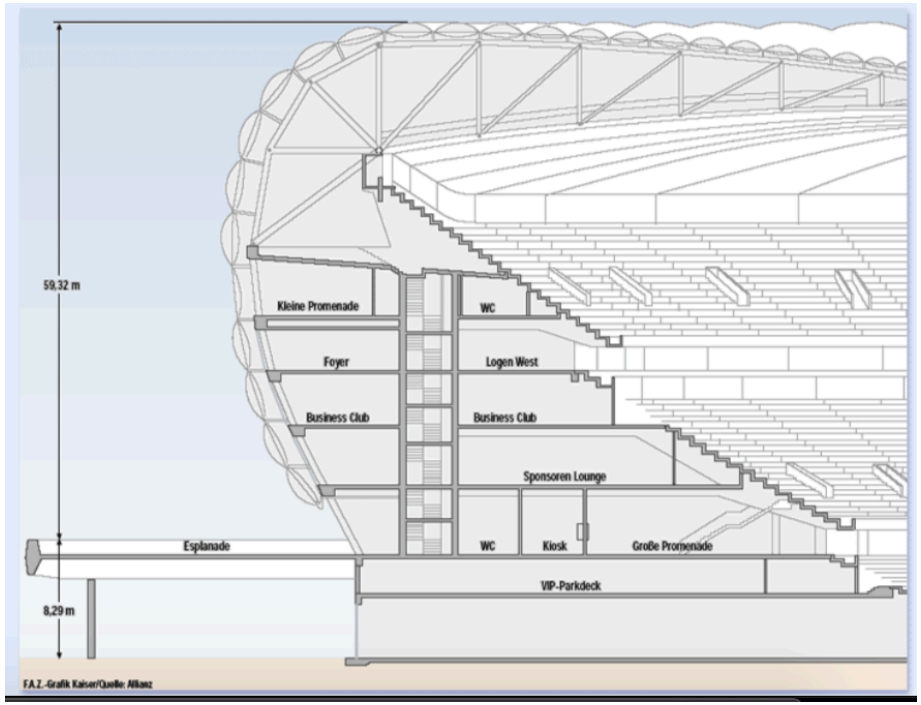
ARCHITECTURAL DESIGN

EGCO343 SOFTWARE DESIGN

KANAT POOLSAWASD
DEPARTMENT OF COMPUTER ENGINEERING
MAHIDOL UNIVERSITY

WHAT IS BUILDING DESIGN ?

- Münchner Allianz Arena, built for FC Bayern and TSV1860 (2002-2005)
- Inputs ?
- Constraints ?



WHAT IS SOFTWARE DESIGN?

- Requirements specification was about **WHAT** the system will do
- Design is about **HOW** the system will perform its functions

WHAT IS DESIGN ?

"What is design? What makes something a design problem? It's where you stand with a foot in two worlds – the world of technology and the world of people and human purposes – and you try to bring the two together. "

Mitchel Kapor, A Software Design Manifesto (1991)

KAPOR GOES ON TO SAY

"Design disciplines are concerned with ***making artifacts for human use.*** *Architects work in the medium of buildings, graphic designers work in paper and other print media, industrial designers on mass-produced manufactured goods, and software designers on software."*

The software designer should be the person with overall responsibility for the conception and realization of the program.

COST OF NOT PLANNING (1)

COST OF NOT PLANNING (2)

WHAT IS DESIGN IMPORTANT?

- Design allows a software engineer to model the system or product that is to be built.
- This model can be assessed for quality and improved before code is generated, tests are conducted, and end-users become involved in large numbers.
- Design is the place where **software quality is established**.

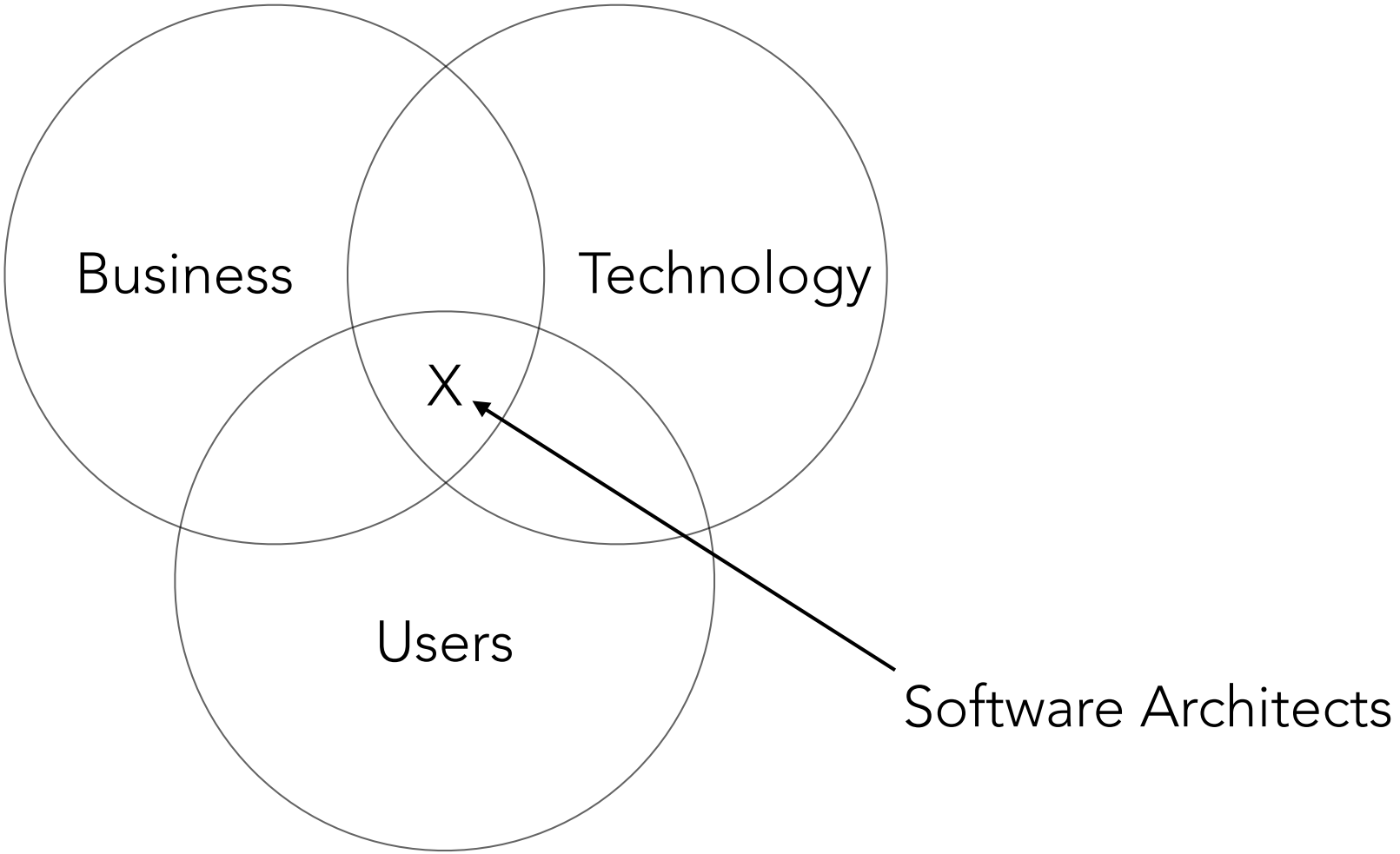
HOW TO APPROACH DESIGN ?

- Study and understand the problem from different viewpoints Identify potential solutions and evaluate the trade-offs (the first half of this course)
- Develop different **models of system** at different levels of abstraction: start global, subdivide (top-down), iterate (design is often a combination of top-down and bottom-up)

TWO COMMON PHASES OF SOFTWARE DESIGN

- Architectural design
  - Overall structure: main components and their connections; determining which sub-systems you need
- Detailed design
  - Inner structure of main components
  - Take programming language into account

WHAT SOFTWARE ARCHITECTS DO?



WHAT IS SOFTWARE ARCHITECTURE?

- Software architecture is the set of significant design decisions about how the software is organized to promote desired quality attributes and other properties.
- It is the **structure of the system** which consists of software components, the externally visible properties of those components and the relationship between them.

ARCHITECTURE VS DESIGN (1)

- **Software architecture is the high-level structure** of a system, focusing on defining which components exist and how they interact. It involves big-picture decisions that are difficult to change later.
- **Software design is the mid-to-low-level detail** within each component or module, focusing on how the components are implemented.

ARCHITECTURE VS DESIGN (2)

- **Software Architecture Concern:**
  - System organization (layers, modules, services, APIs).
  - Technology stack (databases, frameworks, protocols).
  - Deployment and scaling strategy (monolith vs. microservices, cloud-native vs. on-prem).
  - Cross-cutting concerns (security, performance, reliability, maintainability).
- Example
  - Decide that the system will use microservices, with services communicating via REST APIs, deployed on Kubernetes.
  - Split the system into modules: User Service, Order Service, Payment Service.

ARCHITECTURE VS DESIGN (3)

- **Software Design Concern:**
  - Algorithms and data structures.
  - Class and object design (OOP principles like SOLID, design patterns).
  - Interfaces and method signatures.
  - Error handling and validation logic.
- Example
  - Inside Payment Service: design classes like *PaymentProcessor, CreditCardValidator*.
  - Decide to use the Strategy Pattern for handling different payment methods.

ARCHITECTURE VS DESIGN (4)

Aspect	Architecture	Design
Level	High-Level	Mid-to-Low Level
Focus	Structure & Interaction	Implementation Details
Time	Early, hard to change later	Later, easier to refactor
Output	Architectural Diagrams, Deployment View	Class Diagrams, Sequence Diagrams, Pseudocode
Analogy	Blueprint of a building (rooms, floors, plumbing layout)	Interior design (furniture arrangement, decorations)

ARCHITECTURE STYLES

- In software design, there isn't a single fixed number of architecture types. Instead, there are families of architecture styles that have evolved over time.
- They can be grouped into two main levels:
  - System (Server) Architecture (Deployment and Communication)
  - Application Architecture (Internal structure of the system)
- Today, there are specialized or emerging architectures, such as data-centric architectures (e.g., data lakes) and edge computing architectures.

SYSTEM ARCHITECTURE

- These describe how applications/services interact and are deployed. Common types include:
  - Client-Server Architectures
  - Peer-to-Peer (P2P)
  - Service-Oriented Architecture (SOA)
  - Microservices Architecture
  - Event-Driven Architecture (EDA)
  - Etc.

APPLICATION ARCHITECTURE

- These describe how code and components inside an application are organized. Common types include:
  - Layered (N-Tier) Architecture
  - Pipes and Filters Pattern
  - Broker Pattern
  - Publish-Subscribe
  - Model–View–Controller (MVC)
  - Etc.

DISTRIBUTED SYSTEM

DISTRIBUTED SYSTEMS

- Virtually all large computer-based systems are now distributed systems
- Information processing is distributed over several computers rather than confined to a single machine
- Distributed software engineering is very important for enterprise computing systems

DISTRIBUTED SYSTEM CHARACTERISTICS

- Resource sharing
  - Sharing of hardware and software resources
- Openness
  - Use of equipment and software from different vendors
- Concurrency
  - Concurrent processing to enhance performance
- Scalability
  - Increased throughput by adding new resources
- Fault tolerance
  - The ability to continue in operation after a fault has occurred

DISTRIBUTED SYSTEM DISADVANTAGES

- Complexity
  - Distributed systems are more complex than centralized systems
- Security
  - More susceptible to external attack
- Manageability
  - More effort required for system management
- Unpredictability
  - Unpredictable responses depending on the system organization and network load

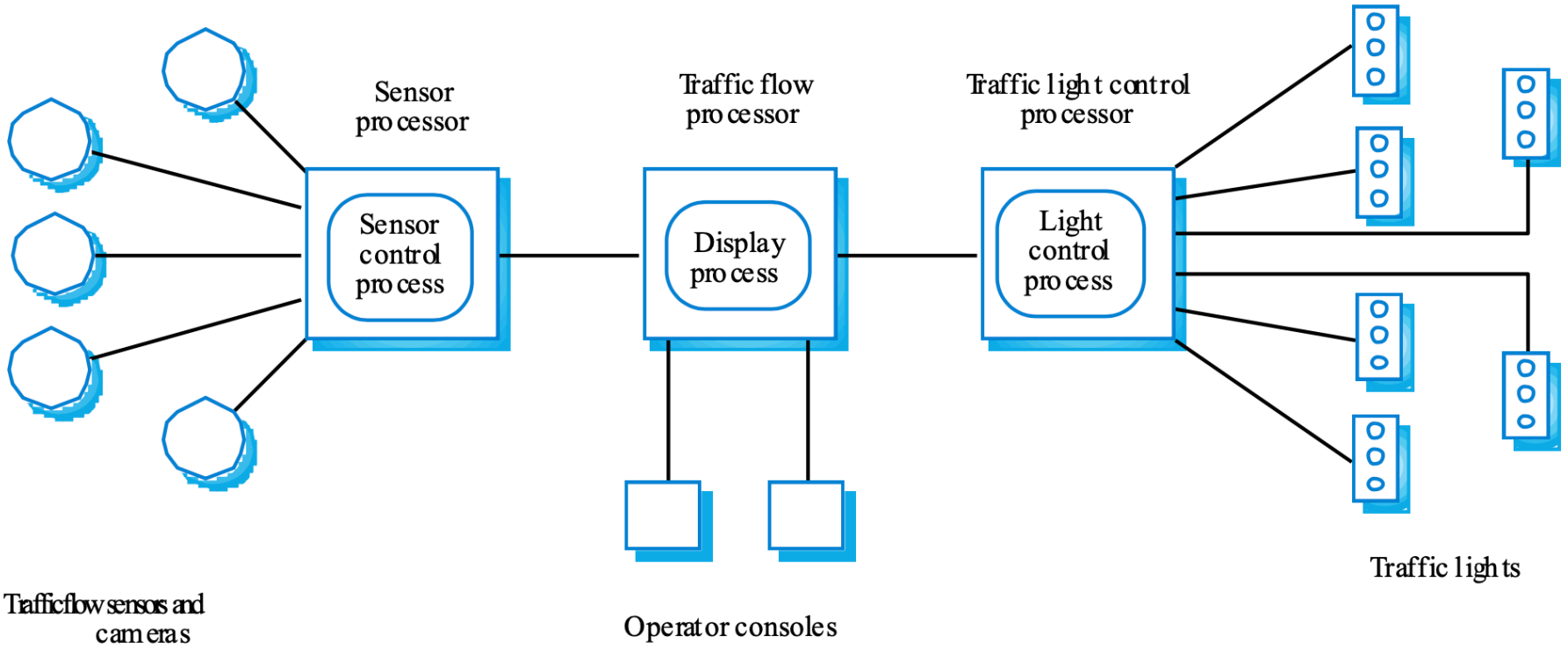
DISTRIBUTED SYSTEMS ARCHITECTURES

- Client-server architectures
  - Distributed services which are called on by clients
  - Servers that provide services are treated differently from clients that use services
- Distributed object architectures
  - No distinction between clients and servers
  - Any objects on the system may provide and use services from other objects

MULTIPROCESSOR ARCHITECTURES

- Simplest distributed system model
- System composed of multiple processes which can execute on different processors
- Architectural model of many large real-time systems
- Distribution of processor to processor may be pre-ordered or may be under the control of a dispatcher

MULTIPROCESSOR TRAFFIC CONTROL SYSTEM

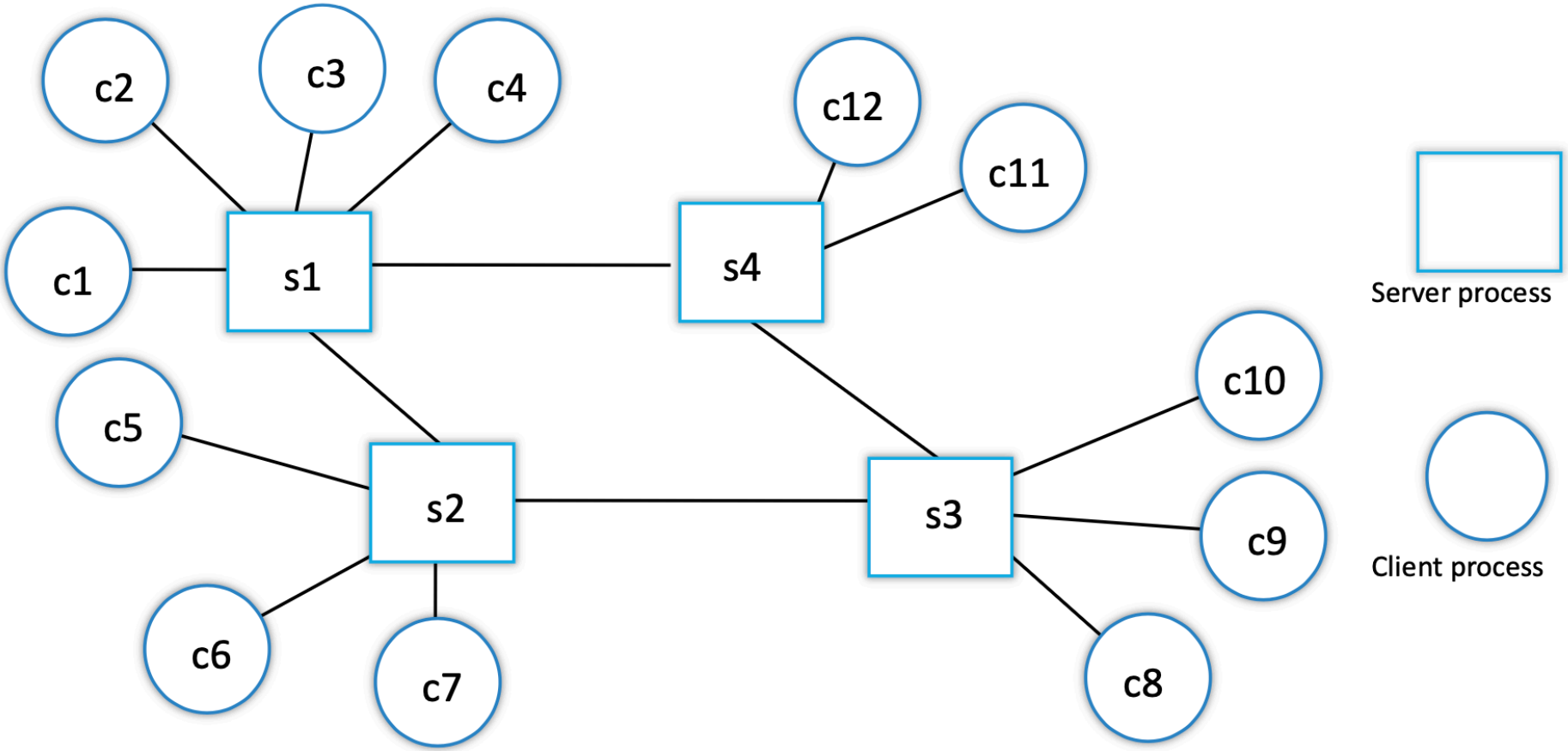


CLIENT/SERVER

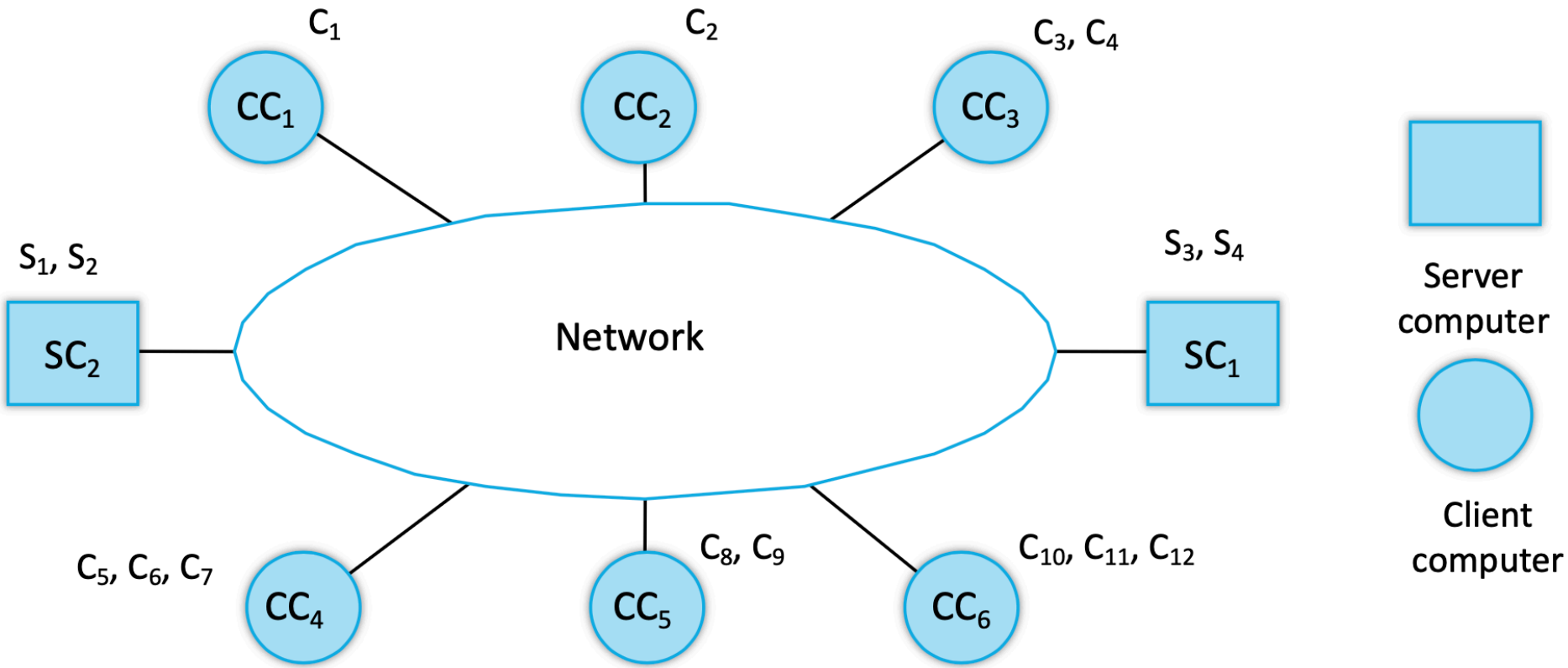
CLIENT-SERVER ARCHITECTURES

- Application is modeled as a set of services and provided by **servers and a set of clients** that use these services
- Clients know of servers but servers do not need to know of clients
- Mapping of processors to processes is not necessarily 1 : 1

CLIENT-SERVER SYSTEM



COMPUTERS IN A C/S NETWORK

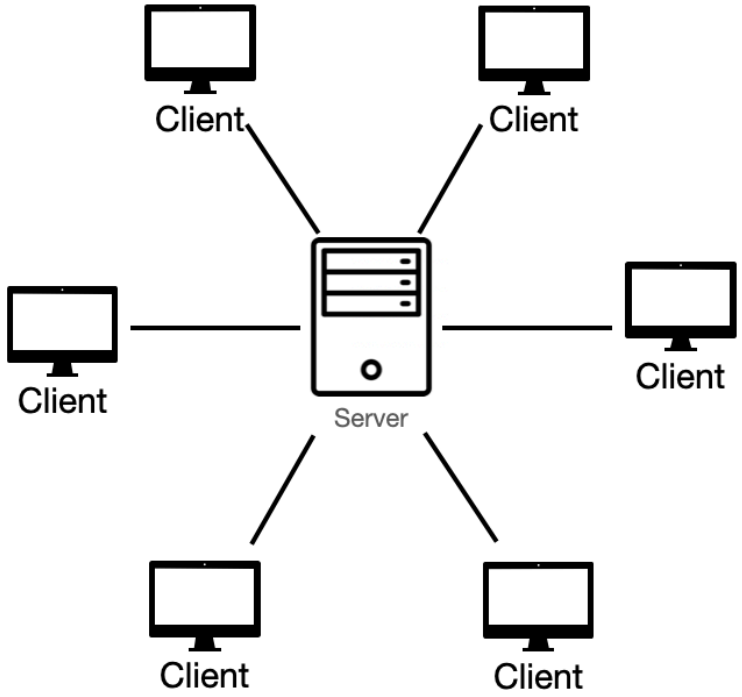


PEER-TO-PEER (P2P)

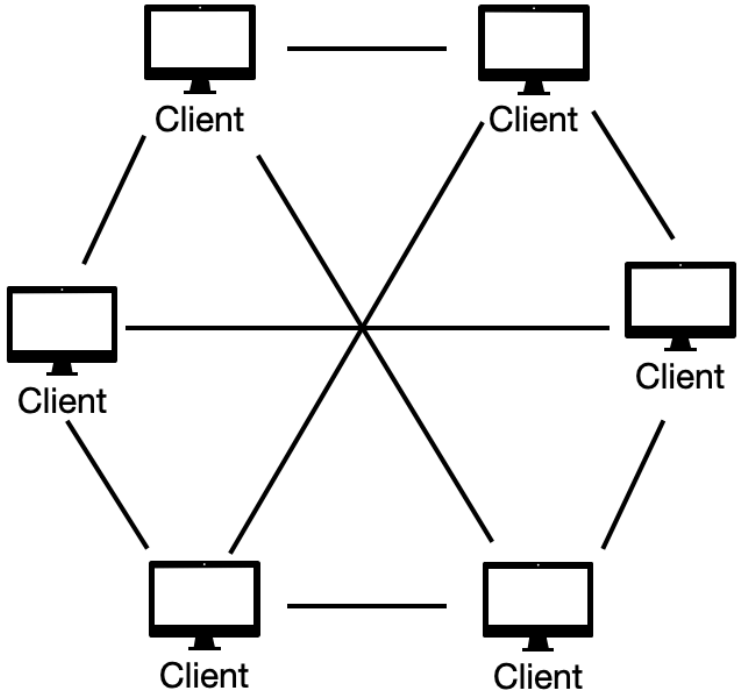
PEER-TO-PEER (P2P)

- Peer-to-Peer (P2P) Architecture is a network architecture where all nodes (peers) are equal.
  - Each peer can act as both a client (requesting resources) and a server (providing resources).
  - There is no central server controlling the system.
  - Communication is direct between peers.

P2P VS CLIENT/SERVER



Client Server Architecture



P2P Architecture

SERVICE-ORIENTED ARCHITECTURE

SOA CONCEPT

- Here are the key technical concepts of SOA that allow it to cope with the system characteristics just described:
  - Services
  - Interoperability
  - Loose Coupling

SERVICES

- A service is the IT realization of some self-contained business functionality. By focusing on the business aspects, a service hides technical details and allows business people to deal with it.
- Technically, **a service is an interface** for (multiple) messages that are **exchanged between provider(s) and consumer(s)**.
- The complete description of a service from a consumer's view point is called a "well-defined interface" or "contract".
- A contract is agreed individually between a certain provider, and a certain consumer, and usually **includes nonfunctional aspects such as SLAs**.

SERVICE LEVEL AGREEMENT

- A Service Level Agreement (SLA) is a formal contract or commitment that defines the expected level of service between a service provider and a customer or end user.
- Example of the SLA within NFR
  - Security - Data encrypted in transit and at rest (AES-256)
  - Performance - Average response time ≤ 2 seconds
  - Reliability - Mean Time Between Failures (MTBF) ≥ 1000 hours

HIGH INTEROPERABILITY

- With heterogeneous systems, the first goal is to be able to connect those systems easily. This is usually called high interoperability.
- High interoperability is not a new idea. Before SOA, we had the concept of “Enterprise Application Integration” (EAI), and reading interoperability, SOA is nothing new.
- However, for SOA, high interoperability is the beginning, not the end. It is the base from which we start to implement business functionality (services) that is spread over multiple distributed systems.

LOOSE COUPLING

- Loose coupling is the concept of **minimizing dependencies**. When dependencies are minimized, modifications have minimized effects, and the systems still runs when parts of it are broken or down. Minimizing dependencies contributes to fault tolerance and flexibility, which is exactly what we need.
- In addition, loose coupling leads to **scalability**. Large systems tend to challenge limits. Therefore, it is important to avoid bottlenecks; otherwise, growing might become very expensive.
- **Avoiding bottlenecks** is important from both a technical and an organizational point of view.

SOA INGREDIENTS

- The key technical concepts for SOA are services, interoperability, and loose coupling, you might conclude that all you have to do to enable SOA is to introduce services, interoperability, and loose coupling.
- A lack of ingredients is what the problem in a real SOA projects.
- To establish SOA successfully, you have to care for your **infrastructure, architecture, and processes.**

INFRASTRUCTURE

- Infrastructure is the technical part of SOA that **enables high interoperability**.
- The infrastructure of a SOA landscape is called an enterprise service bus (ESB). This term is taken from enterprise application integration, where it was called the EAI bus or just enterprise bus.
- The key feature of the ESB is that it enables you to **call services between heterogeneous systems**.
- Its responsibilities include data transformation, (intelligent) routing, dealing with security and reliability, service management, monitoring, and logging.

ARCHITECTURE

- Architecture is necessary to restrict all the possibilities of SOA in such a way that it leads to a working, maintainable system.
- SOA concepts, SOA tools, and SOA standards leave room for specific decisions that you must make in order to avoid a mess.
- You have to classify different types of services, you have to decide on the amount of loose coupling, you have to restrict the data model of service interfaces, you have to define policies, rules, and patterns, you have to clarify roles and responsibilities, you have to decide on the infrastructure technology, you have to decide which (version of) standards to use, and so on.

PROCESSES

- One thing that makes large systems complicated is that many different people and teams are involved in them.
- Because there is not typically only one person or a few people controlling everything, you will have to set up appropriate processes (these processes may be explicitly defined, or just implicitly evolve).
- These processes include:
  - Business process modeling (BPM)
  - Service lifecycles
  - Model-Driven Software Development (MDSD)

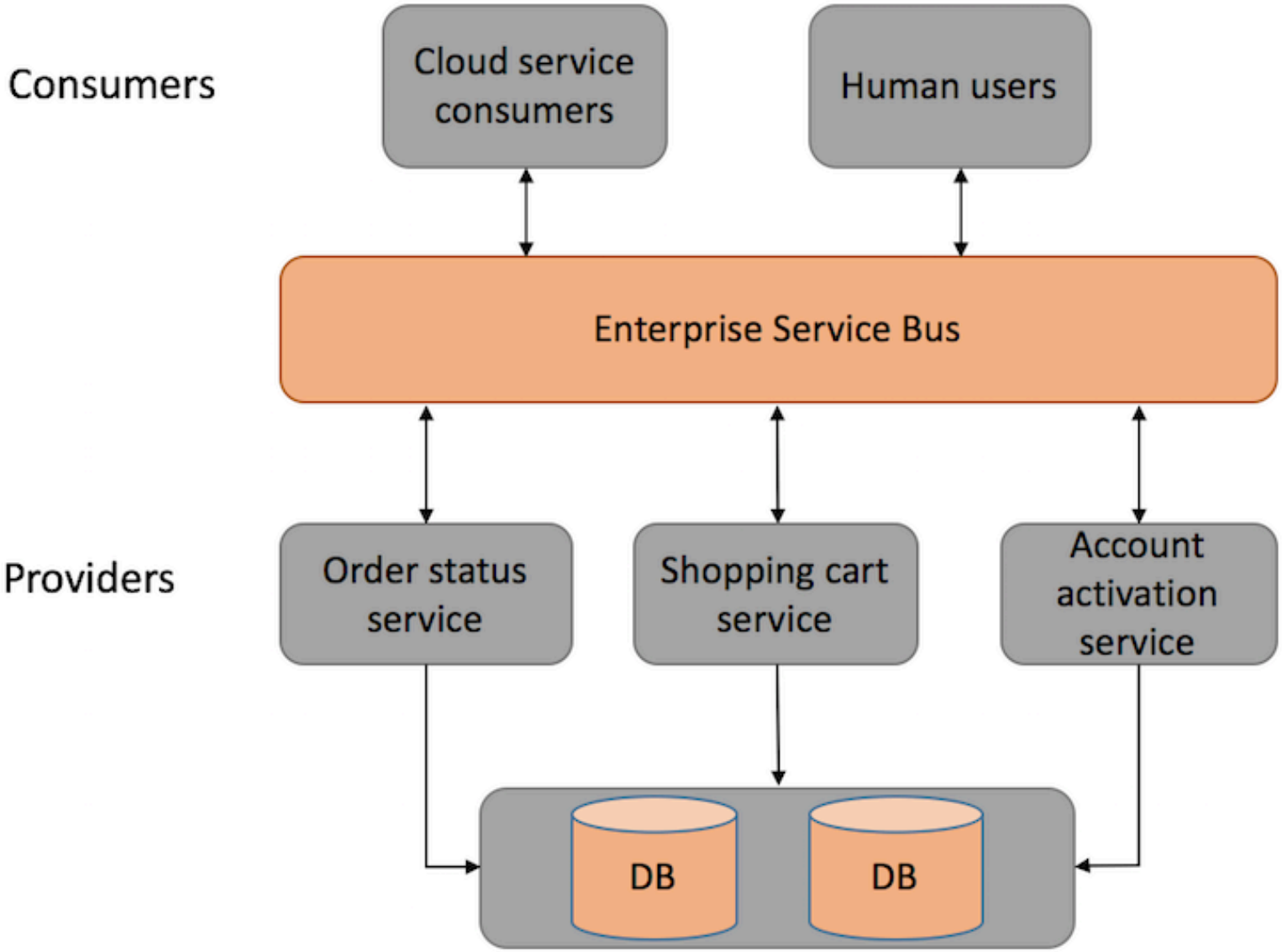
SOA IS NOT A SILVER BULLET

- SOA is the **ideal solution** for very special circumstances: heterogeneous distributed systems with different owners.
- There's a price to pay for dealing with heterogeneity and different owners, and providing flexibility, scalability, and fault tolerance.
- For this reason, if you don't have the type of system I've just described, think about not using SOA. If you have everything under control SOA might be pointlessly expensive for you.

SOA IS NOT A SPECIFIC TECHNOLOGY

- SOA is not the same as Web Services. **SOA is the paradigm**; Web Services are one possible way to realize the infrastructure by using a specific implementation strategy. This is an important distinction.
- This does not mean that building SOA with Web Services will solve all your problems. Web Services might help provide the infrastructure, but you will still have to construct the architecture, and set up all the complicated processes that are necessary for successful SOA.

SERVICE-ORIENTED ARCHITECTURE



SOA SUMMARY

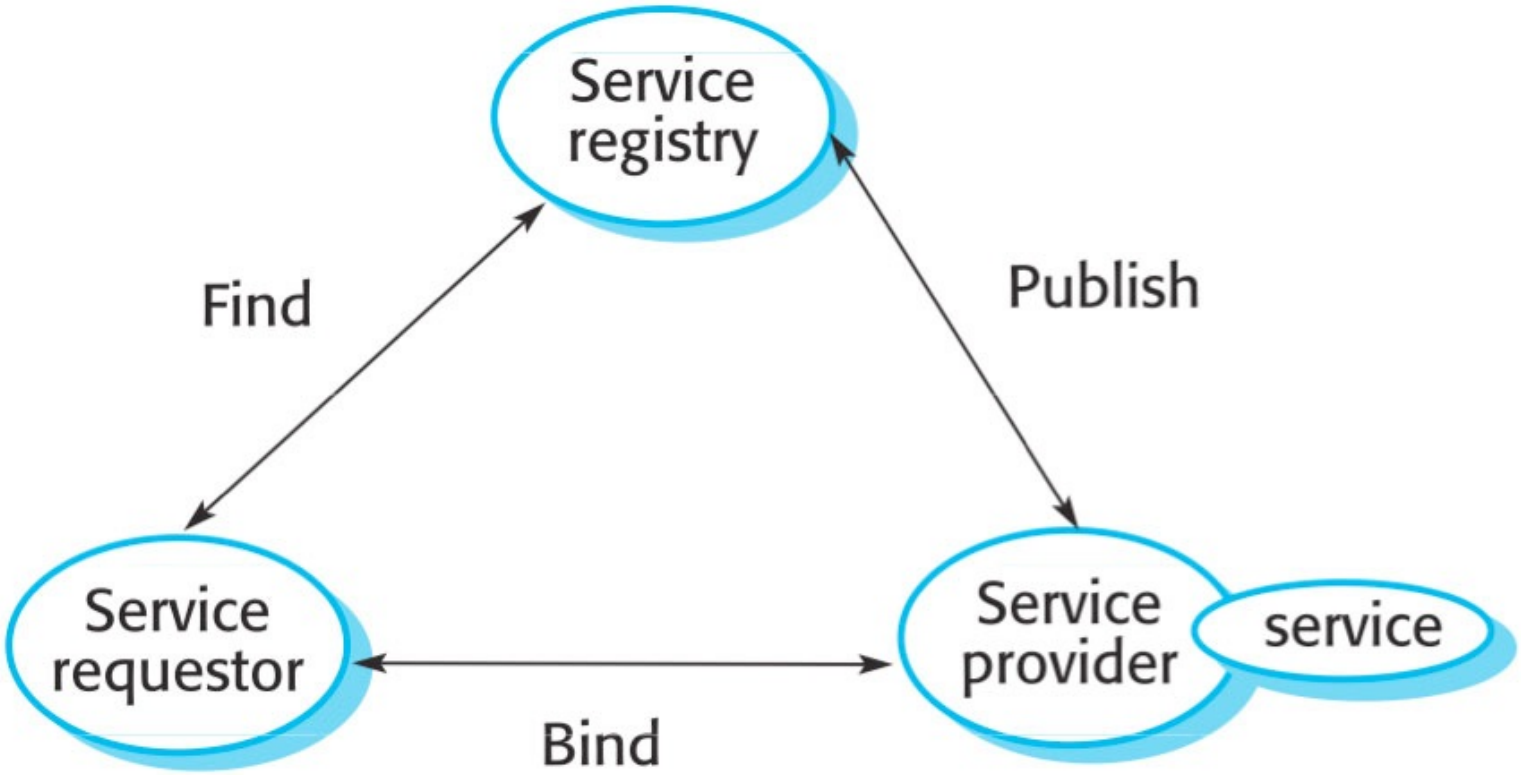
- SOA is an **architectural paradigm** for dealing with business processes distributed over a large landscape of existing and new heterogeneous systems that are under the control of different owners.
- The key technical concepts of SOA are s**ervices, interoperability, and loose coupling**.
- The key ingredients of SOA are **infrastructure, architecture, and processes** (including the meta-process of establishing SOA, governance).
- SOA is neither a specific technology nor a silver bullet.
- Web Services are one possible way of realizing the infrastructure aspects of SOA.

SERVICES

WHAT IS A SERVICE?

- Platform-independent computational entity that can be used in a platform-independent way
- Entities or application functionalities accessed via exchange of messages
- Software is “somewhere”, deployed on as-needed basis
  - SaaS: Software as a Service
- Often just used in connection with something else: SOA, Web services, etc.

WEB SERVICES



SERVICES ARE LOOSELY COUPLED

- The services are loosely coupled, meaning each **service operates independently** with minimal dependencies on others.
- This allows services to be modified, replaced, or scaled without affecting the rest of the system, enhancing flexibility, maintainability, and system resilience.

SERVICES ARE STATELESS

- The services are stateless, meaning **they do not retain client-specific data between requests**.
- Each request contains all necessary information, enabling easy scaling, fault tolerance, and independent processing by multiple service instances.

SERVICES ARE AUTONOMOUS, HIDE THEIR LOGIC

- The services are autonomous and hide their internal logic, meaning each service encapsulates its implementation details and exposes only a well-defined interface.
- This autonomy allows services to **operate independently**, minimizing dependencies between components, while hiding their internal processes ensures flexibility, maintainability, and the ability to change or replace the service **without impacting other parts of the system**.

SERVICES ARE REUSABLE

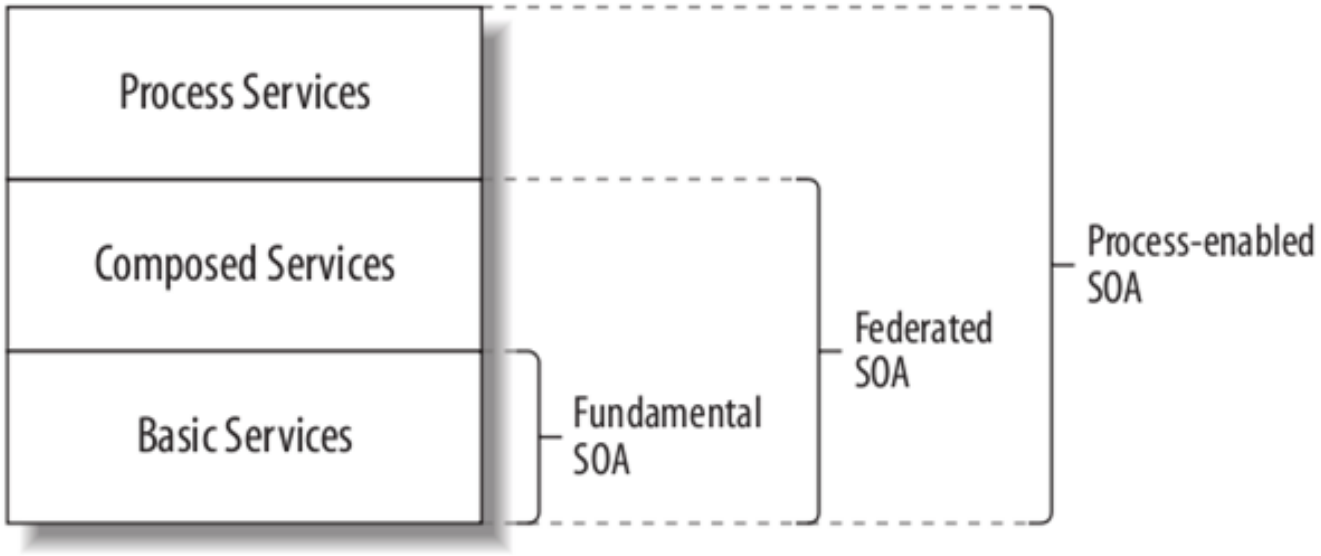
- The services are reusable, meaning a single **service can be leveraged across multiple applications or processes without modification**.
- This promotes efficiency, reduces duplication, and accelerates development by allowing consistent functionality to be shared throughout the system.

SERVICE USE OPEN STANDARDS

- The services use open standards, meaning they communicate and exchange data using widely accepted protocols and formats (e.g., HTTP, JSON, XML).
- This ensures **interoperability, compatibility across platforms**, and easier integration with other systems.

A FUNDAMENTAL SERVICE CLASSIFICATION (1)

- The starting point for service classification introduces the following three categories:
  - Basic Services
  - Composed Services
  - Process Services



BASIC SERVICES

- The first stage of expansion provides only basic services, which are services that each provide a basic business functionality and that it does not make sense to split into multiple services.
- Usually, these services provide the first fundamental business layer for one specific backend or problem domain. Typically, these services are short-term running and conceptually stateless. Thus, synchronous calls may be useful.
- There are two types of basic services:
  - Basic Data Services
  - Basic Logic Services

BASIC DATA SERVICES

- Basic data services read or write data from or to one backend system. These services typically each represent a fundamental business operation of the backend.
- Typical examples (in this case, for customer services) are services that:
  - Create a new customer. (CREATE)
  - Return the address of a customer. (READ)
  - Change the address of a customer. (UPDATE)
  - Cancel the ordered item. (DELETE)
- In fact, basic services should have the so-called ACID properties

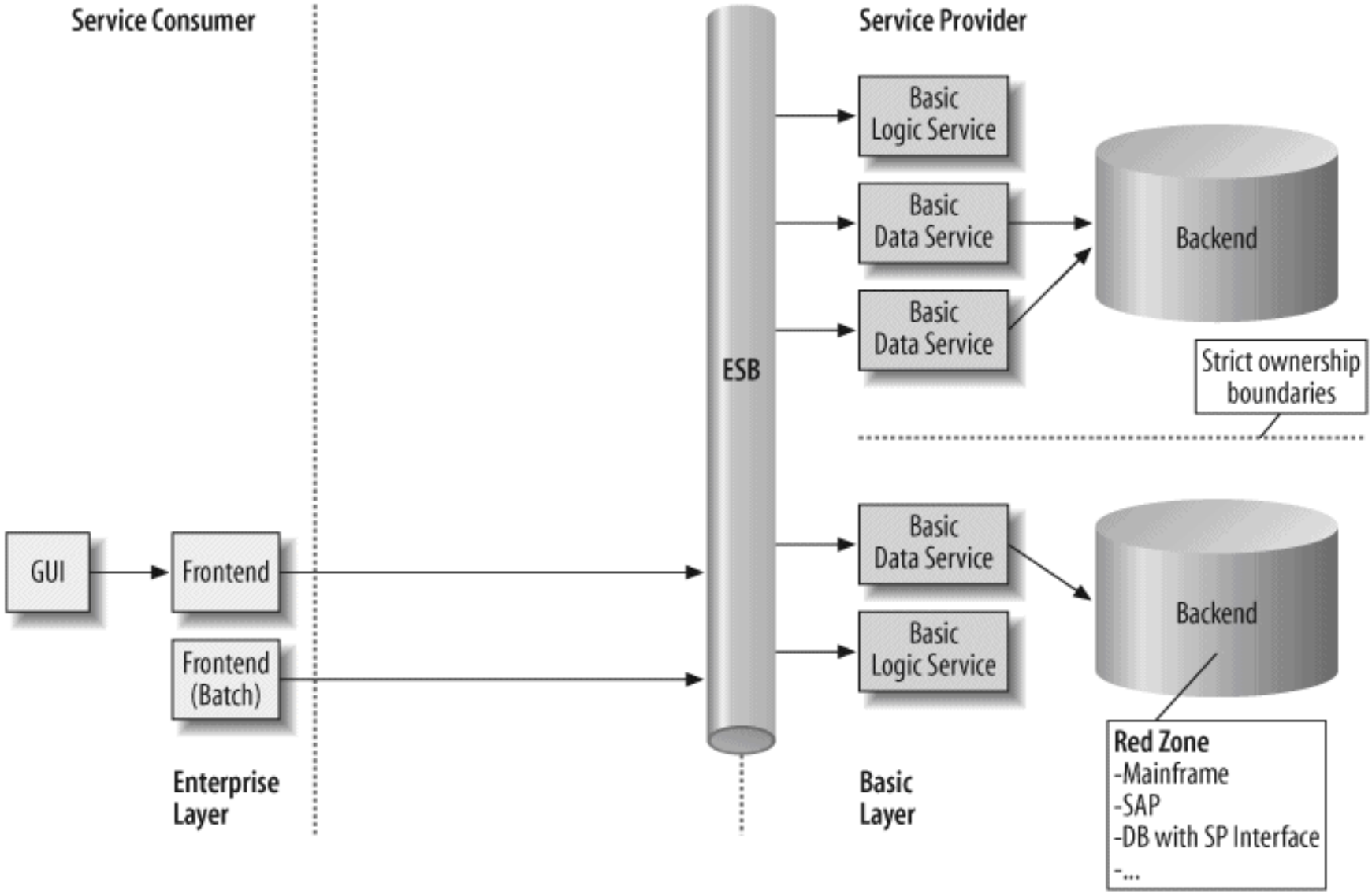
BASIC LOGIC SERVICES

- Basic logic services represent fundamental business rules. These services usually process some input data and return corresponding results.
- Compared with basic data services, basic logic services are a minority.
- Typical examples of basic logic services might be services that:
  - Define product catalogs and price lists.
  - Define rules for changing customer contracts.

FUNDAMENTAL SOA (1)

- By introducing basic services you get the first stage of expansion of SOA, called fundamental SOA.
- With basic services introduced, service consumers can use an ESB to process the business functionality that one backend is responsible for (next slide).

FUNDAMENTAL SOA (2)



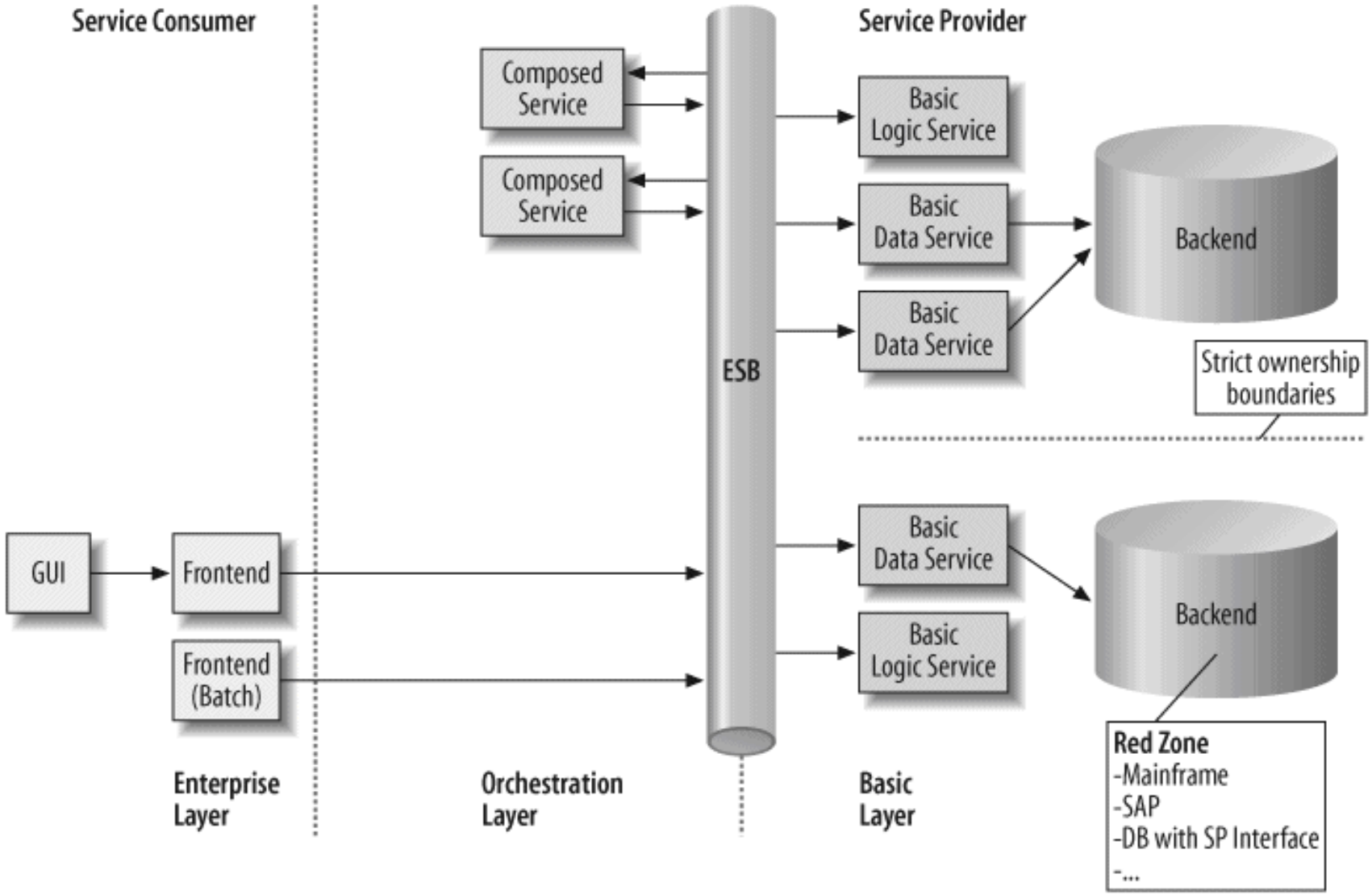
COMPOSED SERVICES

- The second stage of expansion adds composed services. These represent the first category of services that are composed of other services (basic and/or other composed services).
- In SOA terminology, **composing new services out of existing services** is called orchestration or orchestrated services.
- These services operate at a higher level than basic services, but they are still short-term running and conceptually stateless.
- To use a workflow term, a composed service represents a micro-flow, which is **a short-running flow of activities** (services, in this case) inside one business process.

FEDERATED SOA (1)

- The introduction of composed services raises another issue: security.
- Conceptually, the ESB provides interoperability. So, even when a composed service is provided to allow a task such as updating a customer's address consistently across all backends, consumers still have the ability to call the low-level basic services that allow individual changes on each backend.
- So, we need some mechanisms that help to guarantee that those **basic services are called only by higher services**.

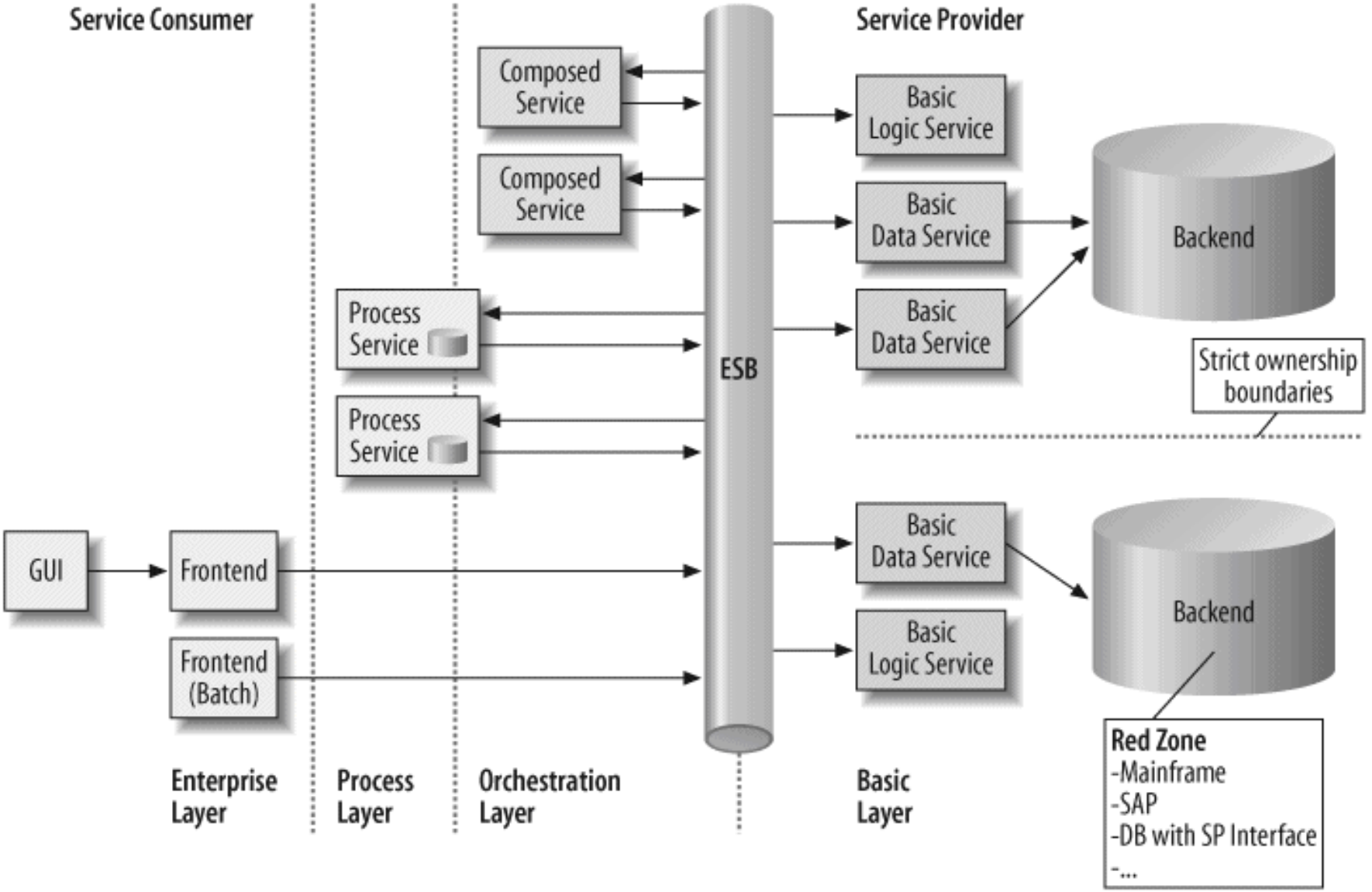
FEDERATED SOA (2)



PROCESS SERVICES

- The third stage of expansion adds process services, which represent **long-term workflows** or business processes.
- From a business point of view a process service represents a macro flow, which is a long-running flow of **activities (services) that is interruptible** (by human intervention).
- Unlike basic and composed services, a process service often manages a long-running workflow that spans multiple service calls; therefore, it must **maintain a persistent state to track progress** and ensure consistency across those interactions.

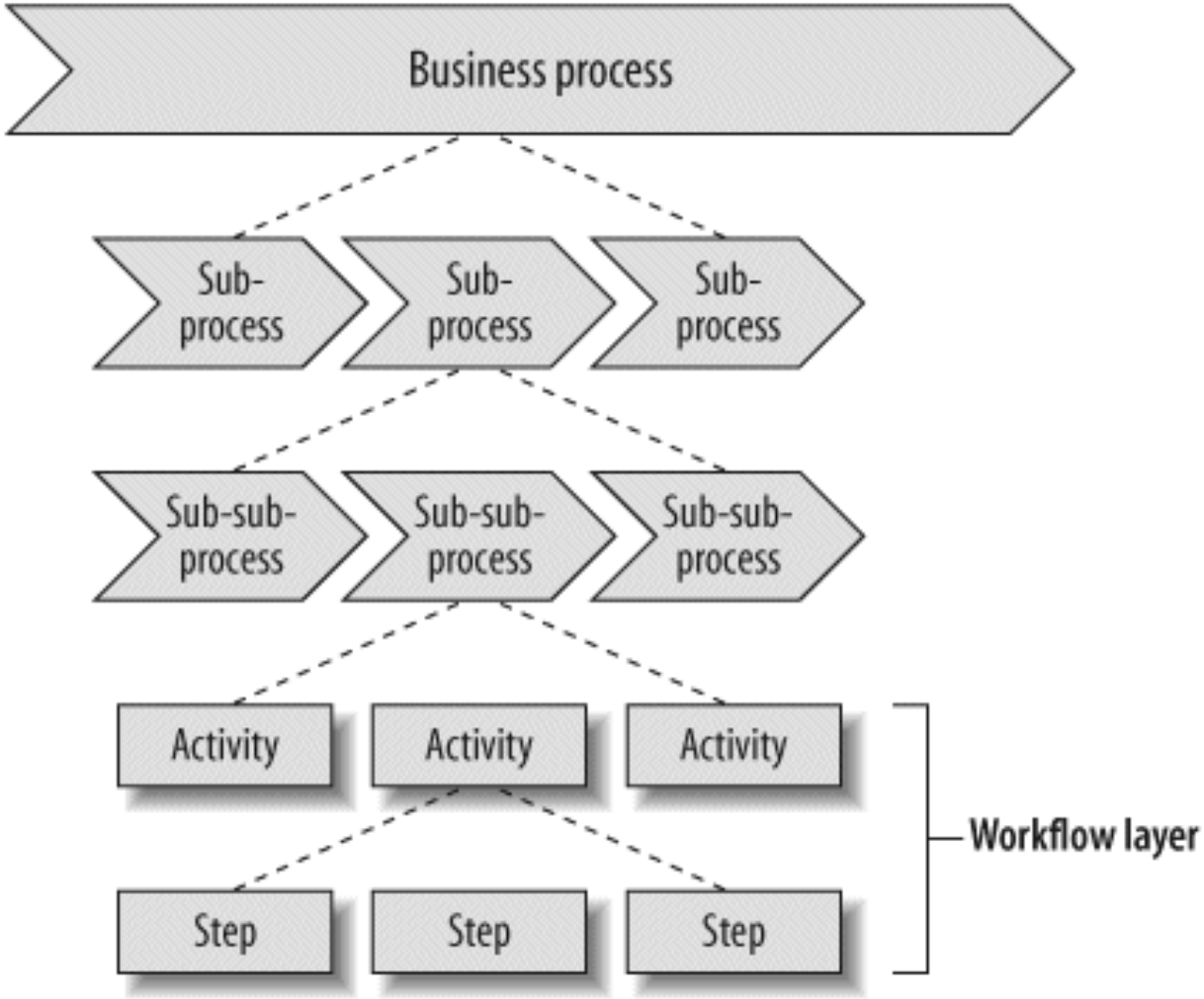
PROCESS-ENABLED SOA



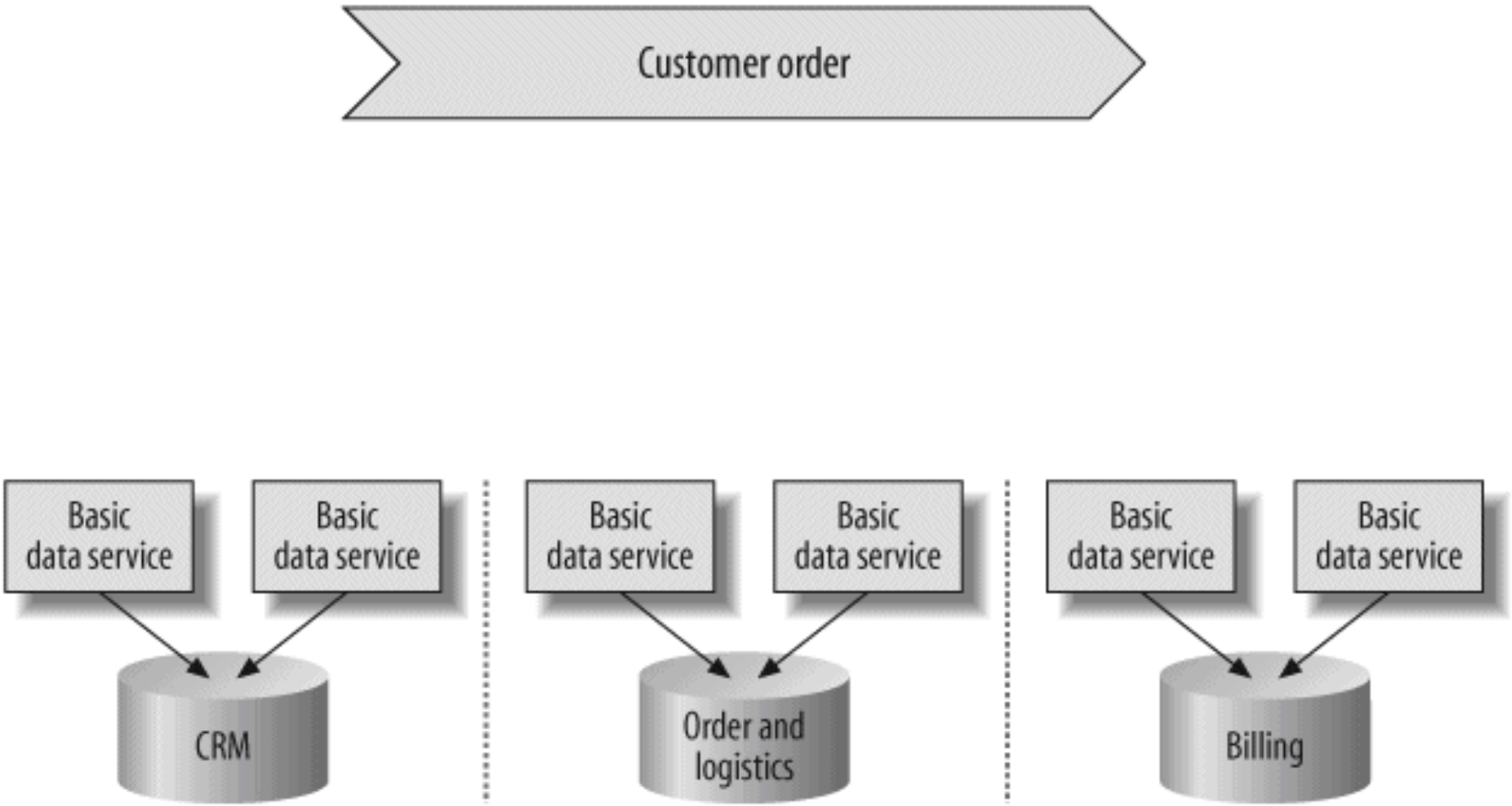
BUSINESS PROCESS MANAGEMENT (BPM)

- In SOA, services are typically parts of one or more distributed business processes. Thus, the main motivations for services come from business processes. And, of course, the question of how to identify services arises.
- This leads to the term of business process management (BPM).

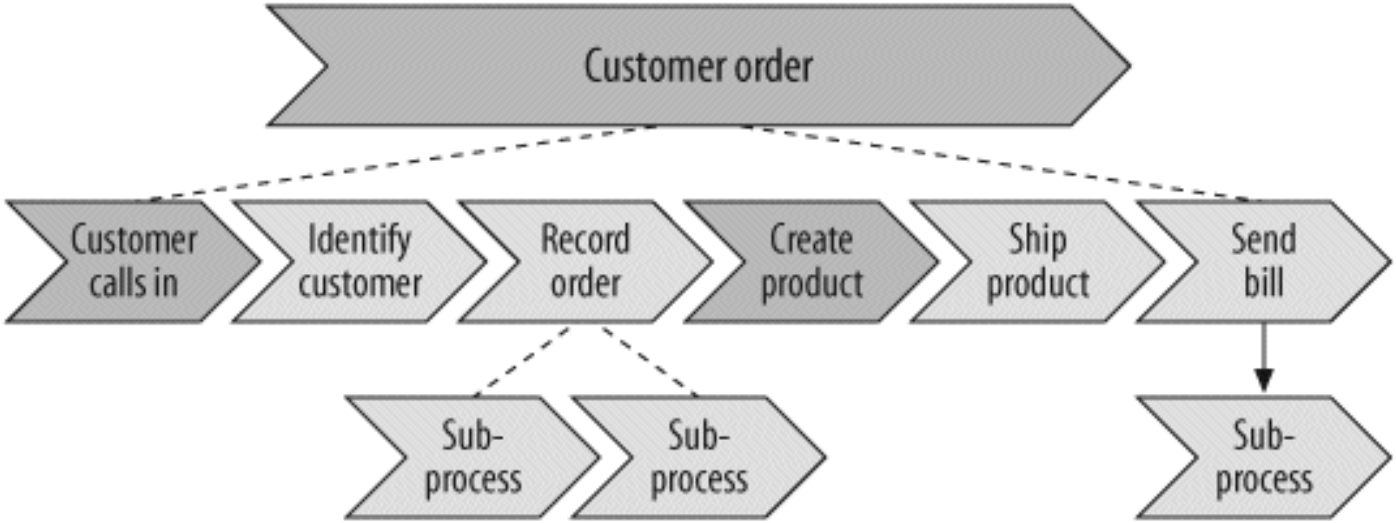
BUSINESS PROCESS HIERARCHY

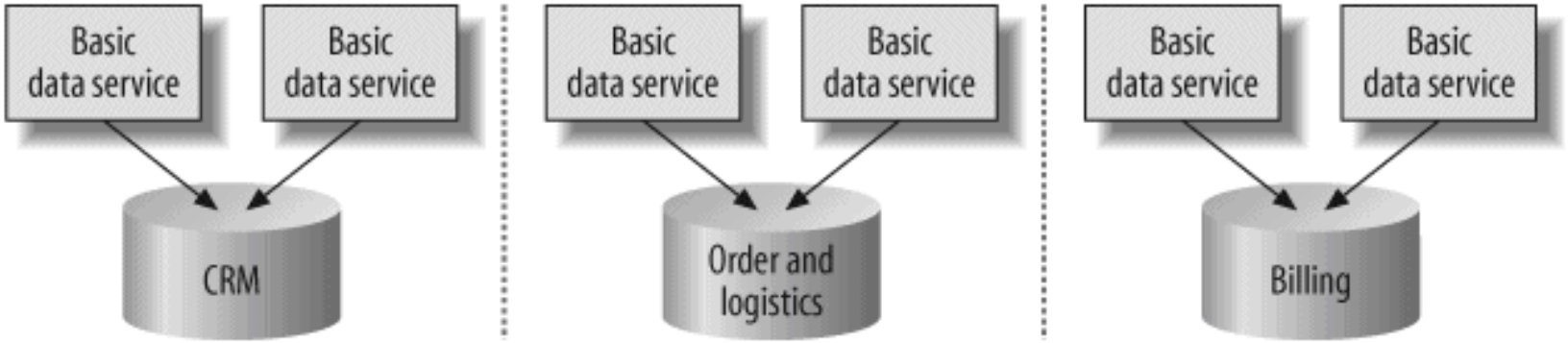


EXAMPLE FOR BPM WITH SERVICES (1)

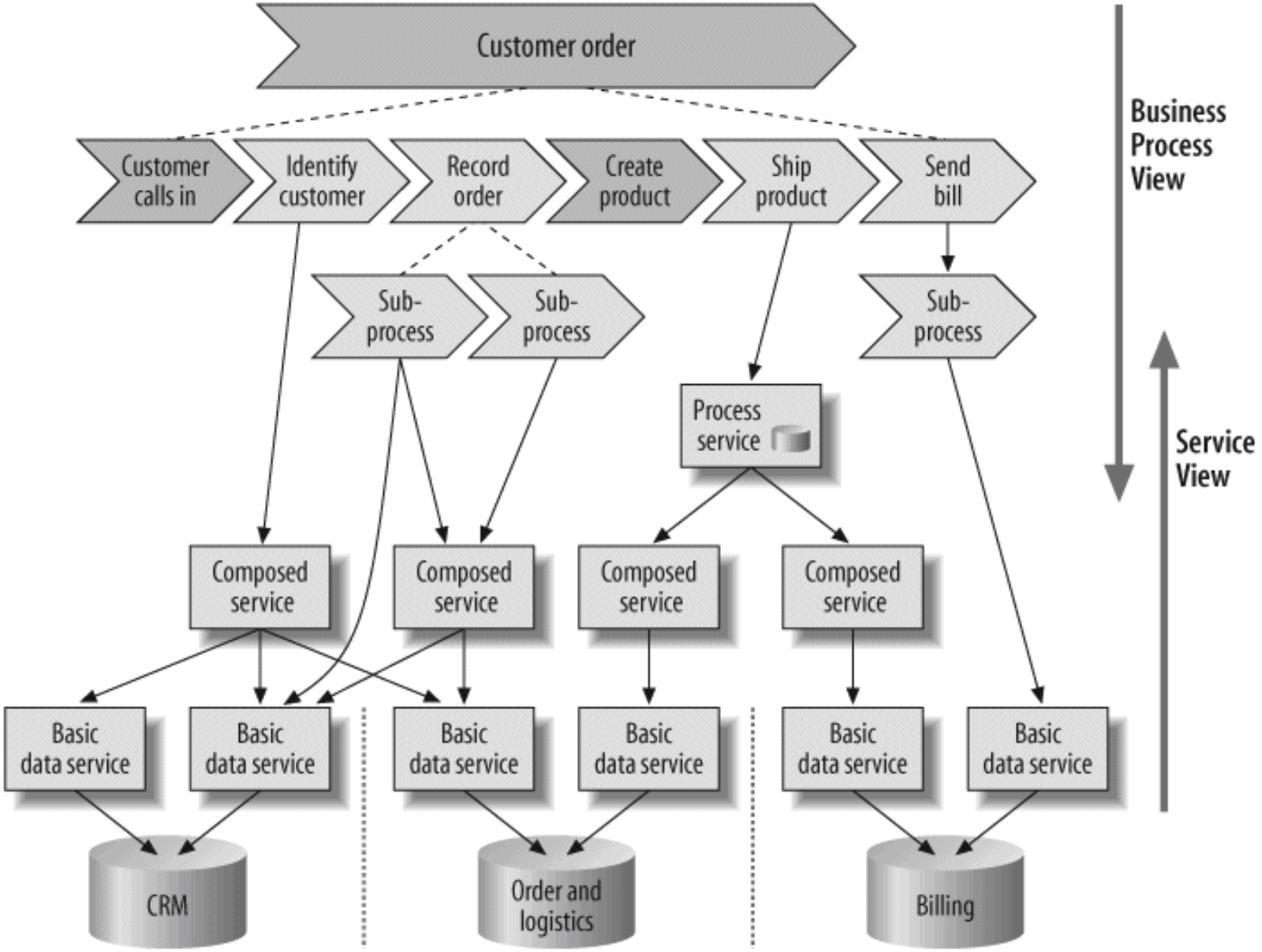


EXAMPLE FOR BPM WITH SERVICES (2)

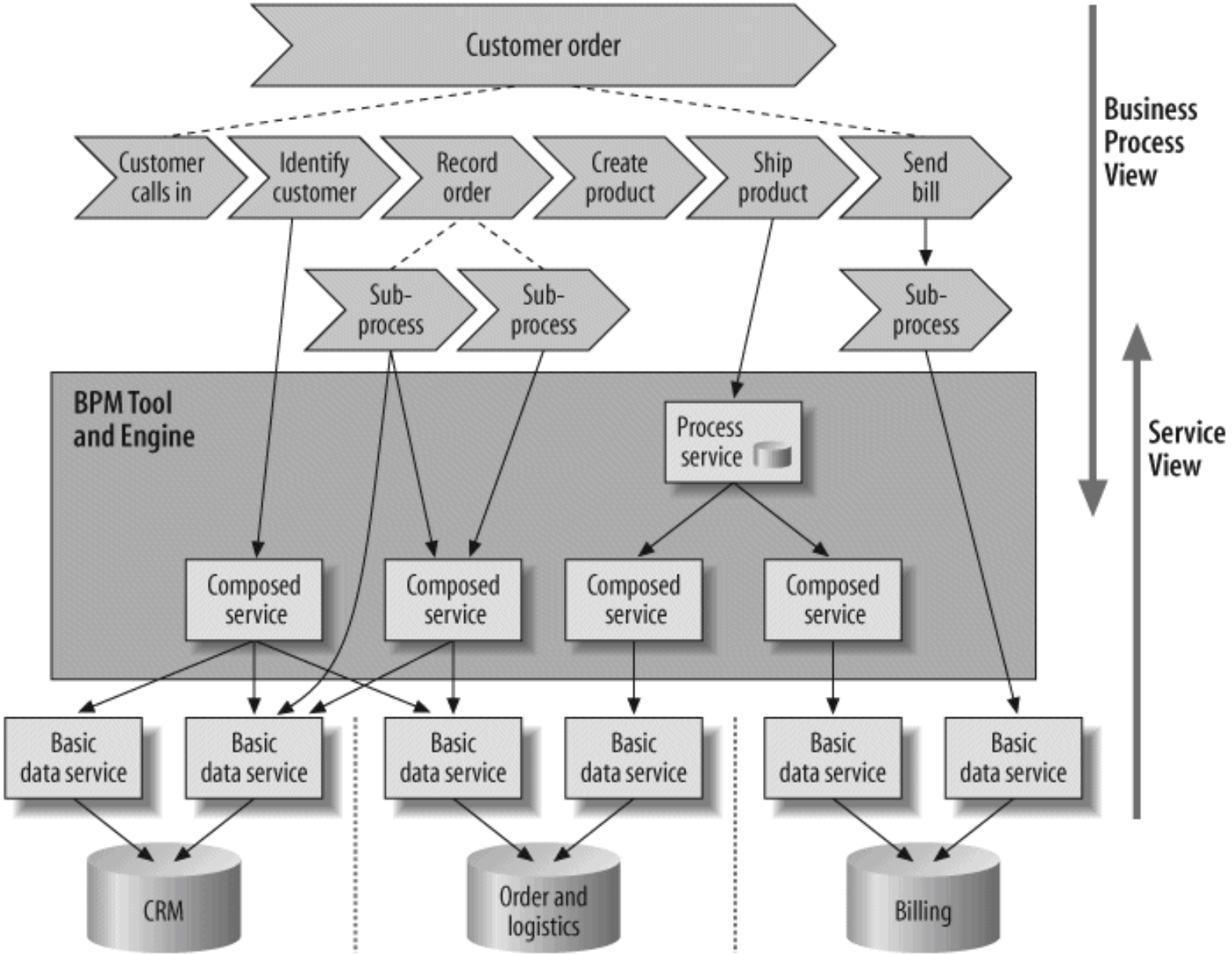




EXAMPLE FOR BPM WITH SERVICES (3)



USING BUSINESS PROCESS MODELING TOOLS



BUSINESS PROCESS EXECUTION LANGUAGE

- The Business Process Execution Language (BPEL) plays a fundamental role in modeling and running business processes in tools and engines.
- BPEL has such momentum in the SOA movement that it is rapidly becoming the standard for designing and running business processes.
- Conceptually, BPEL is an XML language for describing business flows and sequences, which in themselves are services.
- Language elements provide the ability to call services, process responses, and deal with process variables, control structures (including timers), and errors (including compensation).

EXAMPLE FOR BPEL FILE

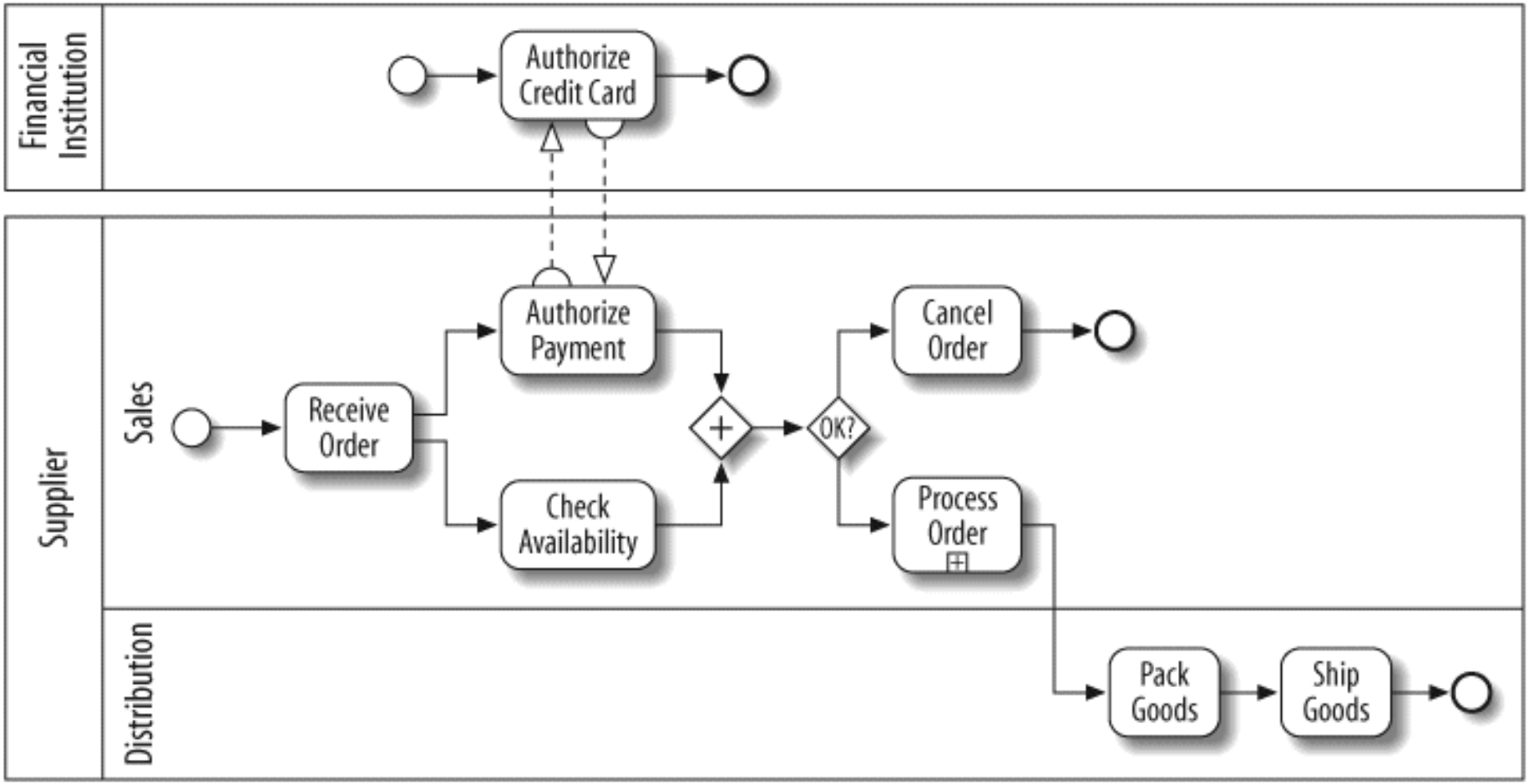
```
<?xml version="1.0"?>
<process name="changeAddress" …>
   <variables>
      variable messageType="…" name="…">
   </variables>
   <flow>
      <receive … />   <!- for request ->
      <invoke … />    <!- call other service ->
      <assign … />    <!- map data ->
      <reply … />     <!- return data ->
   </flow>
</process>
```

BUSINESS PROCESS MODEL AND NOTATION

- Business Process Model and Notation (BPMN) is the global standard for process modeling and one of the most important components of successful Business-IT-Alignment.

* For more information: http://www.bpmnquickguide.com

EXAMPLE OF A BPMN DIAGRAM



ASSIGNMENT 6

- จงออกแบบ Services ของ “ระบบจองห้อง”
  - Business Process ของการ “การจองห้อง"
  - ออกแบบ Service เพื่อรองรับ Business Process
    - Basic / Composed / Process

MICROSERVICES

MODULARITY

Monolithic Application

Modular Application

MICROSERVICES (1)

- Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of loosely coupled services, which implement business capabilities.
- The microservice architecture enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack.

MICROSERVICES (2)

1990s and earlier	2000s	2010s
Coupling		
Pre-SOA (monolithic)	**Traditional SOA**	**Microservices**
Tight coupling	Looser coupling	Decoupled

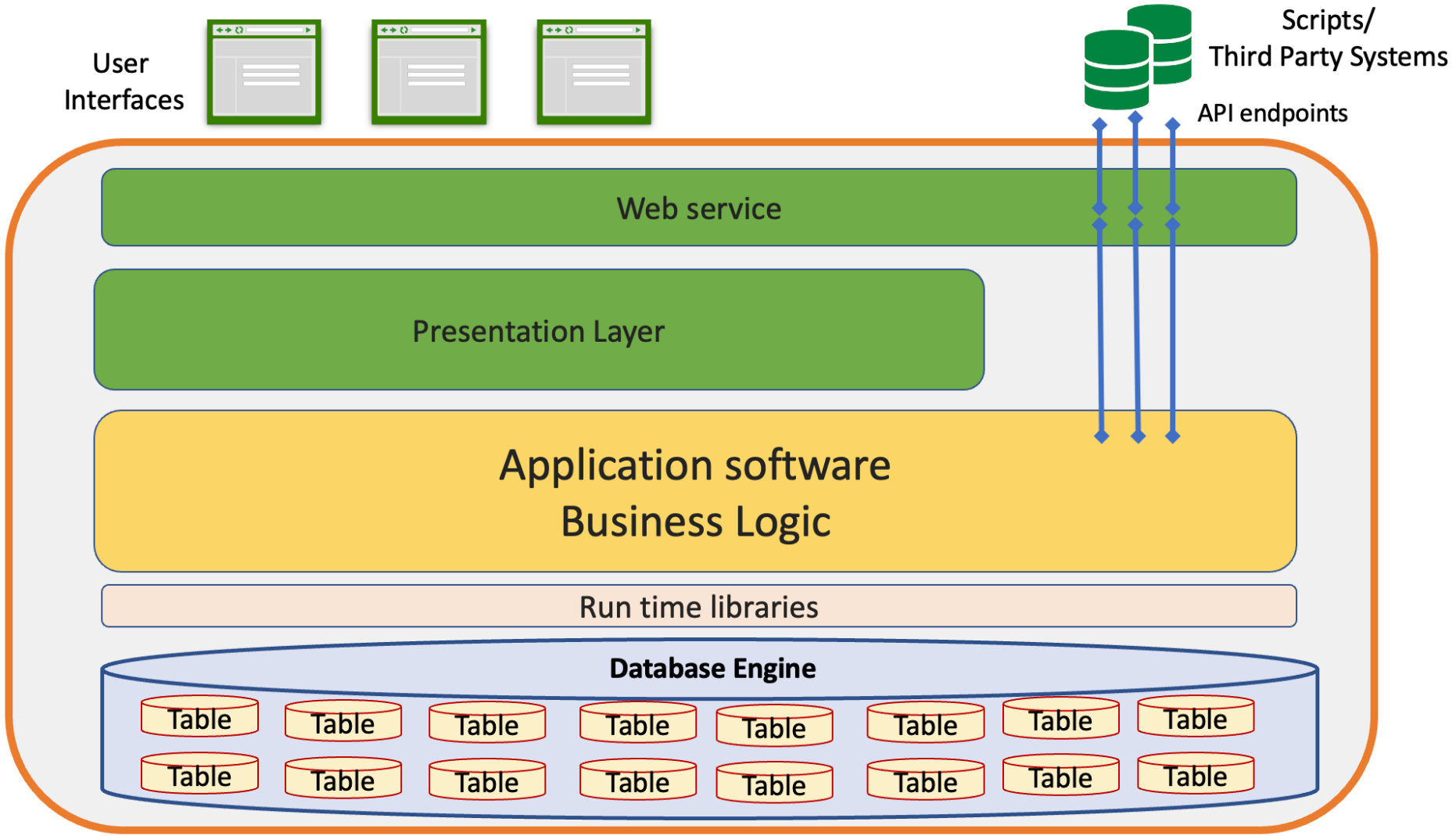
SOFTWARE DEVELOPMENT STYLES

- Monolithic Applications
  - Codebase of application deployed as a single bundle of executables and libraries on a unified platform
- Microservices Architecture
  - Multiple independent software components orchestrated to form a unified application
  - Common infrastructure:
    - User interface toolkit
    - API Gateway
    - Persistence layer

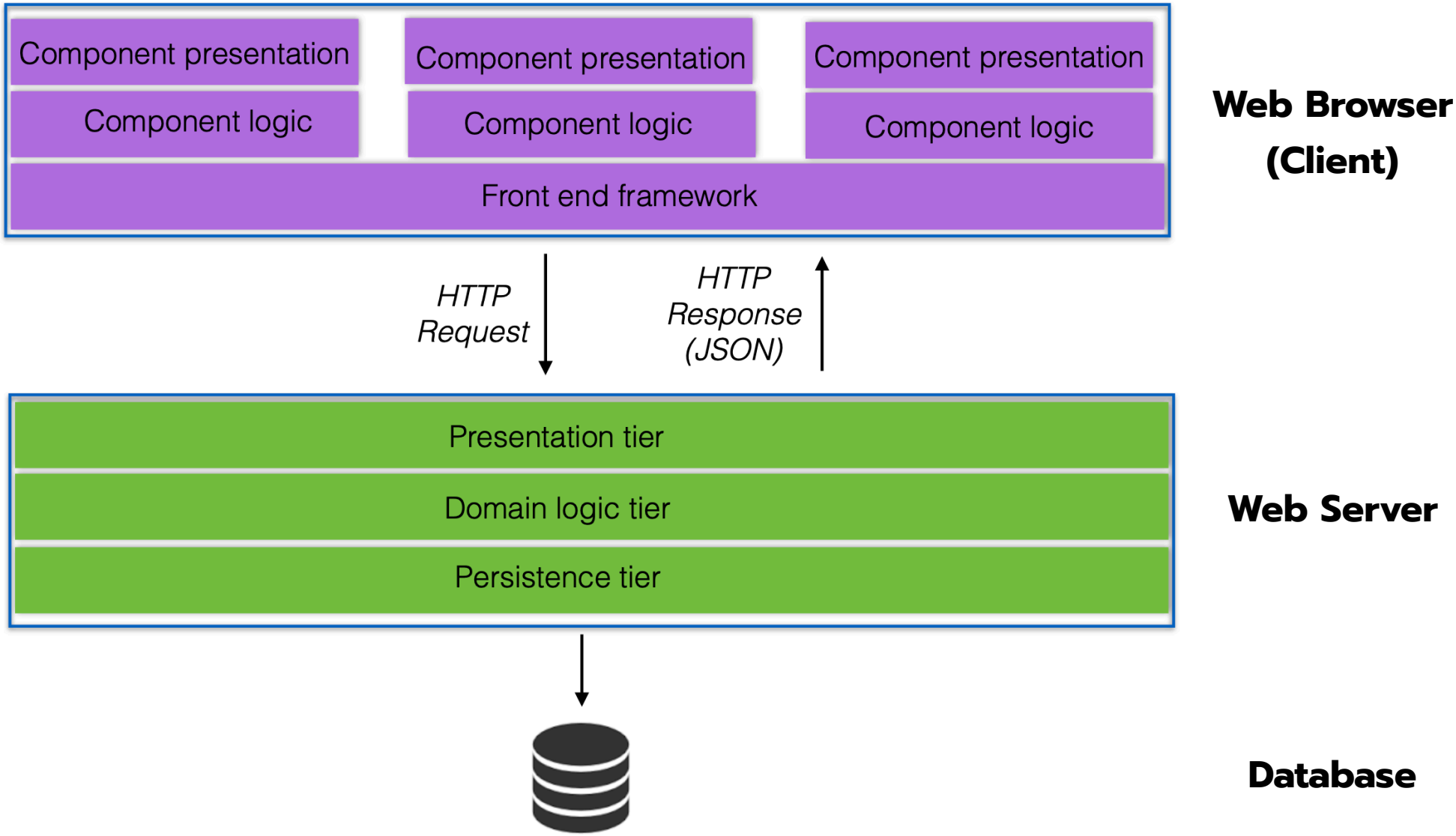
MONOLITHIC SOFTWARE (1)

- Consolidated executable application
  - Plus supporting libraries and modules
- Can be massively distributed across computing clusters
- Entire application based on a uniform technology stack:
  - Server platform
  - Operating system
  - Programming language
  - Database layer
- Enhancements mean recompilation of entire application

MONOLITHIC SOFTWARE (2)



MONOLITHIC BACK-END



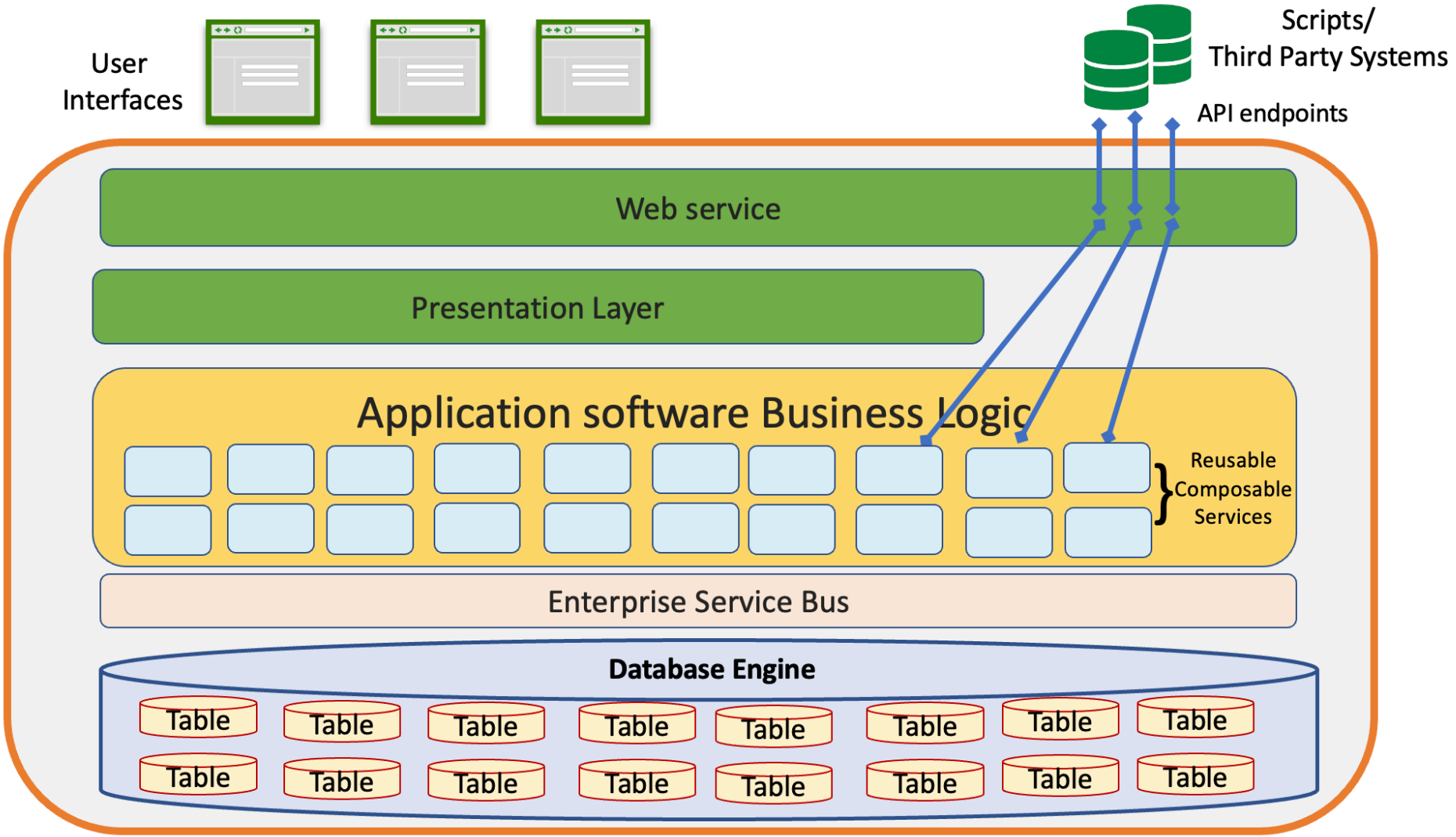
SERVICES-ORIENTED ARCHITECTURE

- Monolithic application based on reusable services
- Complex applications rely on an enterprise service bus to manage communications among services, database connectivity, event triggers, etc
- Single uniform technology platform
- Code assembled into a monolithic package
- Scales to very high performance through clustered deployment

SOA DEVELOPMENT ISSUES

- Services are closely interrelated throughout the application
- Developers must understand all aspects of the application
- Single technology stack
- Small changes require full recompilation
- Complex applications can hit hardware or OS limits
- Centralized development pattern
- Operations separated from Development

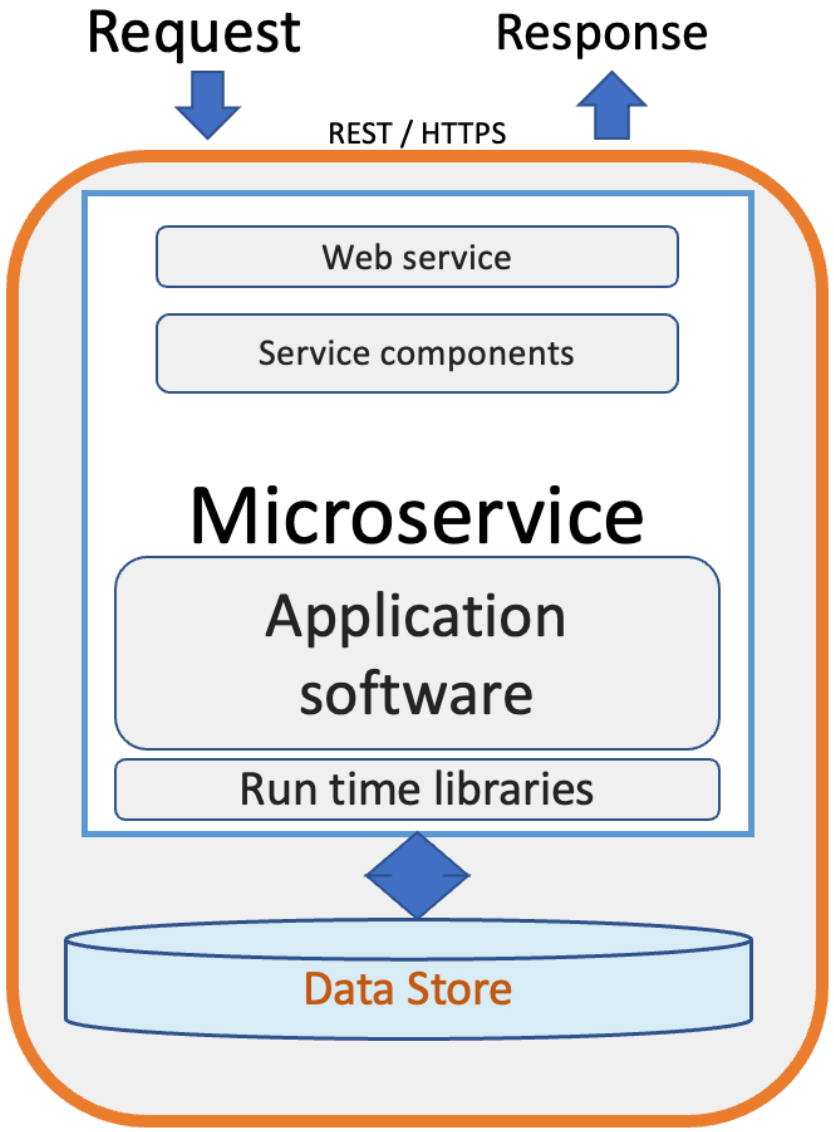
MONOLITHIC: SOA MODEL



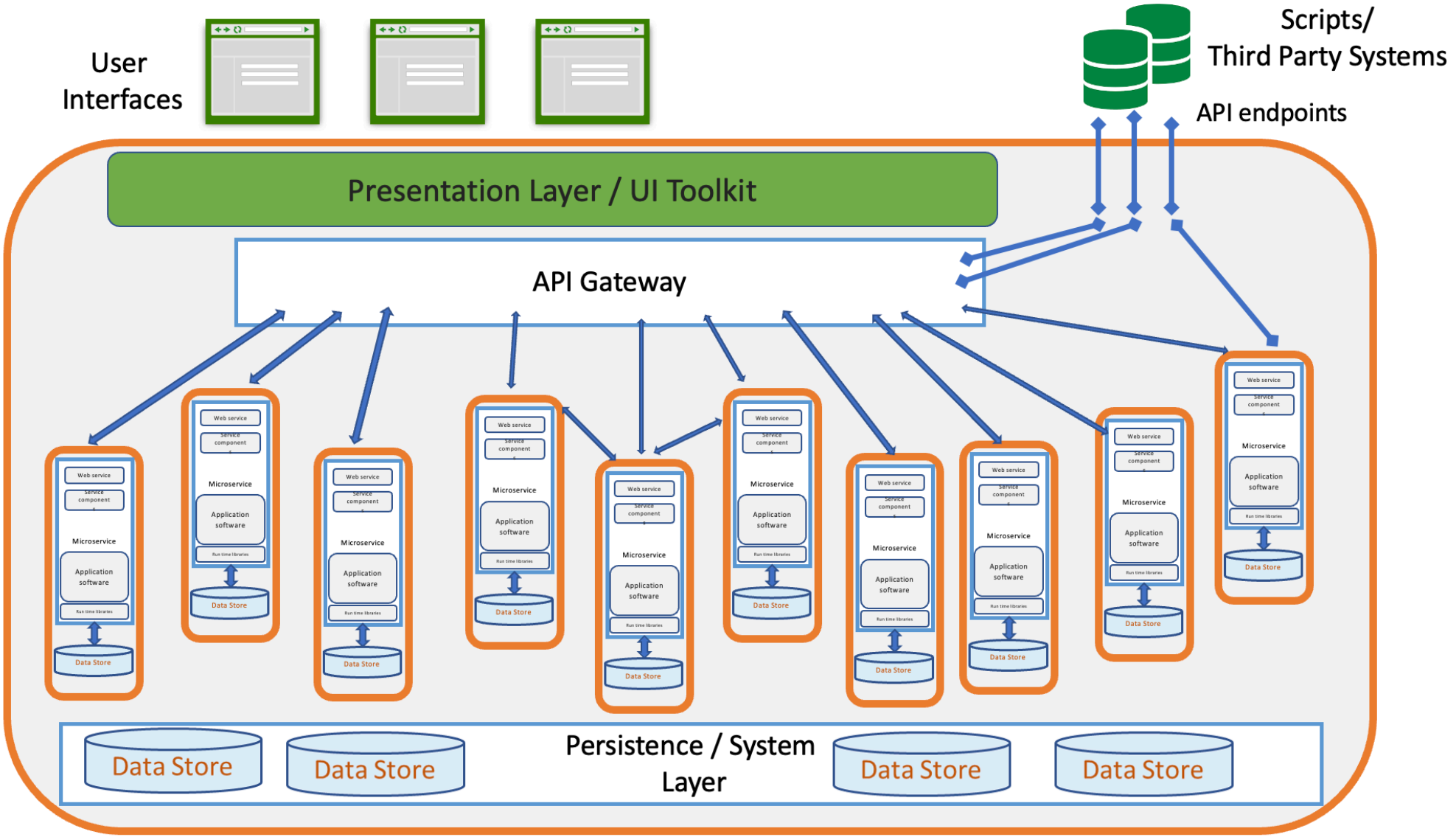
BUILDING A MICROSERVICE

- Small unit of functionality
- Complete and independent technology stack
- Separate data stores
  - Synchronization with other services as needed through persistence layers
- Invoked through API Request / Response
- Usually: REST, HTTP, JSON
- Self-contained components
- Inner workings not exposed externally
  - Developers have free reign to select tech components

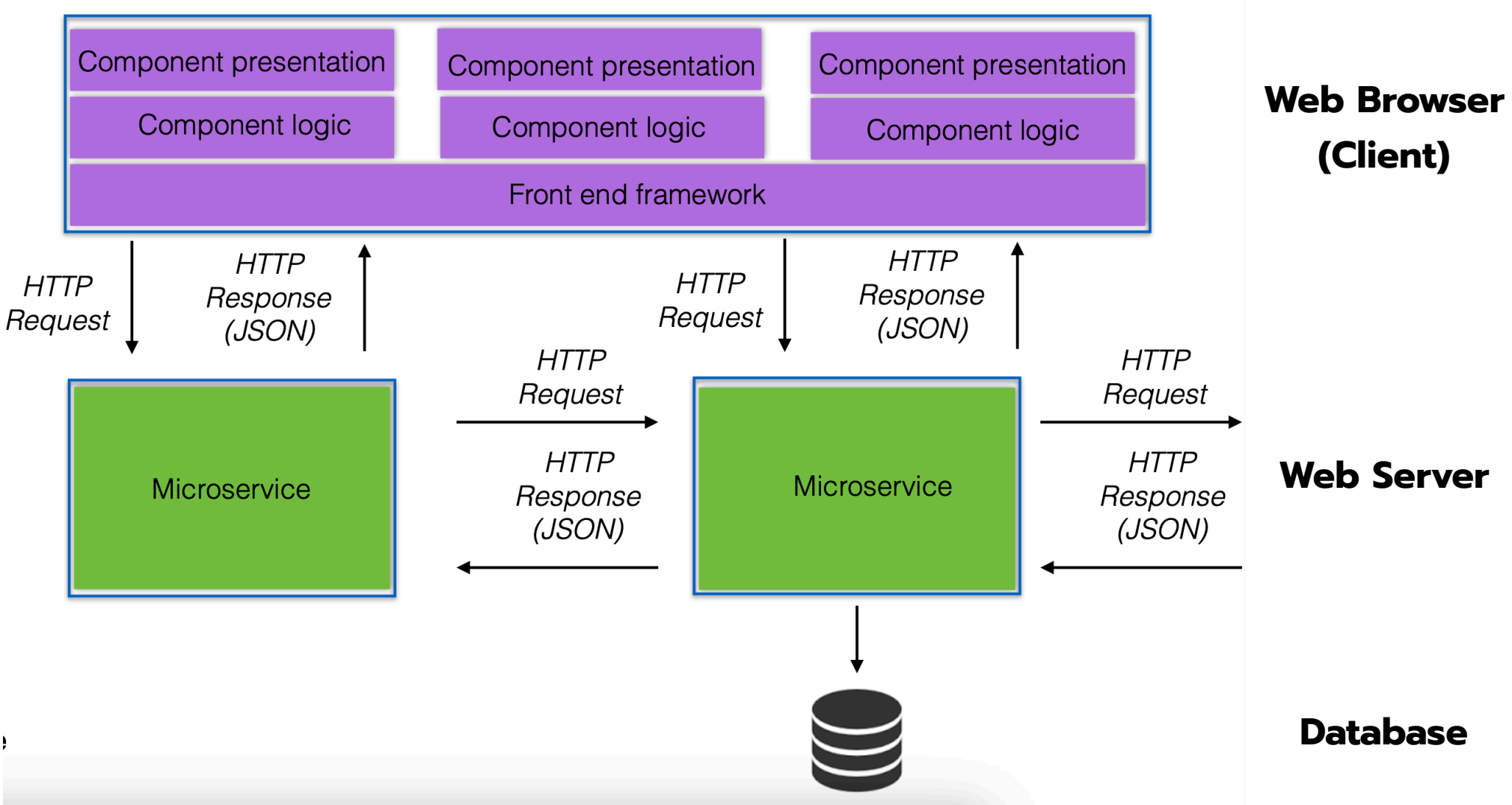
MICROSERVICE CONCEPTUAL MODEL



MICROSERVICE CONCEPTUAL MODEL



MICROSERVICES BACK-END



RESTFUL API (1)

- A RESTful API is an architectural style for building web services that use standard HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources identified by URLs.
- It emphasizes stateless communication, uniform interfaces, and scalability through simple, lightweight interactions.

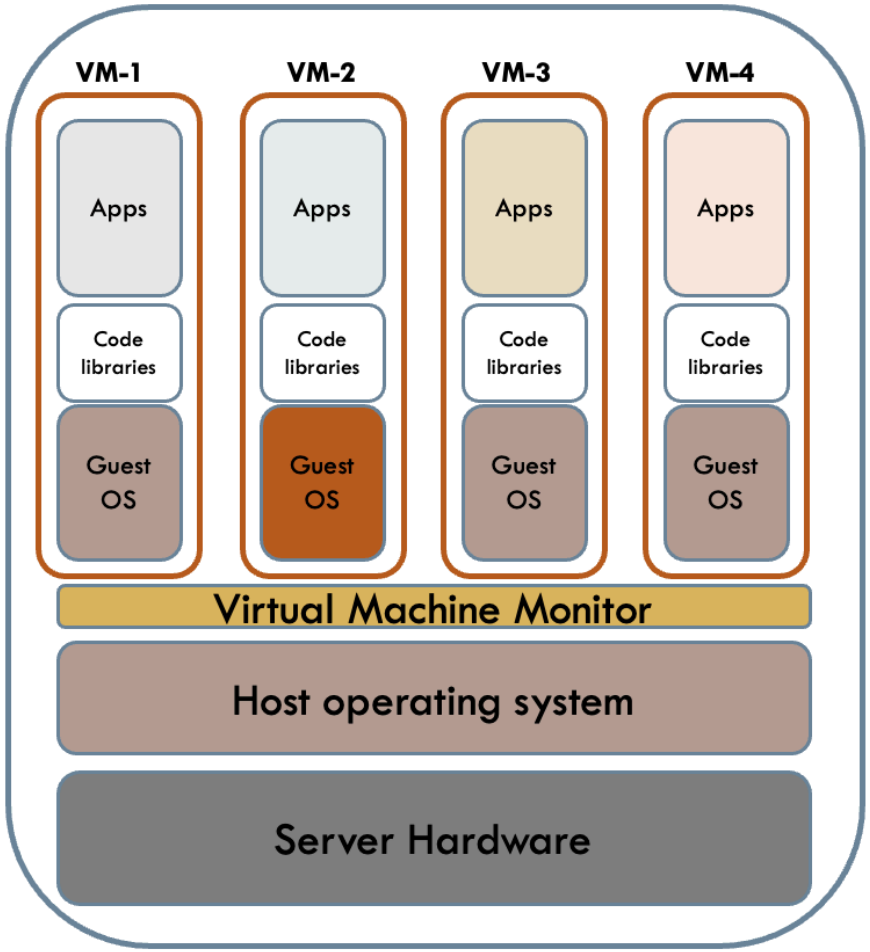
RESTFUL API (2)

- Support Scaling
  - Use HTTP actions to support intermediaries
- Support Change
  - Leave anything out of URI that might change
  - Ensure any URI changes are backwards compatible
- Support Reuse
  - Design URIs around resources that are expressive abstractions that support a range of client interactions.
  - Resources are nouns; use HTTP actions to signal verbs (GET, POST, PUT, DELETE)

VIRTUAL MACHINES TO CONTAINERS

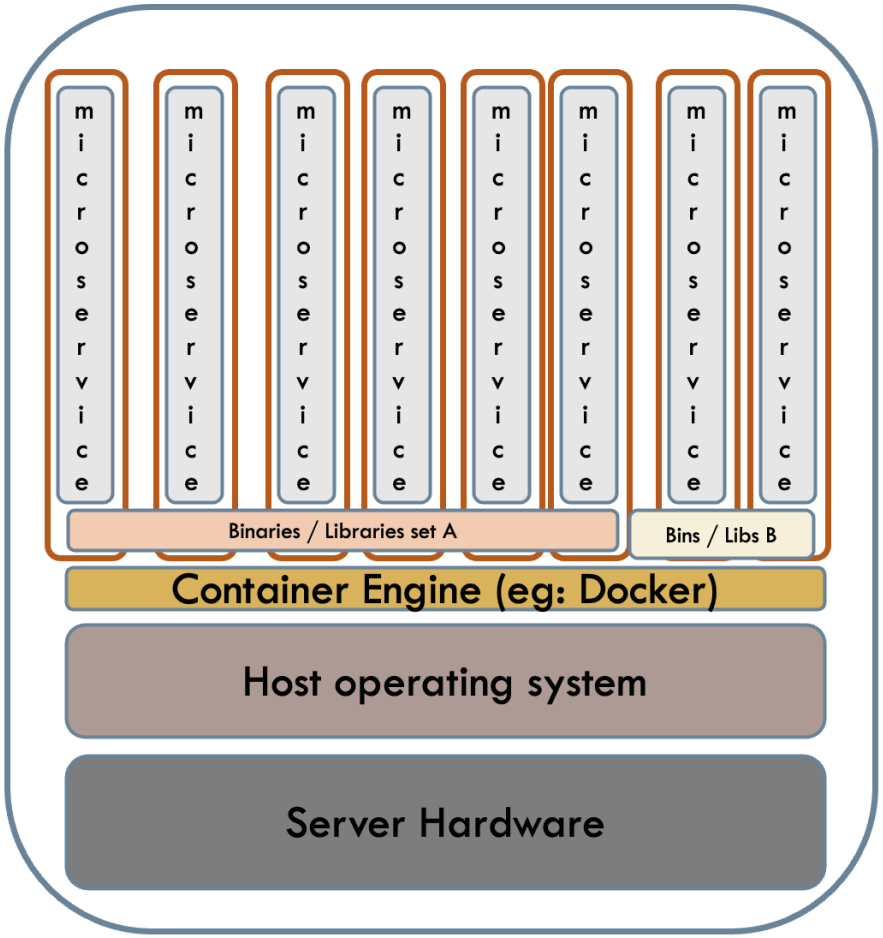
- Virtual machines are a common technology for optimizing use of computing hardware
- Most applications use a small portion of computing and storage capacity
- VM managers enable multiple instances of operating environments to co-exist on each physical server

VIRTUAL MACHINE ENVIRONMENT



- Multiple Virtual Machines can share a single physical server
- Each VM contains its own operating system, code libraries, applications, web services, and other components
- Each VM independent: can host any OS, libraries, apps

CONTAINERS



- A container in Docker is a lightweight, isolated runtime environment that packages an application with all its dependencies, ensuring consistent behavior across different systems.
- It shares the host OS kernel but runs independently, enabling portability, scalability, and efficient resource utilization.

API GATEWAY

- API Gateway is a single entry point for clients to access multiple microservices, handling request routing, authentication, rate limiting, and protocol translation, which simplifies client interactions and enables flexible service evolution.

DEVOPS

- Merges **dev**elopment with **op**erations
- A single team takes responsibility for development, deployment, operations, and maintenance of a microservice
- Make independent technology decisions
  - Programming language
  - Databases
  - Etc.
- No need to understand all aspects of the application

DOCKERS

- Leading technical environment for the management of containers
- Supported in most infrastructure environments

CONTAINERS

- Allocate computing resources in even smaller increments than Virtual Machines
- Share low-level components
- Self-contained pre-configured operating environment
- Container management environments provide fast and easy approach to deploy microservices or other computing components
- Used in all types of computing environments, but especially well suited to microservices.

CONTAINER ORCHESTRATION

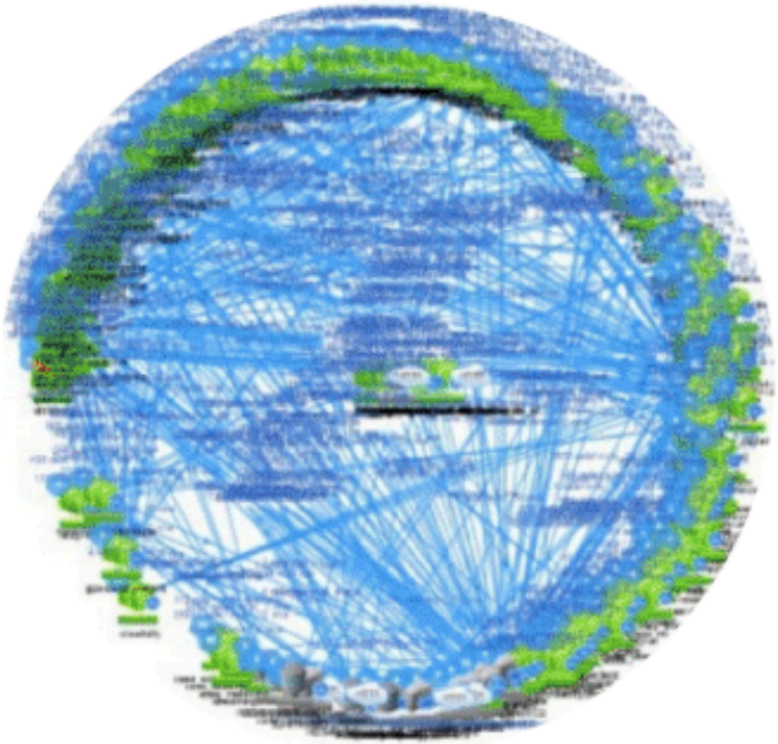
- In a complex environment tools are needed to manage the deployment of containers
- Components need to be allocated and deallocated dynamically based on load and demand
- Examples:
  - Docker Swarm
  - Kubernetes (developed by Google)
  - Apache Mesos

BENEFITS OF MICROSERVICES

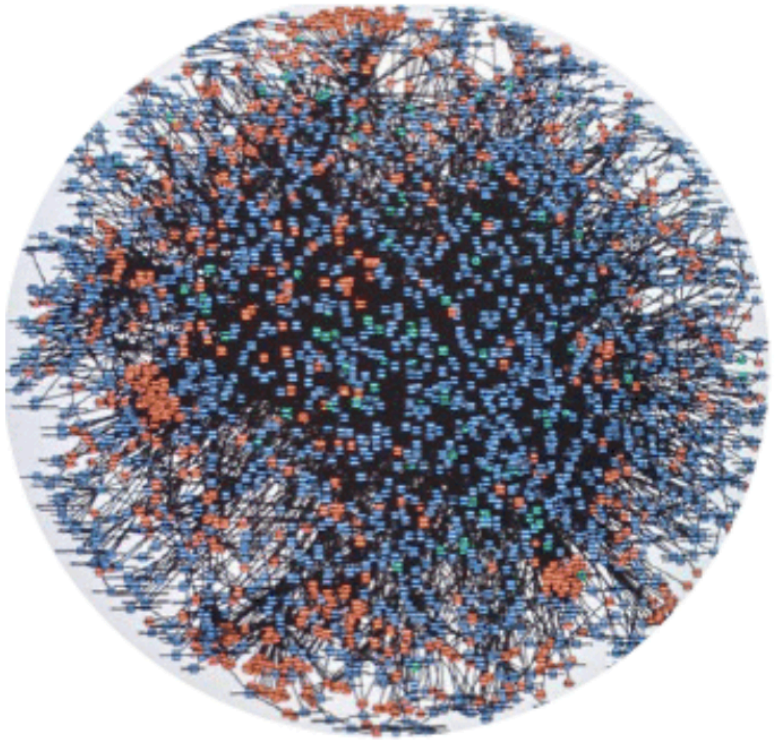
- Rapid development
- Quick path to MVP (minimal viable product)
- Distributed teams
- Modularity
- Scalability

WHO USES MICROSERVICES

NETFLIX & AMAZON

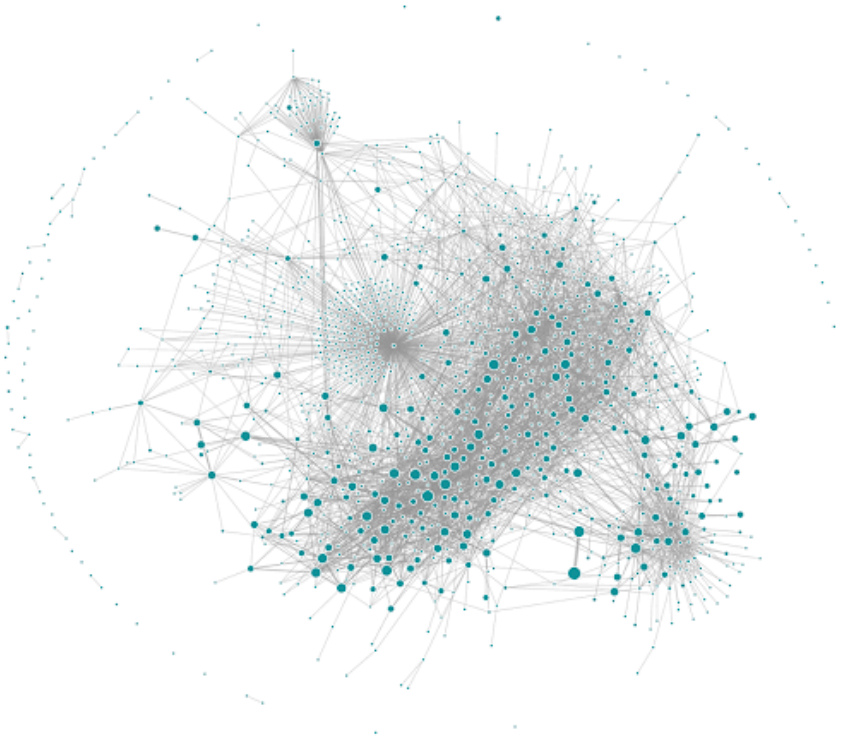


Netflix

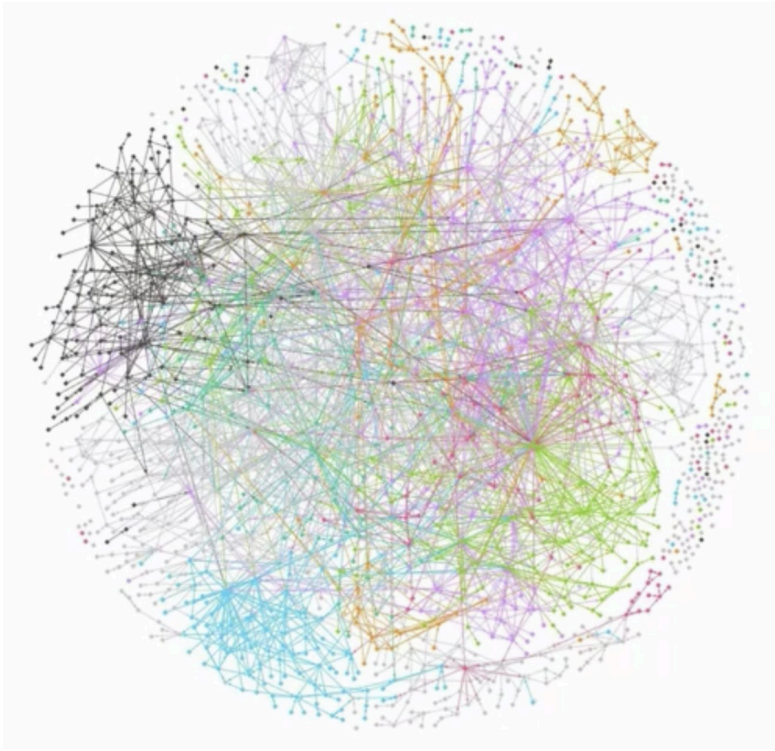


Amazon

UBER & X (TWITTER)

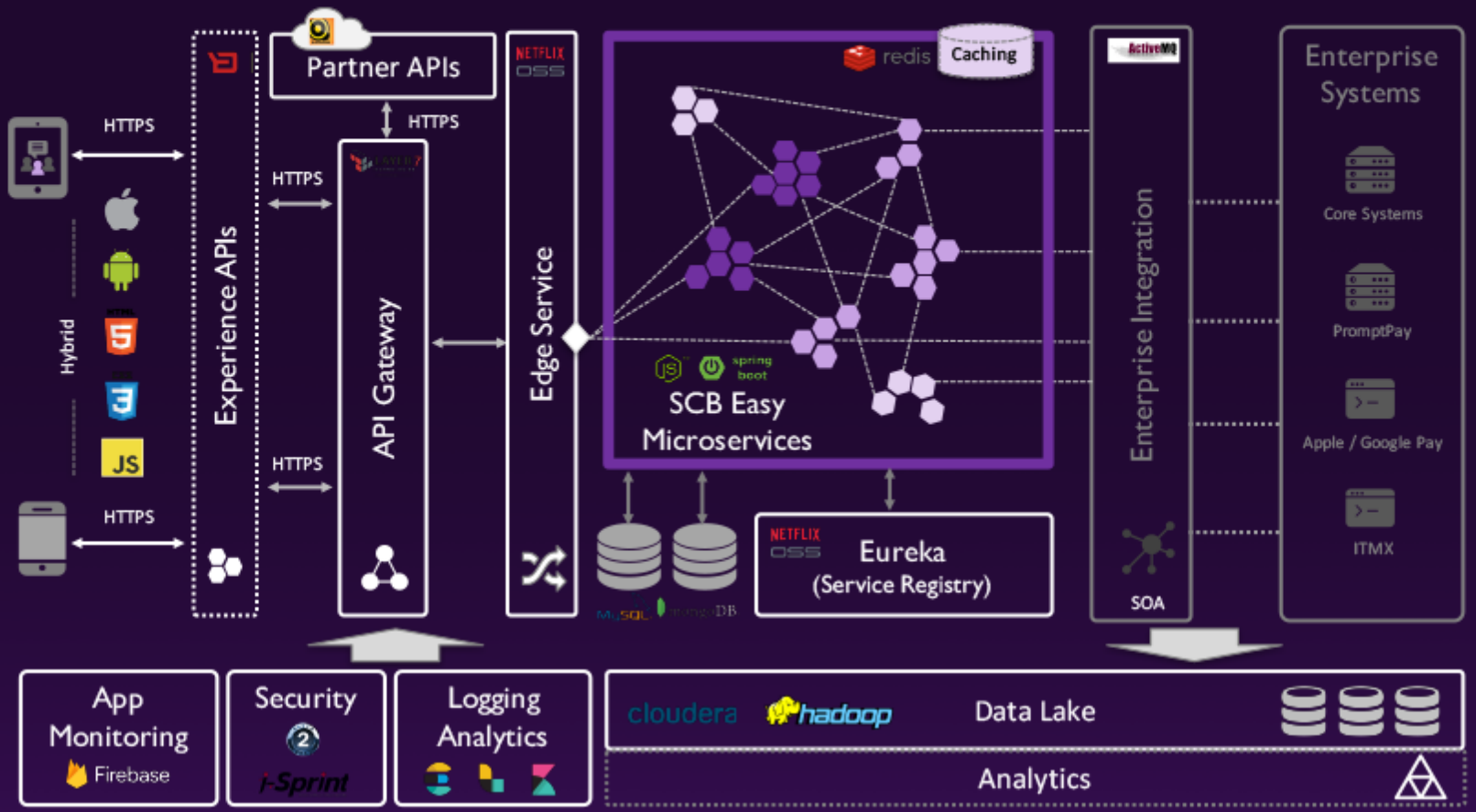


Uber



X (Twitter)

SCB EASY MICROSERVICES ARCHITECTURE



8 DO + DON'T DO ON MICROSERVICES

- Do — Follow 12 factor-app (https://12factor.net)
- Don't — Use shared database
- Do — Use docker + good container orchestration
- Don't — Write too small service
- Do — Automate build + test + deploy
- Don't — Implement distributed transaction
- Do — Centralize log
- Don't — Rely on smart message queue