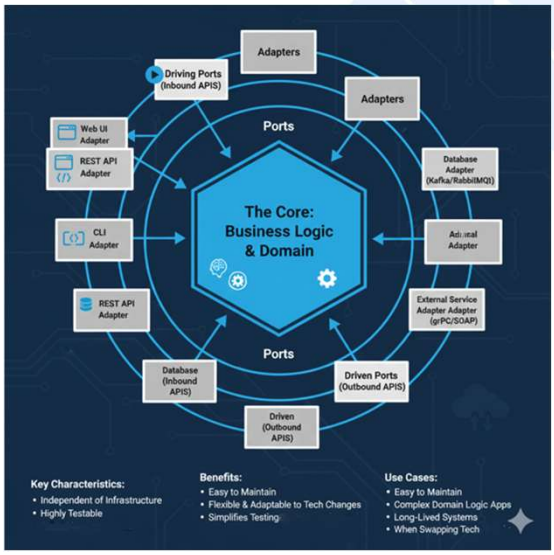


www.cognixia.com

29

Hexagonal Pattern

- Hexagonal Architecture pattern, also called Ports and Adapters Pattern, is a software design pattern that aims to isolate the core application logic from external systems.
  - The core business logic sits in the center of the architecture.
  - **Ports** define interfaces for communication with the outside world.
  - **Adapters** implement these ports to connect external systems like databases, UI, APIs, or messaging systems.
  - This allows the application to be loosely coupled, maintainable, and testable.



www.cognixia.com

30

Hexagonal Pattern … continue

Cognixia

- **When to Choose**
  - Applications with multiple external interfaces (Web, API, Messaging)
  - Projects where external dependencies may change frequently
- **Example (Restaurant)**

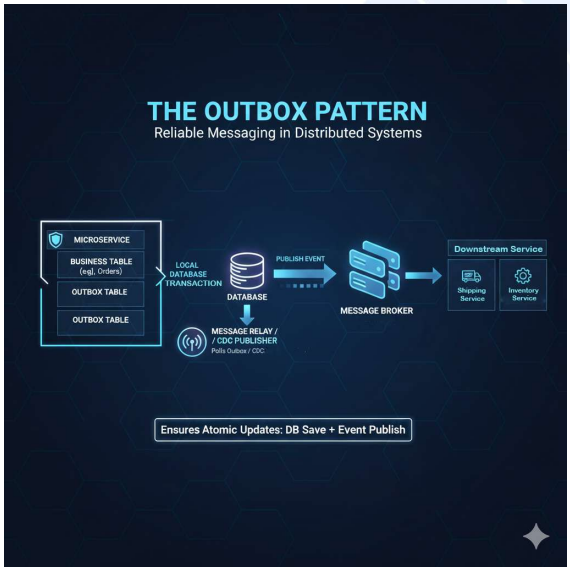
Layer	Meaning (Architecture)	Restaurant Example	Software Example
Core	**Main business logic**, independent of UI, database, or external systems	**Kitchen & Chef** – **actual food preparation** (recipe, cooking rules)	**Order processing**, **price calculation**, **stock check** (pure business logic)
Port	**Interface/Contract** – defines how external systems communicate with core	**Order Counter** / Kitchen Window – standard **way orders are accepted**	**OrderPort interface** – how core receives orders (not tied to any technology)
Adapter	**Real implementation** – connects external systems to the core	**Waiter**, **Phone Order**, **Swiggy**, **Zomato**, **WhatsApp** – different order methods	**REST API**, Database Adapter, Payment Adapter, Kafka Event Adapter

31

Outbox Pattern

Cognixia

- The **Outbox Pattern** is a **reliability pattern** used in microservices to ensure that a **database update** and a **message/event notification happen atomically**.
- In **distributed systems**, you often **need** to **save data** to your **database** (e.g., "Order Saved") and then **tell other services about** it (e.g., "Send Email").
- If the **database saves but** the **message broker fails**, your system becomes inconsistent.
-
- **The Problem: Dual Writes**
  - **When** a **service tries** to **write to two different systems** (*the Database and the Message Broker*) in **one go**, it faces the **"Dual Write" problem**:
    - **DB succeeds, Broker fails: The order** is **saved**, but the **customer never gets** an **email**.
    - **Broker succeeds, DB fails: The customer gets** an **email**, but the **order doesn't exist** in the **system**.



www.cognixia.com

32

Outbox Pattern … continue

- **The Solution: The Outbox Pattern**
  - **Instead** of **sending** the **message directly** to the **broker**, the **service saves** the **message** into a **special table called** the **Outbox** inside the **same database** as the **business data**.
  - **"Don't send the message immediately. Save it first".**
  - **How it works (Step-by-Step):**
    - **Local Transaction**
      - ✓ The **service starts** a **database transaction**.
      - ✓ It **updates** the **business table** (*e.g., Orders*) and inserts a record into the Outbox table.
    - **Commit**
      - ✓ Since **both happen** in the **same transaction**, they **either both succeed** or **both fail**.
    - **Message Relay**
      - ✓ A **separate process** (*called a **Message Relay** or **Publisher***) **polls** the **Outbox table**.
    - **Dispatch**
      - ✓ The **Relay sends** the **message** to the **Message Broker**.
    - **Cleanup**
      - ✓ **Once** the **Broker acknowledges** receipt, the **Relay marks** the **outbox message** as **"Sent"** or **deletes** it.

33

Outbox Pattern … continue

- **Example: E-commerce Checkout**
  - **Procedural Step**
    - A user clicks "Buy."
  - **Database Action:**
    - Table Orders: Insert New Order.
      - Table Outbox: Insert "OrderPlaced" event.
      - *Transaction Commits.*
  - **Reactive/Event-Driven Relay**
    - A background worker sees the new Outbox entry and pushes it to Kafka.
  - **Downstream**
    - The Shipping service and Email service react to the Kafka event.
- **Key Tools for Implementation**
  - **CDC (Change Data Capture)**
    - Instead of a "polling" worker, **tools** like **Debezium** can **watch** the **database transaction logs** and **automatically push Outbox** entries to **Kafka**.
  - **Idempotency**
    - Since the **Outbox pattern guarantees *at-least-once* delivery**, the **receiving service** must be **able** to **handle** the **same message twice** without errors.

34

Cognixia

Solution Architecture

www.cognixia.com

35

Cognixia

What is Solution Architecture

- System Architecture/ Solution Architecture is the high-level blueprint that defines how a software system is structured, how its components interact, and how it meets technical and business requirements.
- It answers questions like:
  - How should the system be organized?
  - Which components should exist and how do they communicate?
  - How do we ensure scalability, reliability, security, and performance?
- Just like a building architecture defines structure, materials, and design, software architecture defines structure, components, technologies, and interactions.
- **Key Elements**:
  - Components: Building blocks (services, modules, databases)
  - Connectors: Communication between components
  - Constraints: Rules, restrictions, standards

www.cognixia.com

36

Cognixia

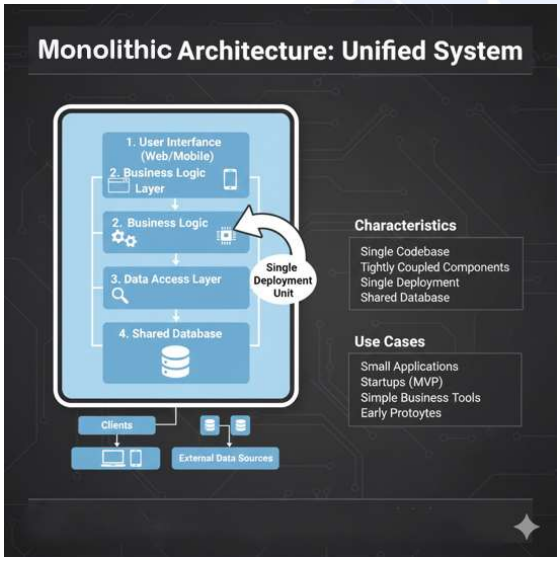
Monolithic Architecture

37

Cognixia

What is Monolithic Architecture

- Monolithic Architecture is a traditional way of building applications where all components *(UI, business logic, database access, APIs*) are built as a single, tightly-coupled unit.
  - All features are part of one big application
  - Runs as one deployable package (.jar, .war, .exe)
  - If one module fails → whole system may crash
  - Any change requires redeploying entire application
  - Good for small to medium applications but hard to scale as system grows



38

Monolithic Architecture | Core Elements

Cognixia

Element	Definition	Impact
UI Layer	Front-end or presentation layer (*Web pages, mobile screens*)	Handles user interactions; tightly integrated
Business Logic Layer	Core application logic (*order processing, calculations*)	All features share same codebase, strongly coupled
Data Access Layer	Functions to interact with database	Single shared database; centralized control
Database	Single relational database (*SQL*)	Easier consistency but can be a bottleneck
Connectors	Function calls or in-memory method calls	Very fast, but tightly coupled
Deployment Unit	Entire application packaged as one unit	Requires full redeployment on any change
Technology Constraint	Usually single tech stack (*Java/Spring, .NET*)	Easier to manage, but limits flexibility
Scalability Constraint	Can only scale as a full unit (*Vertical scaling*)	Expensive and less efficient

www.cognixia.com

39

Monolithic Architecture | When to Use

Cognixia

- **Small to Mid-size applications**
  - Easy to build and maintain
  - When the application has fewer modules (*like login, products, orders*), it's easy to build everything in one codebase.
  - Development, testing, and management are simpler because all components reside in a single place.

- **Startups or MVPs (Minimum Viable Product)**
  - Faster development and deployment
  - At startup stage, speed matters more than scalability.
  - Monolithic architecture allows rapid development, quicker changes, and faster go-to-market since everything is built as one application.

- **Strong coupling is acceptable**
  - Small team, single codebase
  - If one small team is working on the project and they all understand the entire system, having tightly connected modules is not a problem.
  - It reduces coordination overhead and makes collaboration easier.

www.cognixia.com

40

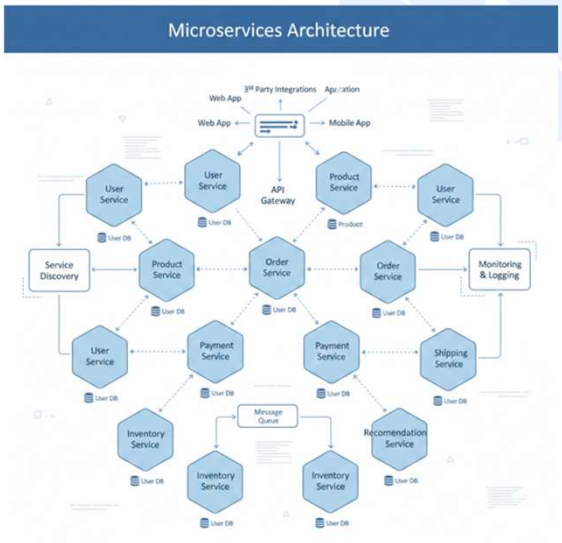
Microservices Architecture

41

What is Microservices Architecture

- Microservices Architecture is a design approach where an application is broken into small, independent, loosely coupled services, and each service performs one specific business function (*e.g., Order Service, Payment Service, Inventory Service*).

- Each service:
  - Has its own database
  - Runs independently
  - Can be developed, deployed, and scaled separately
  - Communicates via APIs or message queues



42

Microservices Architecture | Core Elements

Cognixia

Element	Definition	Impact
Service Components	Small, independent modules (Order, Payment, Inventory)	Loose coupling, easier scaling & development
API Gateway	Entry point for all client requests	Centralized access, routing, security
Databases (Polyglot Persistence)	Each service owns its own database	Avoids dependency, prevents data conflicts
Message Broker (Kafka, RabbitMQ)	Enables asynchronous communication	Helps event-driven actions, improves decoupling
Connectors	REST API, gRPC, or Message Queue	Enables communication between services
Deployment Unit	Each service is deployed independently (Docker, Kubernetes)	Enables CI/CD, rapid updates
Tech Stack Freedom	Each service can use different technology	Best-fit tech for each service
Design Constraint	Services must remain loosely coupled	Forces clear boundaries and autonomy

43

Microservices Architecture | When to Use

Cognixia

- **Large and complex applications**
  - Easier to manage by splitting services
- **Applications with high scaling needs**
  - Can scale only high-load services
- **Frequent updates and deployments**
  - Allows independent deployment
- **Distributed teams & multi-service ownership**
  - Teams can own different services
- **Cloud-native applications**
  - Designed for AWS, Azure, Kubernetes
- **Business domains are clearly separable**
  - e.g., Order, Payment, Catalog, Shipping

www.cognixia.com

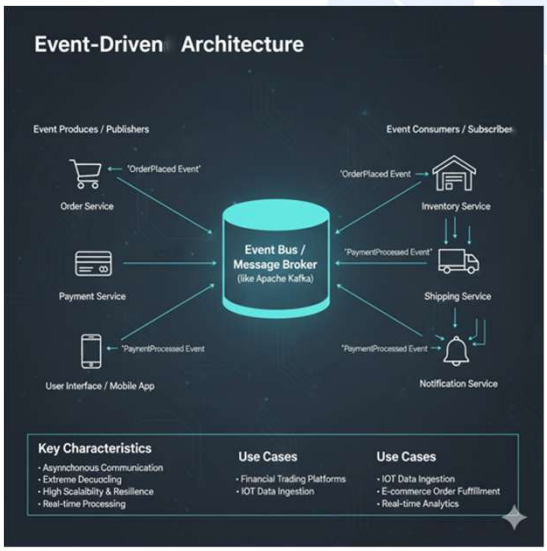
44

Event-Driven Architecture

45

What is Event-Driven Architecture (EDA)

- Event-Driven Architecture (EDA) is a system design approach where components communicate by producing and consuming events.
- What is an Event:
  - An event is something that happened — like Order Placed, Payment Failed, Sensor Triggered, Message Received, etc.
- Instead of making direct API calls, components publish events, and other components react to those events.
- In Event-Driven Architecture , when an event happens, it is published, and other services can listen and react to it — without depending directly on each other.
- System becomes Asynchronous, Real-Time, Highly Scalable, and Loosely Coupled.



46

How Event-Driven Architecture Works

- Event Flow
  1. Event Occurs (e.g., OrderPlaced in e-commerce)
  2. Event is Published to an Event Broker (Kafka, AWS SNS, RabbitMQ)
  3. Subscribers (Consumers) React to that event:
     - Payment Service → starts payment
     - Inventory Service → updates stock
     - Notification Service → sends email/WhatsApp
     - Shipping Service → creates shipping order
- Producer doesn't know who consumes the event — that's the power of decoupling.

47

Event-Driven Architecture | When to Use

- **Real-time processing needed**
  - **Use Case Examples**: Stock trading, ride matching (Uber), fraud detection, live sports updates
  - In real-time systems, responses need to be instant, based on live events. With EDA, events are immediately generated → consumed → action performed.
  - **Like**:
    - Stock Price Changed
      - ✓ Trading bot executes automatic buy/sell
    - Market Crash Detected
      - ✓ Alerts risk management; triggers stop-loss
    - Trade Executed
      - ✓ Updates portfolio, sends SMS/email to investor

48

Event-Driven Architecture | When to Use

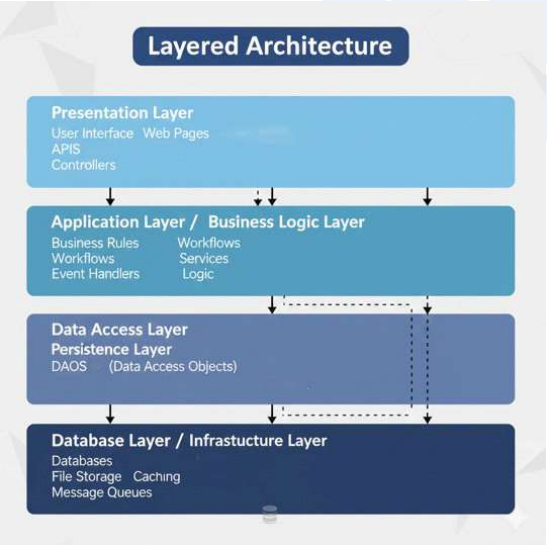
- **Multiple services react to one action**
  - **Use Case Examples**: Order fulfillment, ticket booking, hotel reservation
  - One action (*event*) in business triggers multiple reactions in parallel
  - **Like**:
    - Event: OrderPlaced [**Services That React → What Each Service Does**]
      - ✓ Payment Service → Validates card, initiates payment, confirms success/failure
      - ✓ Inventory Service → Checks available stock and reserves or deducts quantity
      - ✓ Shipping Service → Creates shipping request, generates tracking ID
      - ✓ Notification Service → Sends order confirmation emails/SMS/push notifications
      - ✓ Loyalty Service → Calculates and adds reward points to customer account

Layered Architecture

What is Layered Architecture

Cognixia

- The layered architecture organizes the application into horizontal layers, each responsible for a specific, distinct technical role.
- This arrangement ensures that components within a layer interact only with components in the layer immediately below it.
- The classic model is the
  - 3-Tier (or 3-Layer) Architecture, consisting of:
    - Presentation Layer (Top)
    - Business Logic Layer (Middle)
    - Data Access Layer (Bottom)
  - 4-Tier (or 4-Layer) Architecture, consisting of:
    - Presentation Layer (UI, screens, web pages)
    - Business Layer (Business logic, rules, validations)
    - Data Access Layer (Communication with database)
    - Database Layer (Actual data storage)



51

Layered Architecture | Layers

Cognixia

- While 3-Tier is common, modern systems often use four primary layers:
  - **Presentation Layer (UI Layer)**
    - Responsibility: Handling user interaction and displaying information.
    - Components: User Interface screens, HTML templates, Controllers (in MVC), or dedicated frontend applications (like React/Angular).
    - Role: Translates user actions into service calls and formats the results received from the Business Layer for display.
    - Example: **E-commerce Order Placement** - Receives the user's "Place Order" click and passes the request data to the layer directly below it. It cannot check inventory rules.
  - **Business (or Application) Layer**
    - Responsibility: Executing the core business rules, workflows, and transactional logic.
    - Components: Services, Managers, or Domain Models that encapsulate the "how-to" of the business.
    - Role: Coordinates between the Presentation and Persistence layers, ensuring that rules (*e.g., "A customer cannot order more than 10 units")* are enforced.
    - Example: **E-commerce Order Placement** - Receives the order data, executes the core logic: Checks inventory (rule), processes payment (external call), and then calls the next layer down to save the final state.

52

Layered Architecture | Layers

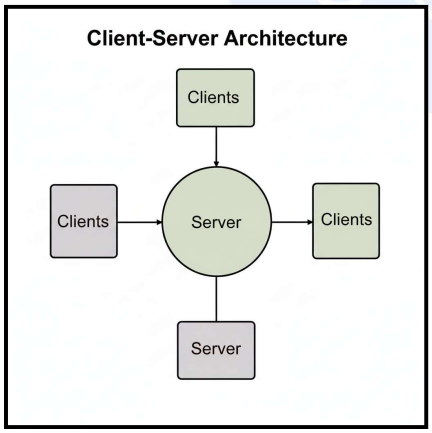
- **Persistence (Data Access) Layer**
  - Responsibility: Providing access to data from the data store.
  - Components: Repositories, Data Access Objects (DAOs) {*Perform standard database CRUD operations without requiring knowledge of the underlying SQL commands*}, or Object-Relational Mappers (ORMs) {*Automatically handles data translation by converting object operations into the appropriate SQL queries behind the scenes*}.
  - Role: Translates business layer data objects into database records (and vice versa) and handles connection management and SQL execution.
  - Example: **E-commerce Order Placement** - Receives the confirmed order object and translates it into a language the database understands (e.g., a SQL INSERT statement). It cannot execute business rules like checking the user's credit limit.

- **Database Layer (Infrastructure)**
  - Responsibility: Managing physical data storage and retrieval.
  - Components: The actual database server (e.g., PostgreSQL, MySQL, MongoDB).
  - Role: Ensures data integrity, durability, and simultaneous access control.
  - Example: **E-commerce Order Placement** - Physically writes the order record to the disk, guaranteeing durability and transaction safety.

Client-Server Architecture

Client-Server

- The **Client-Server** architecture distributes tasks between **service providers (Servers)** and **service requesters (Clients)**.
- The **Client initiates** the **request**, and the **Server manages** the **resources** and **business logic**. The goal is to **separate** the **user interaction layer** (Client) from the **data management** and **processing layer** (Server).
- **Components**
  - **Client**
    - **Requests services from** the **Server**.
    - **Initiates communication**.
      - ✓ **Like**: Your Chrome or Safari **web browser** that **sends HTTP requests**.
  - **Server**
    - **Provides services** and manages **shared resources**.
    - **Listens for requests**.
      - ✓ **Like**: An Amazon Web Services (AWS) EC2 **instance running** an **application server**.
  - **Network**
    - The **medium connecting** the **Client and Server**.
      - **Like**: The **Interne**t or a **company's internal intranet**.
- **Example** (**Email Services**):
  - Your **browse**r (Client) **sends** a **request** to the **email server** (Server).
  - The **Server stores all your emails**, **manage**s the **sending/receiving protocols** (SMTP/POP3), and handles **spam filtering**, reducing the client's burden.



www.cognixia.com

55

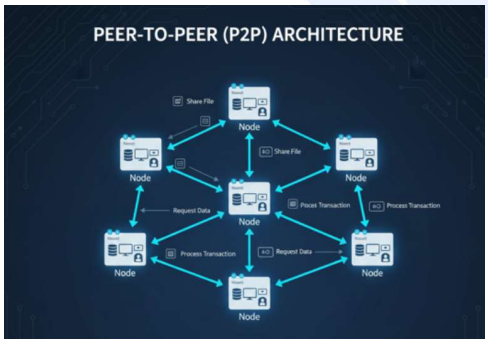
Peer-to-Peer Architecture

www.cognixia.com

56

Peer-to-Peer (P2P)

- The **Peer-to-Peer (P2P)** style is a distributed architecture where all participants (peers) are equal, acting as both resource requesters and providers.
- **Core Concept: Decentralization and Equality**
  - No single node is a dedicated server; all nodes share the workload and data management.
- **Components**
  - **Peer** (*Communicates directly with other peers to exchange data*)
    - An **individual node** (computer/device) that **acts a**s both a **consumer** and **provider**.
  - **Network Protocol** (*Facilitates resource discovery and transfer*)
    - Connectors that **enable direct**, bilateral **communication** between any **two peers**.
- **Example** (**Bitcoin**)
  - To **function** without a **central bank**, the **ledger (blockchain)** must be **copied**, **validated**, and **maintained** by **thousands of independent nodes** (Peers).
  - This **prevents any single entity** from **censoring transactions** or **altering history.**



www.cognixia.com

57

Microkernel Architecture

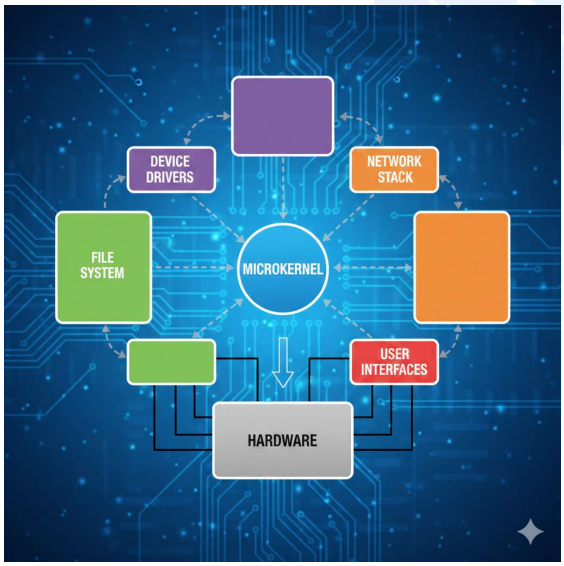
www.cognixia.com

58

What is Microkernel Architecture

Cognixia

- The **Microkernel Architecture** (also called **Plug-in Architecture**) is a design style where:
- **The core system does very little, and all features are added as plug-ins.**
- **Main Components**
  - **Core System**
    - **Minimal logic required** to **make the system run**.
    - It **handles security**, **resource management**, and the **plugin registry**.
    - Like Motherboard of a PC.
  - **Plug-in Modules**
    - **Independent components** that **add specialized features** or **business logic**.
    - Like Graphics Card or RAM, you plug in.



www.cognixia.com

59

What is Microkernel Architecture

Cognixia

- **Key Characteristics**
  - **Separation of Concerns**
    - The **core doesn't know what** the **plugins do**; it **only knows how** to **talk** to **them**.
  - **Scalability**
    - You can **add new features** just by **writing** a **new plugin**, without touching the existing code.
  - **Isolation**
    - If **one plugin crashes**, a well-designed **core can keep the rest** of the **system running**.
- **Example: E-Commerce System**

Component	Role in E-commerce	Why it's a Plugin?
The Core	**Manages the Shopping Cart**, **User Auth**, and the **"Order Total" calculation**.	This **logic never changes**, **regardless** of **how** the **user pays**.
Payment Plugin (Stripe)	**Processes Credit Cards**.	If **Stripe goes down**, you **can swap** it for a **Braintree plugin** instantly.
Tax Plugin (Avalara)	**Calculates VAT or Sales Tax** based on location.	**Tax laws change** constantly; you **only update this plugin**, not the core.
Shipping Plugin (FedEx)	**Gets real-time shipping rates** and prints labels.	**Different regions** use **different carriers** (*DHL in Europe, BlueDart in India*).

60

Cognixia

Serverless Architecture

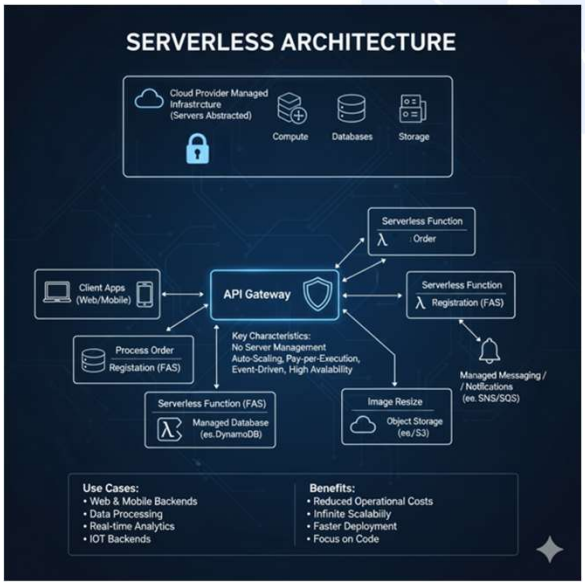
www.cognixia.com

61

Cognixia

What is Serverless Architecture

- Serverless Architecture is an architectural where cloud providers (*like AWS, Azure, or Google Cloud*) manage the entire infrastructure required to run an application.
- Despite its name, servers are still used, but the developer is abstracted away from the operational concerns of managing, provisioning, or scaling those servers.
- Serverless computing means the cloud provider dynamically manages the allocation of machine resources.
- The user pays only for the time their code is actually running, down to the millisecond. When the code isn't executing, there are no charges for idle time.
- Developers shift their focus entirely from infrastructure (OS, patching, capacity planning) to code and business logic.



62

63