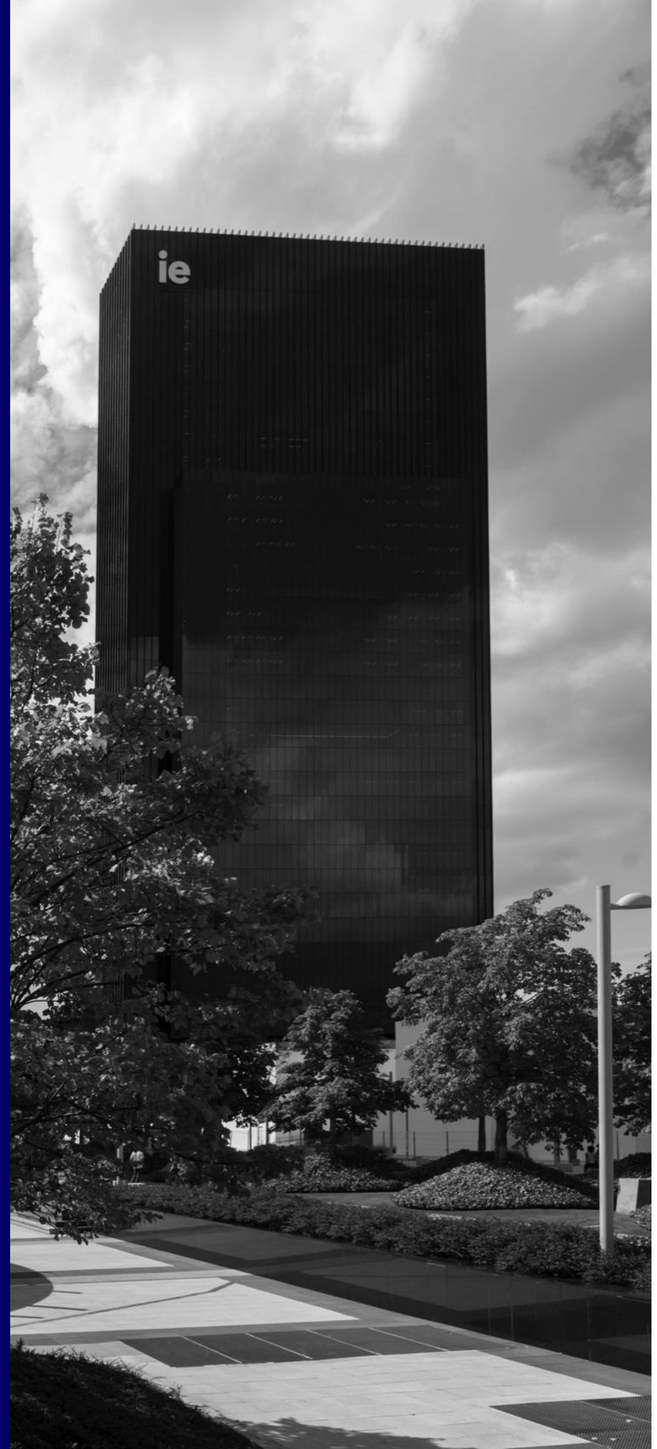


SOFTWARE DEVELOPMENT AND DEVOPS

Software Architectures

ie UNIVERSITY
SCHOOL OF SCIENCE & TECHNOLOGY

Monolith Architecture

Monolith Architecture

- **Traditional approach** to building software
- All components are combined into a **single, unified codebase**
- The term “monolith” started being used **as opposite to distributed** architectures (microservices)
- A well-designed monolith **can be the best choice** for many applications
- **Negative connotations** about monoliths usually stem from **poor code organization**, not the monolithic deployment

Monolith Deployment Model

- **Single Codebase**
  - All code lives in one repository
- **Single Build Process**
  - One compilation/build creates the entire application
- **Single Deployment Unit**
  - You deploy everything together as one artifact
- **Shared Memory Space**
  - Components communicate through direct method calls, not network requests
- **Single Database (typically)**
  - All components share the same database instance

Unstructured Monolith

- No clear architectural pattern
- Components **tightly coupled** with tangled dependencies
- Code mixed together **without clear boundaries**
- **This is the "bad monolith"** people warn about
- Often the result of **technical debt** accumulation

Layered Monolith

- Uses **horizontal layers** (presentation, business, data)
- Clear **separation** of concerns by **technical function**
- Still deployed as **one unit**
- Better than unstructured, but can still have **coupling issues**

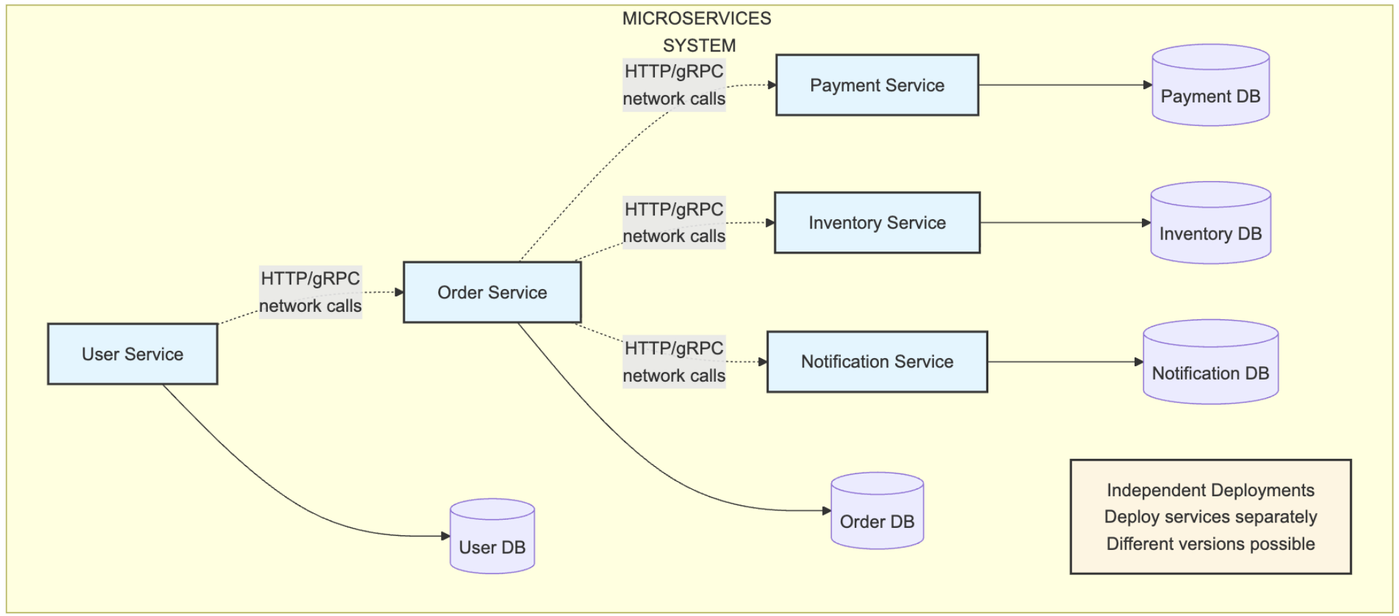
Modular Monolith

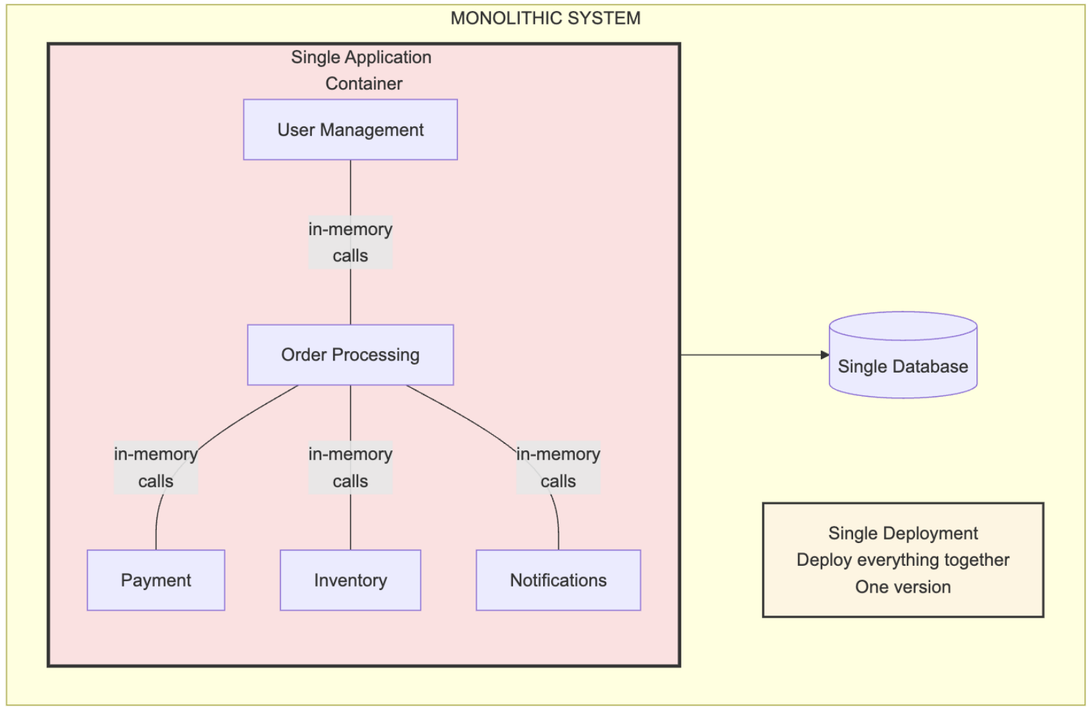
- Organized around business capabilities or **domains**
- **Each module is self-contained** with its own logic and data
- Modules communicate through **well-defined interfaces**
- This is the **"good monolith"** that combines monolithic deployment with **microservice-like modularity**
- Often the **sweet spot** for many applications

Modular Monolith with Internal Architecture

- **Each functionality** internally uses layered, hexagonal, or other **patterns**
- **Best of all worlds**: modular organization, clean architecture, single deployment
- **Most complex to build but easiest to maintain** and potentially migrate later

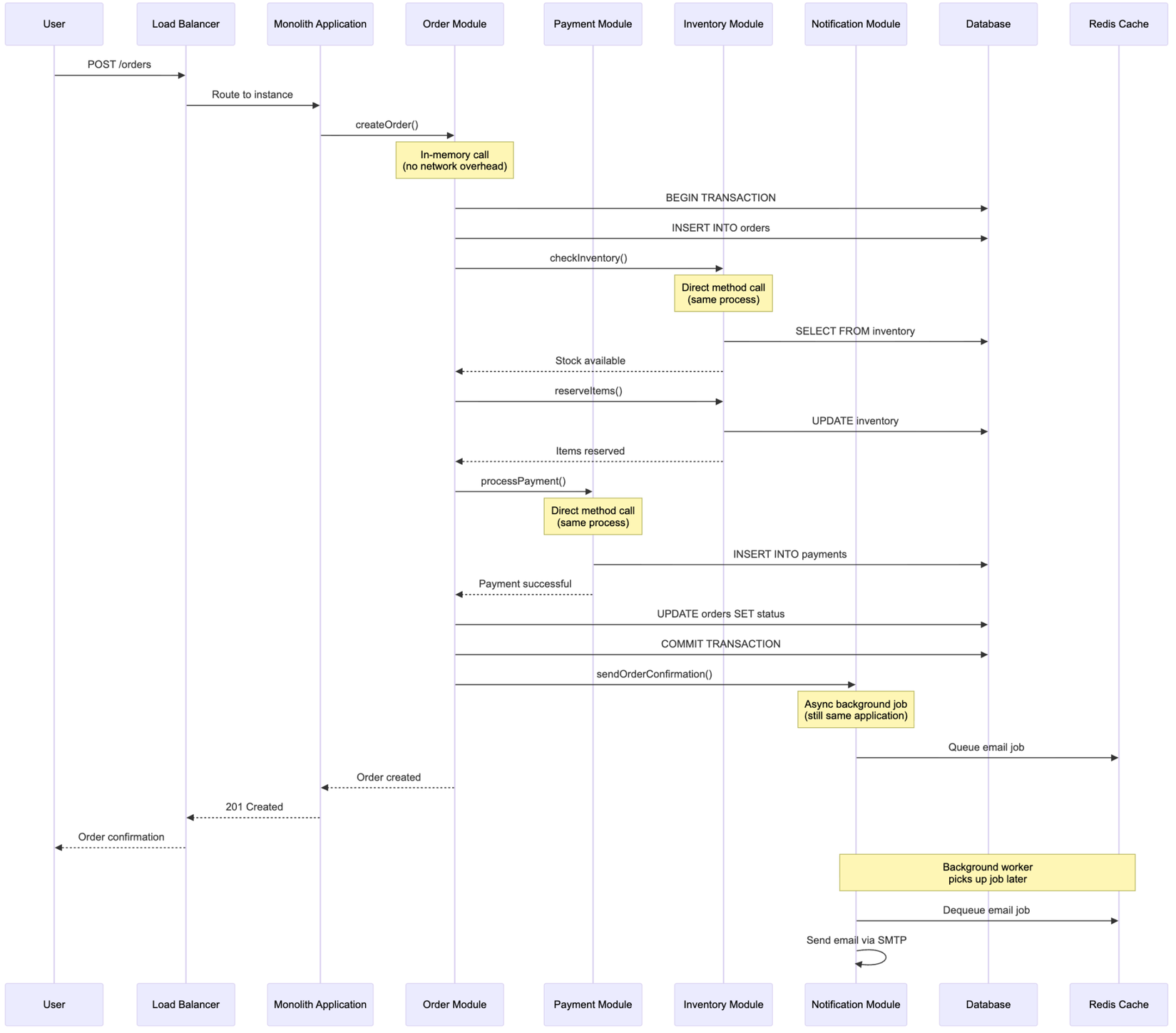
Monolith vs Microservices





How a Monolith Works:

Request Flow Example



UNIVERSITY

SCHOOL OF SCIENCE & TECHNOLOGY

Monolith Advantages

- Simplicity and Ease of Development
- Strong Data Consistency
- Performance Efficiency
- Simpler Deployment
- Easier Testing
- Lower Operational Complexity
- Easier Refactoring
- Better for Small to Medium Teams

Monolith Disadvantages

- Scaling Limitations
- Long Build and Deployment Times
- Technology Lock-in
- Team Coordination Challenges
- Deployment Risk
- Tight Coupling Risk
- Limited Fault Isolation
- Onboarding Complexity (for large monoliths)
- Database Bottleneck
- Slower Innovation

Monolith Health Indicators

Healthy architecture

- Feature velocity remains high
- Deployment confidence is strong
- Team collaboration is smooth
- Performance meets requirements
- Bugs are easy to find and fix

Warning signs

- Deployment taking > 1 hour
- Frequent production incidents
- Hard to understand what code does
- Developers afraid to make changes
- Teams constantly blocking each other
- Unable to scale to meet demand

When to consider switching to Microservices

- **Team size** > 15-20 developers working on same codebase
- Clear, stable domain **boundaries identified**
- Need to **scale different components independently**
- Different parts **have different availability requirements**
- Want to use **different technology stacks**

- **Approach**
  - Migrate only critical parts
  - Leave as much logic as possible in the monolith

Common Hybrid Pattern

- Main Monolith
  - User Management
  - Orders
  - Products
  - Admin
- Extracted Services
  - Image Processing Service (CPU-intensive)
  - Search Service (Elasticsearch, different tech)
  - Recommendation Engine (Python/ML)
  - Real-time Chat (WebSocket, needs high availability)

Monolith Migration Antipatterns

- Big Bang Migration
- Microservices Too Early
- Premature Optimization
- Ignoring Data Migration
- No Clear Boundaries
- Some applications should **never** leave the monolith:
  - Internal tools (low traffic, small team)
  - Simple CRUD apps (admin panels, forms)
  - Mature products (no growth, stable features)
  - Small SaaS (< 100k users, small team)
  - E-commerce stores (until huge scale)

Layered Architecture

aka n-tier architecture

Layered Architecture

- One of the most **common and fundamental** architectural patterns
- Like floors in a well-organized building, where each layer has a **specific purpose**
- Emerges from the need to **separate concerns** in software systems
- **Different aspects** of the application should be kept **independent** from each other
- This separation makes the code **more maintainable, testable, and easier to understand**

Layered Architecture Principles

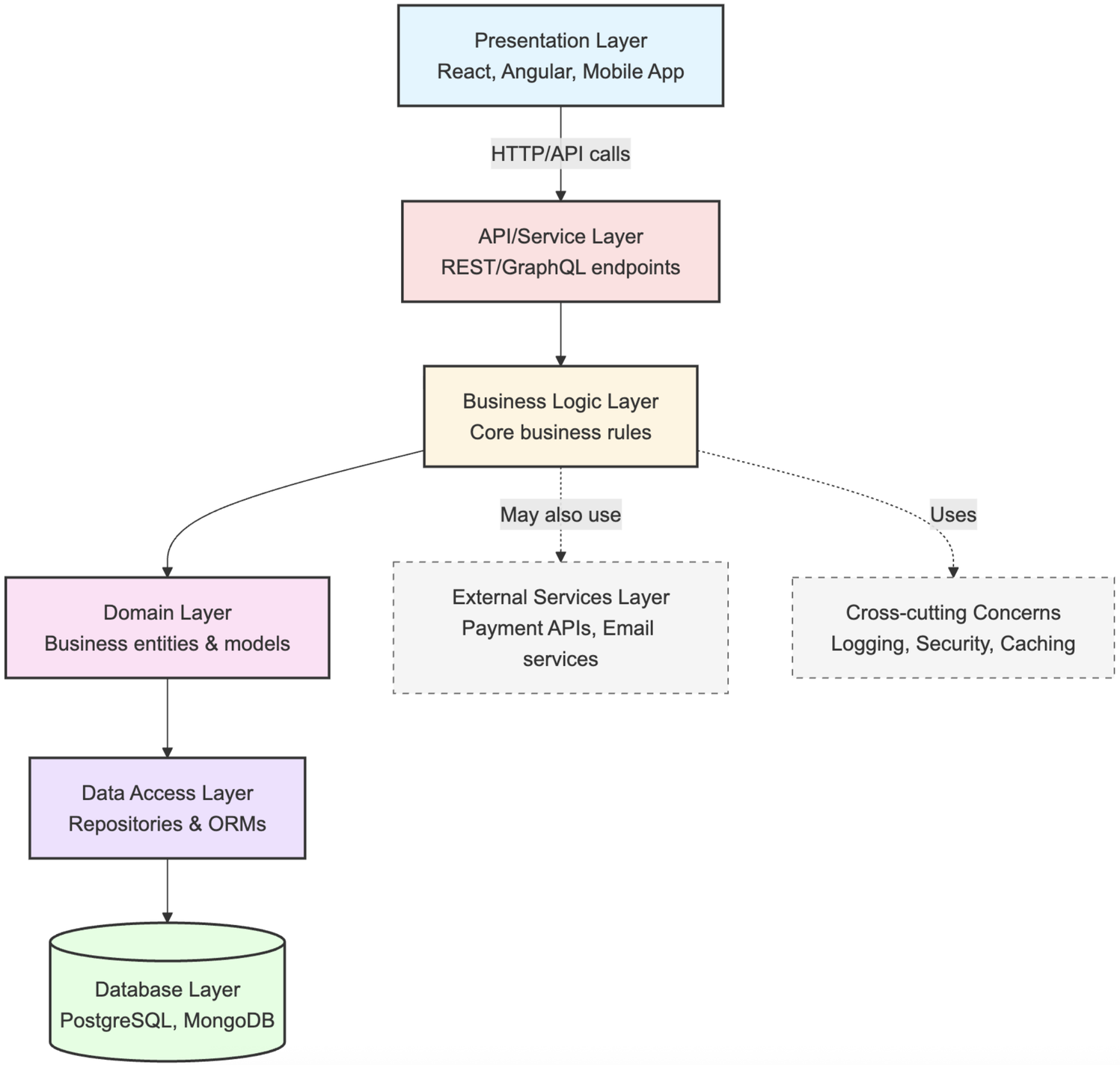
- Separation of Concerns
  - Each layer handles one specific aspect of the application.
- Dependency Rule
  - Layers can only depend on layers below them, never above.
- Abstraction
  - Each layer hides its implementation details from the layers above it.

Common Layers

- 1. Presentation Layer (UI Layer)
  - What users see and interact with
  - Displays data, captures user input, basic validation, etc.
- 2. Business Logic Layer (Application Layer)
  - The "brain" of the application
  - Contains business rules, calculations, and workflows
- 3. Data Access Layer (Persistence Layer)
  - Manages communication with the database
  - Translates business objects to database queries and vice versa
- 4. Database Layer
  - Where data is actually stored

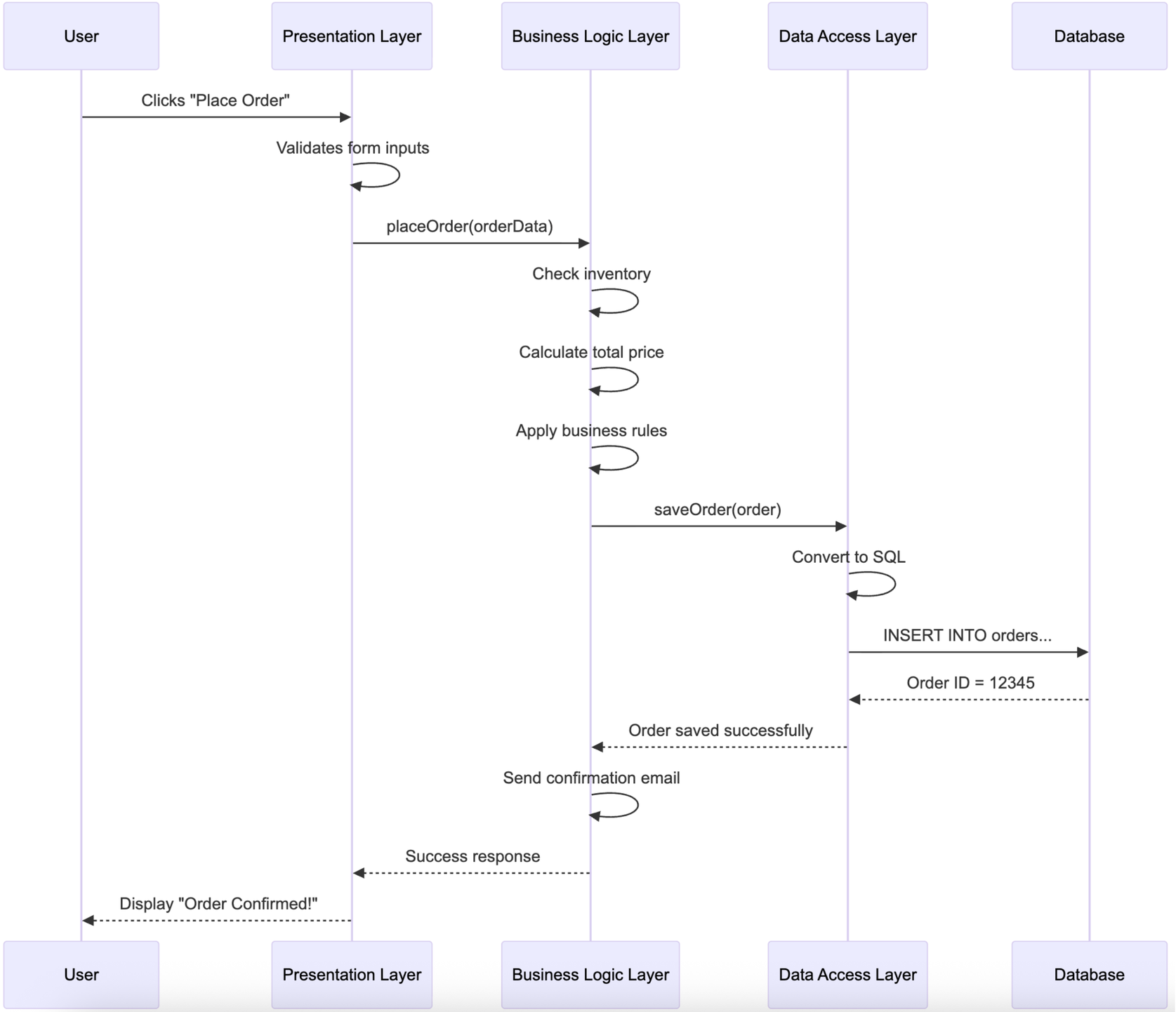
Extended Example

Additional Layers



Layered Architecture

Data Flow Example



UNIVERSITY
SCHOOL OF SCIENCE & TECHNOLOGY

Layered Architecture Advantages

- Easy to Understand and Learn
- Maintainability
- Testability
- Team Scalability
- Reusability

Layered Architecture Disadvantages

- Performance Overhead
- Monolithic Tendency
- Overengineering for Simple Applications
- Database-Centric Design
- Layer Leakage

Hexagonal Architecture

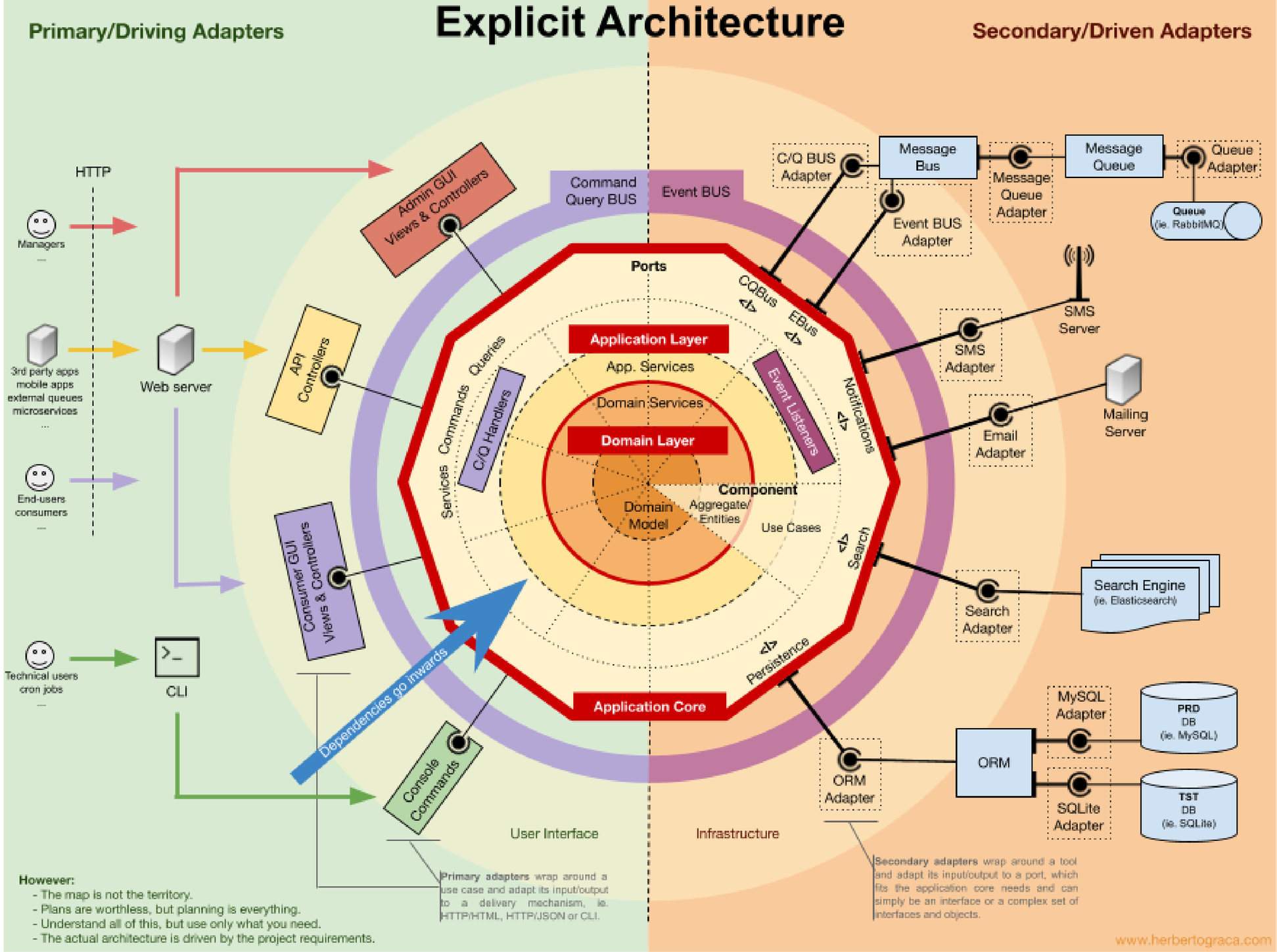
aka ports and adapters architecture

Hexagonal Architecture

- In **layered systems**, **business** rules often **depend** on databases, web frameworks, or external APIs.
  - Problem: **business** logic becomes **contaminated** by **infrastructure** concerns
- **Hexagonal architecture** flips this around:
  - Business logic (the **core**) defines interfaces (**ports**) that describe what it needs.
  - External components (**adapters**) implement these interfaces.

Hexagonal Architecture Example

The hexagon shape is just a visual metaphor suggesting the core business logic has multiple (e.g. 6) sides for different interactions



ie UNIVERSITY
SCHOOL OF SCIENCE & TECHNOLOGY

Main Components: Application Core

- Contains **business logic** and domain models
- Has **zero dependencies** on external frameworks or libraries
- Defines its needs through **interfaces** (**ports**)
- Completely **testable without** any **infrastructure**
- Examples: Order processing rules, user authentication logic, pricing algorithms

Main Components: Interfaces / Ports

- Primary/Driving Ports (Input Ports)
  - Define how the **outside world uses** the application
  - Implemented by the **core**
  - Examples: OrderService, PaymentProcessor

- Secondary/Driven Ports (Output Ports)
  - Define what the **application needs** from the outside world
  - Implemented by **adapters**
  - Examples: DatabasePort, EmailPort, PaymentGatewayPort

Main Components: Adapters

Concrete implementations that connect the real world to the ports

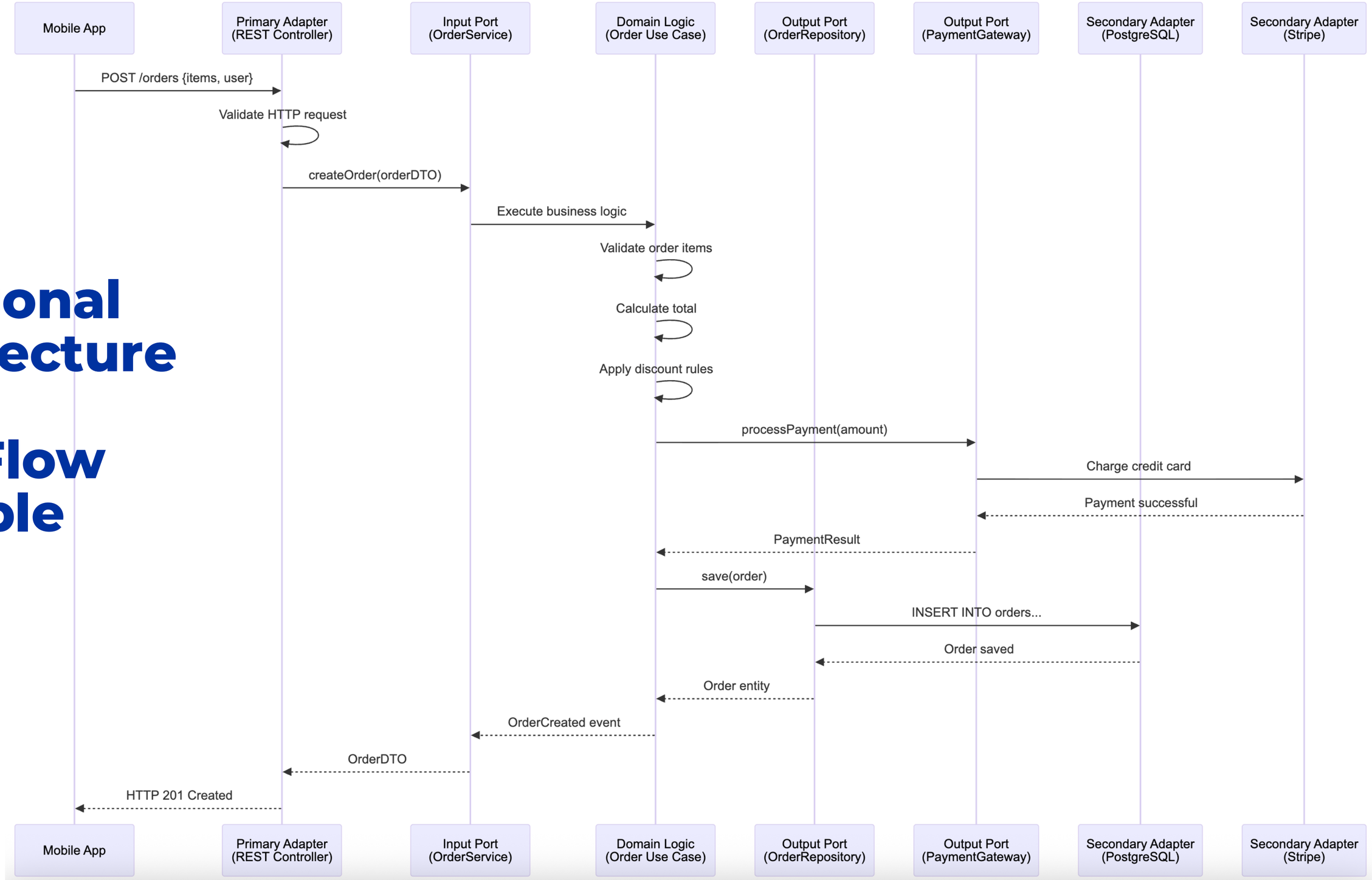
- Primary/Driving Adapters
  - Trigger actions in the application
  - They call the input ports
  - Examples: REST controllers, CLI commands
- Secondary/Driven Adapters
  - Provide services the application needs
  - They implement the output ports
  - Examples: PostgreSQL repository, Stripe payment adapter

Key Principles

- Dependency Inversion
  - The core does not depend on adapters; adapters depend on the core
- Technology Agnostic Core
  - Business logic has no idea where the data is stored or whether it's running in a web application, a CLI tool, or a serverless function
- Testability Without Infrastructure
  - Business logic with simple mock objects
  - No database or external services needed
- Symmetry
  - All external systems are treated equally: they're all outside the hexagon. No distinction between "top" and "bottom" layers.

Hexagonal Architecture

Data Flow Example



UNIVERSITY
SCHOOL OF SCIENCE & TECHNOLOGY

Hexagonal Architecture Advantages

- Complete Business Logic Isolation
- Maximum Testability
- Technology Flexibility
- Clear Boundaries and Contracts
- Parallel Development
- Long-Term Maintainability

Layered Architecture Disadvantages

- Higher Complexity and Learning Curve
- More Code to Write
- Initial Development Slowdown
- Potential for Over-Abstraction
- Performance Overhead
- Harder-To-Understand Data Flow
- Team Alignment Required

Microservices Architecture

ie UNIVERSITY
SCHOOL OF SCIENCE & TECHNOLOGY

Microservices

- Collections of **independent services** that each run in their own process and communicate through lightweight mechanisms.
- Address **scale** issues in monolith architectures, but introduce **complexity**.
- “Micro” does not mean the services are small or lightweight. It means each has a **focused, well-defined purpose.**

Microservices Key Characteristics

- Independent Deployability
  - Each service can be deployed separately
- Business Capability Focus
  - Services organized around business functions, not technical layers
- Decentralized Data
  - Each service manages its own database
- Technology Diversity
  - Different services can use different tech stacks

Microservices Key Characteristics

- Communication via APIs
  - Services interact through network calls (REST, gRPC, messaging)
- Failure Isolation
  - One service failure doesn't bring down the entire system
- Independent Scaling
  - Scale individual services based on their specific needs

Microservices Key Components

API Gateway

- Single entry point for all clients
- Handles authentication, rate limiting, request routing
- Can aggregate responses from multiple services
- Examples: Kong, AWS API Gateway, Nginx

Microservices Key Components

Service Discovery

- Services register themselves when they start
- Other services query to find available instances
- Dynamic routing as services scale up/down
- Examples: Consul, Eureka, etcd

Microservices Key Components

Individual Microservices

- Self-contained applications
- Own their data and business logic
- Expose well-defined APIs
- Can be in different languages/frameworks

Microservices Key Components

Message Broker (for async communication)

- Services publish events when things happen
- Other services subscribe to events they care about
- Decouples services from each other
- Examples: RabbitMQ, Apache Kafka, AWS SQS

Microservices Key Components

Distributed Tracing & Monitoring

- Track requests across multiple services
- Identify performance bottlenecks
- Examples: Jaeger, Zipkin, Datadog

Microservices Key Components

Configuration Management

- Centralized configuration for all services
- Environment-specific settings
- Examples: Consul, Spring Cloud Config

Microservices Communication Patterns

Synchronous Communication

- Traditional request-response pattern where everything happens in real-time:
  - **Client** makes an HTTP request to **Service A**
  - **Service A** needs data from **Service B**, so it makes another HTTP request
  - **Service A** waits (blocks) until **Service B** responds
  - Only then can **Service A** send its response back to the **Client**

Microservices Communication Patterns

Asynchronous Communication

- The event-driven/publish-subscribe pattern:
  - **Producer Service** publishes an event/message to a **Message Queue** or Event Bus
  - The producer does not wait; it immediately continues with other work
  - **Multiple Subscribers** independently consume the message when they're ready
  - Each subscriber processes the message **at their own pace**

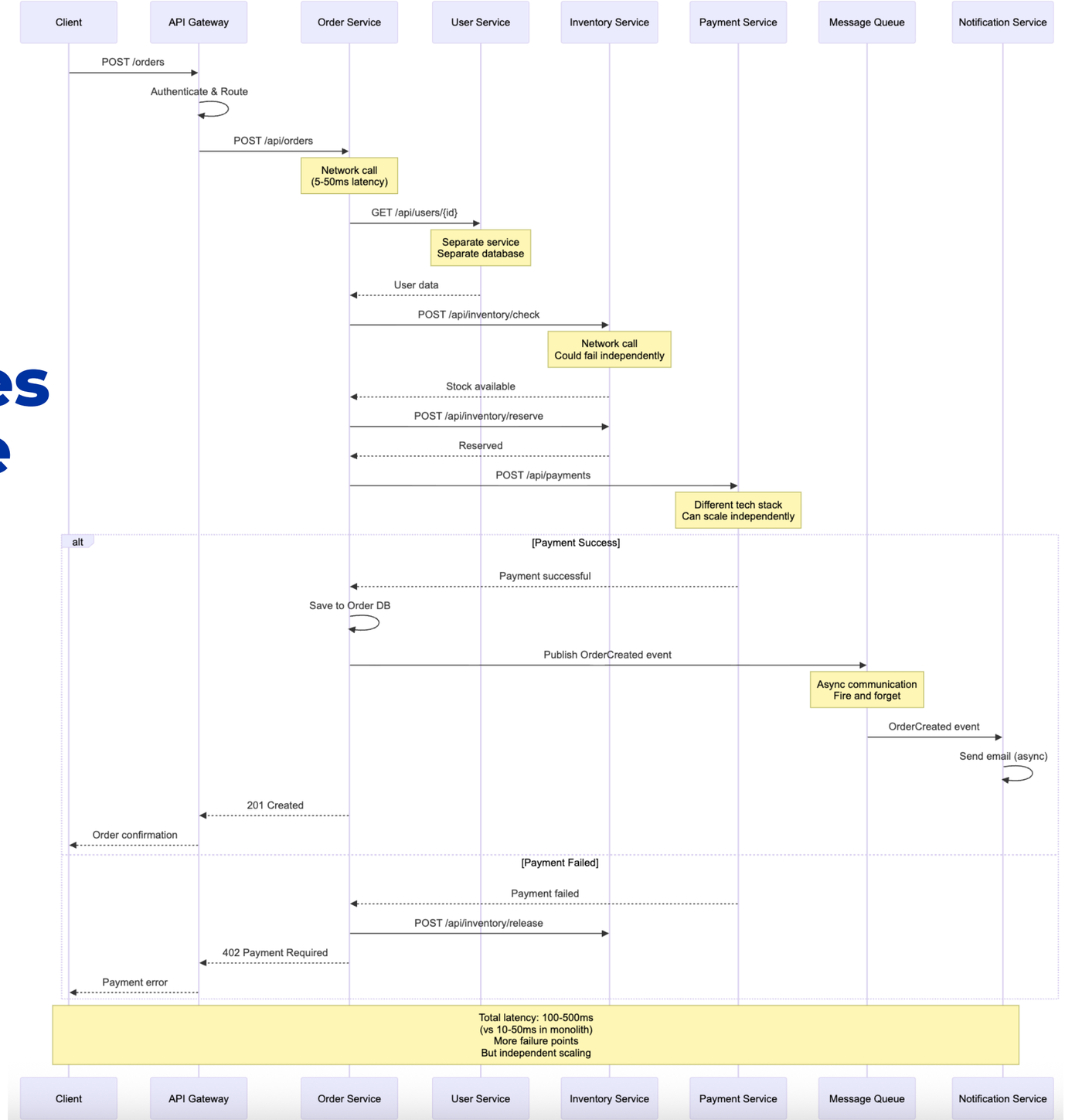
Microservices Communication Patterns

Hybrid: Request-Reply over Messaging

- Both previous approaches for scenarios where you need a response but want messaging benefits:
  - Requester sends a message to a **Request Queue**
  - **Processor Service** picks it up when ready and does potentially long-running work
  - Results go to a **Reply Queue**
  - **Requester** retrieves the response from the reply queue

Microservices Architecture

Data Flow Example



UNIVERSITY
SCHOOL OF SCIENCE & TECHNOLOGY

Microservices Data Management Patterns

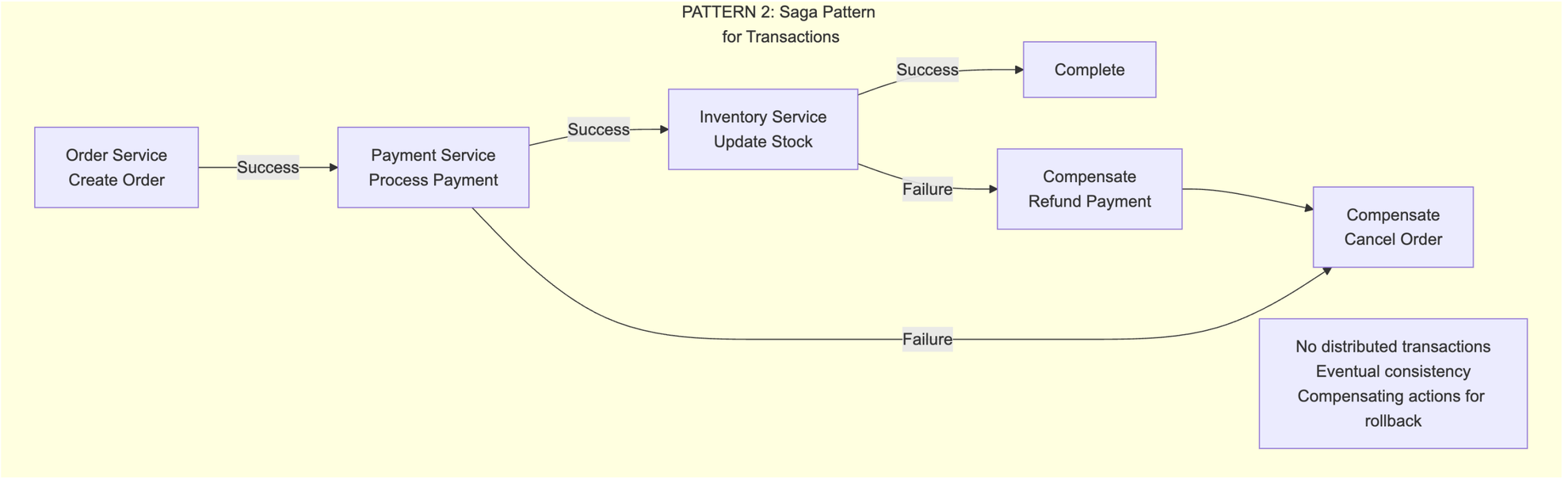
Database Per Service

- Each microservice (Order, User, Payment) has its **own dedicated database**
- Services cannot directly query another service's database
- No shared tables or foreign keys across service boundaries
- This enforces **data ownership** and **loose coupling**.
- **Loss** of convenience of **database joins** and **referential integrity** across services.

Microservices Data Management Patterns

Saga Pattern for Transactions

- Way handle a **transaction that spans multiple services**
- **Sequence of local transactions** instead of a single database transaction
- **Undo** previous steps **on failure**



Microservices Data Management Patterns

Event Sourcing

- Instead of the current state, **store every** event that happened

- The **Event Store** is the source of truth. Different views (Order View, Analytics, Audit) read these events and build their **own representations**

- More **complex** to implement and requires **replaying events** to get current state.

Microservices Data Management Patterns

Command Query Responsibility Segregation (CQRS)

- Separate reads from writes:
- **Write Model**: Handles commands (create, update, delete) using a **normalized** database **optimized for consistency**
- **Read Model**: Handles queries using a **denormalized** database optimized for **fast reads**
- Data flows from Write DB to Read DB **asynchronously**
- Example: an e-commerce site might **write orders** to a **transactional DB**, then sync to **Elasticsearch** for fast **product searches**.

ie UNIVERSITY
SCHOOL OF SCIENCE & TECHNOLOGY

Microservices Architecture Advantages

- Independent Deployability
- Independent Scalability
- Technology Flexibility
- Team Autonomy
- Fault Isolation
- Better for Large Organizations
- Easier to Understand Individually
- Incremental Rewrites
- Optimized for Cloud Native

Microservices Architecture Disadvantages

- Operational Complexity
- Network Latency and Reliability
- Distributed Transactions Are Hard
- Data Management Challenges
- Testing Complexity
- Debugging Nightmares
- Increased Development Time (initially)
- Higher Infrastructure Costs
- Versioning and Compatibility
- Requires Mature Development Practices
- Organizational Challenges
- Security Complexity

When to use Microservices – Planning Phase

Microservices should be the **result of evolution**, not the starting point. The ideal time to adopt microservices is when you have:

- **Evidence** that your monolith is causing specific, measurable problems
- **Expertise** in distributed systems
- **Resources** (people, money, time) to handle the complexity
- **Clear domain boundaries** that are stable and well-understood

When to use Microservices – Planning Phase

When it might make sense

- Greenfield project at a large company
- Team already has extensive microservices experience
- Clear, stable domain model from previous version
- Significant budget
- Regulatory requirements for physical separation
- Proven product with known boundaries

When it definitely does not make sense

- Startup or new product (domain unknown)
- Small team (< 15 developers)
- Limited DevOps experience
- Tight deadline or MVP
- Unclear requirements

Most projects fall into this category

When to use Microservices – Design Phase

Domain Decomposition

- Identify bounded contexts:
  - What are the core business capabilities?
  - Where are the natural boundaries?
  - What data belongs together?

When to use Microservices – Design Phase

Service Boundary Definition

- For each service, define:
  - What data it owns (database tables)
  - What operations it performs (API endpoints)
  - What events it publishes
  - What events it subscribes to
  - Dependencies on other services
  - SLA requirements (latency, availability)

When to use Microservices – Design Phase

Communication Patterns

- Decide for each interaction:
  - Synchronous (REST, gRPC) or Asynchronous (events)
  - Request-response or fire-and-forget
  - Real-time or eventual consistency
  - Timeout and retry strategies

When to use Microservices – Design Phase

Infrastructure

- Must decide on:
  - Container orchestration (Kubernetes, ECS)
  - Service mesh (Istio, Linkerd)
  - API Gateway (Kong, AWS API Gateway)
  - Service discovery (Consul, Eureka)
  - Message broker (Kafka, RabbitMQ)
  - Monitoring (Prometheus, Datadog)
  - Distributed tracing (Jaeger, Zipkin)
  - Log aggregation (ELK, Splunk)
  - CI/CD tooling (Jenkins, GitLab, GitHub Actions)

THANK YOU!

ie UNIVERSITY
SCHOOL OF SCIENCE & TECHNOLOGY