# SERVICE ORIENTED ARCHITECTURE

4th Year — Software Engineering

LECTURE 01

01

# Introduction to SOA

Concepts, motivation, and the evolution from monolithic systems to distributed, service-based architectures.

# Agenda

1. What is software architecture, and why it matters
2. The evolution: monolithic to distributed systems
3. The integration problem in enterprise IT
4. What is Service-Oriented Architecture (SOA)?
5. Core roles: provider, consumer, registry
6. Benefits, challenges, and where SOA fits today

# What is Software Architecture?

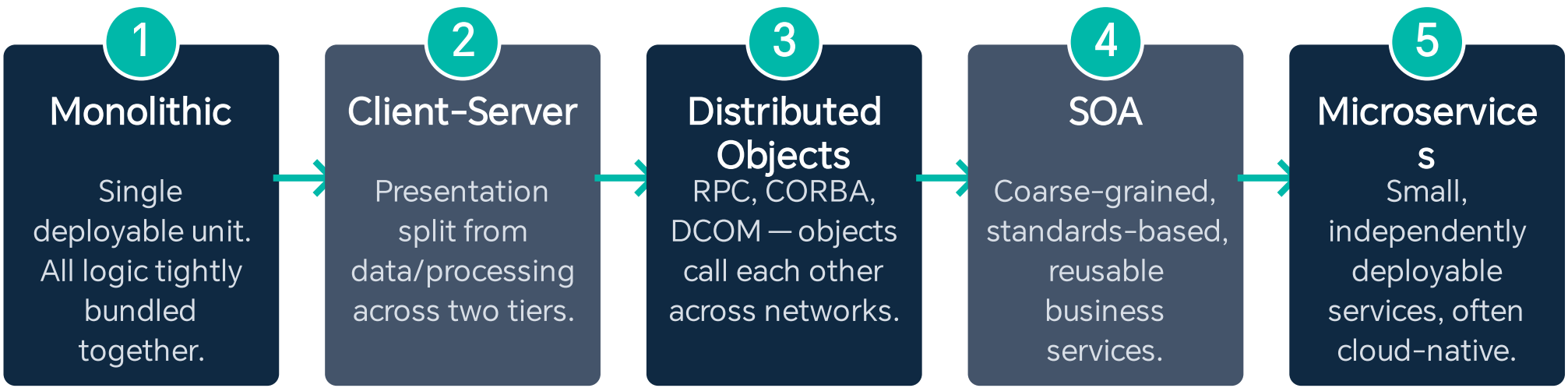
- The fundamental organization of a system, expressed through its components, their relationships, and the principles guiding its design and evolution.
- It defines how a system is decomposed into parts, and how those parts communicate
- Architecture decisions are the hardest to c[illegible] boundaries for everything built on top
- **Examples of architectural styles:**
  - Layered (n-tier) architecture
  - Client-server architecture
  - Event-driven architecture
  - Service-Oriented Architecture (SOA)
  - Microservices architecture

**Analogy**
Think of architecture as city planning, not building design: it decides where the roads (communication) go, not the interior of each building (implementation).

# Why Architecture Matters

- **Scalability**
  - Can the system handle 10x the users without a redesign?
- **Maintainability**
  - Can one team change a module without breaking another team's code?
- **Interoperability**
  - Can systems built in different languages, on different platforms, talk to each other?
- **Reusability**
  - Can existing functionality be reused instead of rebuilt?
- **Business agility**
  - Can the organization respond quickly to new market demands by recombining existing capabilities?

# The Evolution of Enterprise Architecture



DEFINITION

# Monolithic Architecture

*"An application built and deployed as a single, indivisible unit, where the user interface, business logic, and data access layer are combined into one codebase and one process."*

- All functionality — orders, payments, inventory, etc. — lives in one code base
- Deployed together: one build, one binary, one release cycle
- Was the dominant style for enterprise systems well into the 1990s

# Monolithic Architecture: Characteristics

- **Single codebase and single deployment unit**
- Internal modules communicate through simple in-process function or method calls
- Usually built with one technology stack (one language, one framework, one database)
- Development is straightforward at small scale: one team, one repository, simple debugging
- Testing is simpler initially since everything runs in one process
- Works well for small applications, prototypes, or systems with a small, s

**Example**

A university's first student-records system, running as one Java application with one database, is a classic monolith.

# Monolithic Architecture: Limitations

- **Scalability bottleneck**
  - The entire application must be scaled, even if only one function is under load
- **Slow, risky deployments**
  - A one-line change requires rebuilding and redeploying the whole system
- **Technology lock-in**
  - Hard to adopt a new language or framework for just one part of the system
- **Team coordination overhead**
  - Multiple teams editing the same codebase leads to merge conflicts and release bottlenecks
- **Poor fault isolation**
  - A bug or memory leak in one module can crash the entire application

# Client-Server Architecture

- Splits a system into two roles: clients that request services, and servers that provide them
- Introduced a clear separation between presentation and data/processing
- **Common variants:**
  - 2-tier: client talks directly to a database server
  - 3-tier: presentation, business logic, and data are separated into distinct tiers
- Improved on monoliths by allowing the server to be shared across many clients
- Still limited: tightly coupled protocols, and scaling still meant scaling whole tiers rather than individual capabilities

# The Rise of Distributed Computing

- As organizations grew, applications needed to span multiple machines, departments, and even companies
- **Remote Procedure Call (RPC)**
  - Let a program call a function on a remote machine as if it were local
- **CORBA (Common Object Request Broker Architecture)**
  - A vendor-neutral standard for distributed objects across languages and platforms
- **DCOM (Distributed Component Object Model)**
  - Microsoft's proprietary answer to distributed objects
- These technologies proved distributed computing was possible — but they were complex, tightly coupled, and rarely interoperable with each other

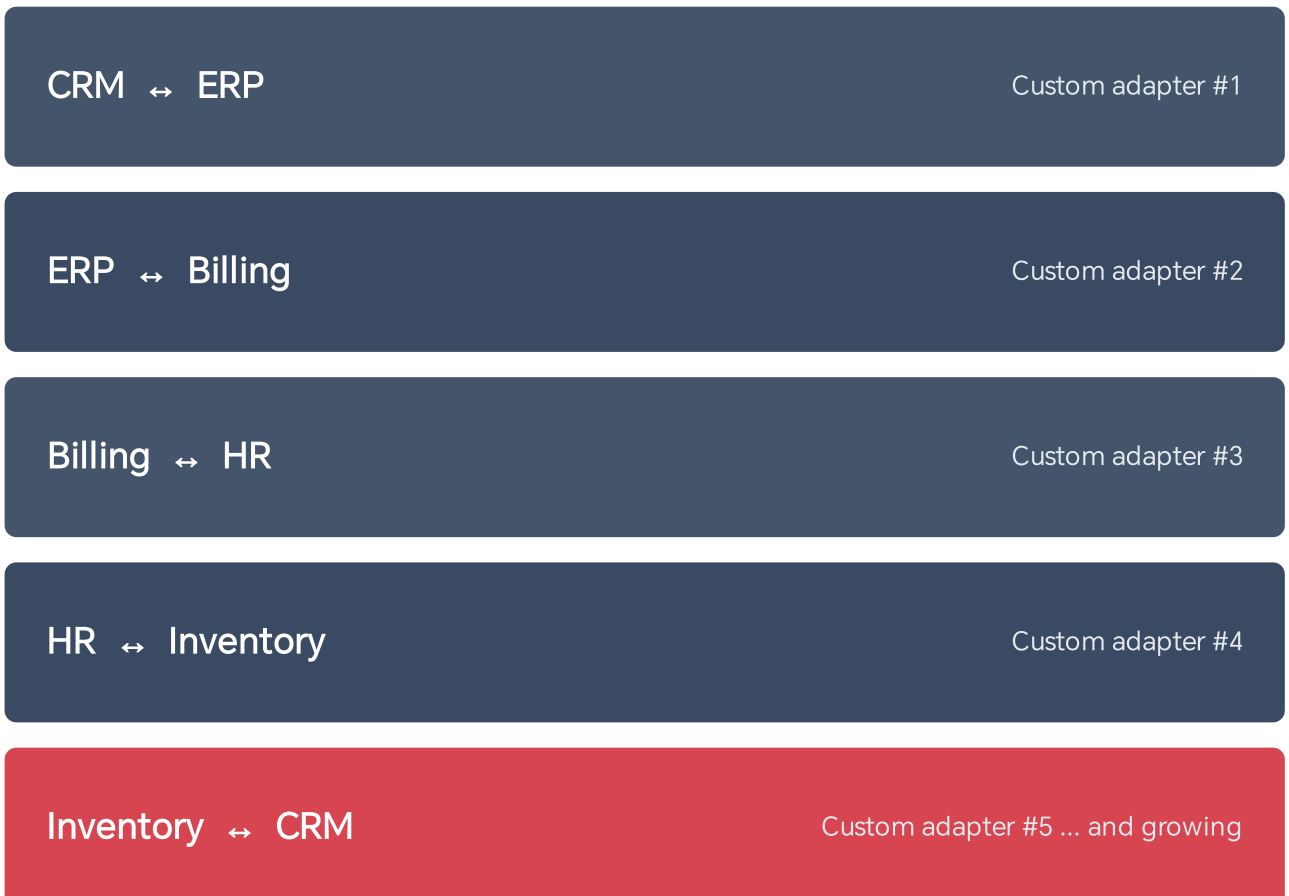
# Key Milestones in the History of SOA

| Year | Milestone |
|---|---|
| 1991 | OMG publishes CORBA 1.0 — first major distributed-object standard |
| 1996 | Microsoft releases DCOM as its distributed-object competitor |
| 1998 | XML 1.0 becomes a W3C Recommendation, enabling platform-neutral data exchange |
| 1999–2000 | SOAP and UDDI are introduced, standardizing service messaging and discovery |
| 2001 | WSDL 1.1 submitted to W3C, formalizing machine-readable service contracts |
| 2003–2005 | Gartner and industry analysts popularize the term 'SOA' for enterprise IT |
| 2010s | REST and lightweight APIs gain popularity; SOA ideas evolve into microservices |

# The Enterprise Integration Problem

- By the late 1990s, large enterprises had dozens of independent systems: CRM, ERP, HR, billing, inventory
- Each system needed data from the others — but each spoke a different protocol and data format
- Developers wrote custom, one-off integration code between every pair of systems that needed to talk
- **Result: an unmanageable web of point-to-point connections**
- Every new system added meant more custom connections — integration cost grew faster than the number of systems
- This tangled mess became known as 'spaghetti architecture'

# Point-to-Point Integration: The Spaghetti Problem




## The Core Problem

With N systems, point-to-point integration can require up to N×(N-1)/2 custom connections. Adding one new system can mean writing several new adapters. This does not scale — organizations needed a shared, standard way for systems to expose and consume functionality.

DEFINITION

# Service-Oriented Architecture (SOA)

*“An architectural style in which software capabilities are exposed as discrete, standards-based, reusable services that communicate over a network using well-defined interfaces, independent of the underlying platform or implementation.”*

- Shifts integration from custom point-to-point code to standardized service contracts
- A service can be consumed by any client that understands the standard interface, regardless of language or platform

# The Core Idea Behind SOA

- **Instead of custom connections between every pair of systems, build reusable services that expose business capabilities**
- A service is a self-contained unit of functionality, such as 'CheckCustomerCredit' or 'ProcessPayment'
- Any authorized consumer can invoke the service through its published interface — without knowing how it is implemented internally
- New consumers can reuse existing services instead of rebuilding the same logic
- SOA turns integration from an N×N custom-wiring problem into an N-connections-to-a-shared-contract problem

DEFINITION

# What is a 'Service'?

*“A self-contained, well-defined unit of business functionality that performs a specific task, is described by a formal contract, and can be discovered and invoked over a network.”*

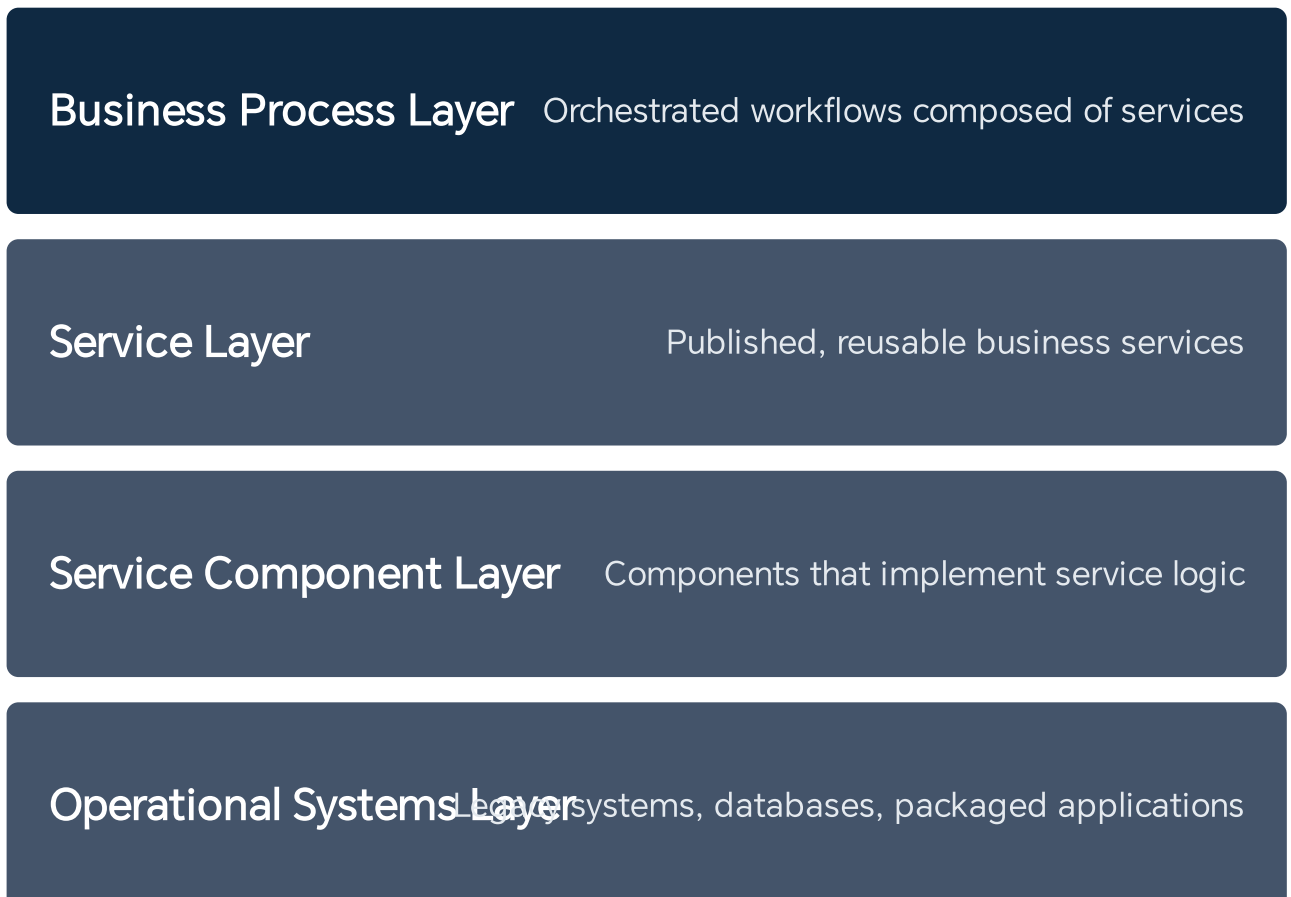
- Examples: 'ValidateAddress', 'AuthorizeCreditCard', 'CalculateShippingCost'
- A service exposes WHAT it does through its interface, and hides HOW it does it internally

# Service Characteristics — A First Look

- **Loosely coupled**
  - A service does not depend on the internal implementation of the services it talks to
- **Well-defined interface / contract**
  - Described formally, often in a language-neutral format such as WSDL
- **Self-contained and autonomous**
  - Controls its own logic and, ideally, its own data
- **Discoverable**
  - Can be found through a directory or registry
- **Reusable across multiple applications and business processes**

**Coming Next**
We will study each of these characteristics in depth — with formal definitions and examples — in Lecture 2.

# SOA Reference Architecture — Layered View




## How to Read This

Each layer depends only on the layer below it through a stable interface. Business processes are composed from services; services are implemented by components; components run on top of existing operational systems — including legacy applications that are wrapped, not rewritten.

# The Publish–Find–Bind Pattern

# Key SOA Roles

- **Service Provider**
  - Implements the service and publishes its contract for others to use
- **Service Consumer (Requester)**
  - Discovers and invokes the service to fulfil a business need
- **Service Registry / Broker**
  - A directory where providers publish service descriptions and consumers search for them (e.g., UDDI)
- These three roles interact through the publish–find–bind pattern shown on the previous slide

# Benefits of SOA

- **Reusability**
  - Write a service once, use it across many applications and processes
- **Interoperability**
  - Standards-based contracts let .NET, Java, and legacy mainframe systems interoperate
- **Business agility**
  - New processes can be assembled quickly by composing existing services
- **Reduced integration cost**
  - Fewer custom point-to-point connections over time
- **Legacy system protection**
  - Wrap old systems as services instead of replacing them outright

# Challenges and Criticisms of SOA

- **Governance overhead**
  - Requires strong discipline: contract versioning, ownership, standards enforcement
- **Performance cost**
  - Network calls and XML/SOAP processing add latency compared to in-process calls
- **Complexity of the ESB and middleware layer**
  - Introduces its own infrastructure to design, secure, and maintain
- **Service sprawl**
  - Without governance, organizations end up with many overlapping, poorly documented services
- **Steep learning curve and upfront investment**

# Monolithic vs. Service-Oriented Architecture

## Monolithic

- Single deployable unit
- In-process function calls
- One technology stack
- Scale the whole application
- Simple to build initially
- Hard to reuse logic elsewhere

## SOA

- Independent, distributed services
- Network calls via standard contracts
- Technology-agnostic, interoperable
- Scale individual services as needed
- Higher upfront design and governance cost
- Services reused across many

# Case Study: Airline Reservation Systems

- Airlines must integrate flight search, seat inventory, pricing, payment, and loyalty programs — often across partner airlines
- **Before SOA:**
  - Each partnership required custom, point-to-point integration code
- **With SOA:**
  - 'CheckFlightAvailability', 'PriceItinerary', and 'BookSeat' are exposed as standard services
  - Partner airlines and travel agencies consume the same published services, instead of custom integrations per partner
- This is a widely cited real-world example of why the travel and airline industry was an early adopter of SOA and web-service standards

# The SOA Standards Landscape (Preview)

| Standard | Purpose | Covered In |
|---|---|---|
| XML | Platform-neutral data format for messages | Lecture 3 |
| WSDL | Formal, machine-readable service contract | Lecture 3 |
| SOAP | Messaging protocol for invoking services | Lecture 4 |
| UDDI | Registry standard for publishing/finding services | Lecture 3 |
| REST / HTTP | Lightweight, resource-based service style | Lecture 5 |
| WS-Security | Security standard for SOAP messages | Lecture 10 |

# Where Does SOA Fit Today?

- Pure 'big SOA' with heavy ESBs and SOAP is less common in new, greenfield projects today
- **But its core ideas are everywhere:**
  - Microservices inherit loose coupling, service contracts, and independent deployability from SOA
  - Cloud API gateways play a similar role to the SOA registry and broker
  - Enterprise integration platforms (iPaaS) modernize the ESB concept
- Understanding SOA gives you the conceptual foundation to understand microservices, API management, and cloud-native architecture — the subject of Lecture 6

# Key Takeaways

1. Architecture evolved from monolithic, to client-server, to distributed objects, to SOA, to microservices
2. SOA emerged to solve the enterprise integration problem — the 'spaghetti' of point-to-point connections
3. A service is a self-contained, contract-based unit of business functionality
4. The publish-find-bind pattern connects service providers, consumers, and registries
5. SOA offers reuse, interoperability, and agility — at the cost of governance and performance overhead

DISCUSSION / REVIEW QUESTION

A university has separate systems for admissions, student records, and library management. The IT department wants to build a new mobile app that needs data from all three. How would a SOA approach differ from writing three separate point-to-point integrations?

## Think about:

- Consider what happens when a fourth system needs to be added later
- Think about who owns and maintains each integration point
- Consider how reuse would work if a second app needed the same data

# References & Further Reading


- Erl, T. — Service-Oriented Architecture: Concepts, Technology, and Design
- Josuttis, N. — SOA in Practice: The Art of Distributed System Design
- W3C — Web Services Architecture (www.w3.org/TR/ws-arch/)
- OASIS — Reference Model for Service Oriented Architecture


NEXT LECTURE

Lecture 2 — SOA Principles & Characteristics