Core design patterns for cloud-native architecture

What is Cloud-Native?

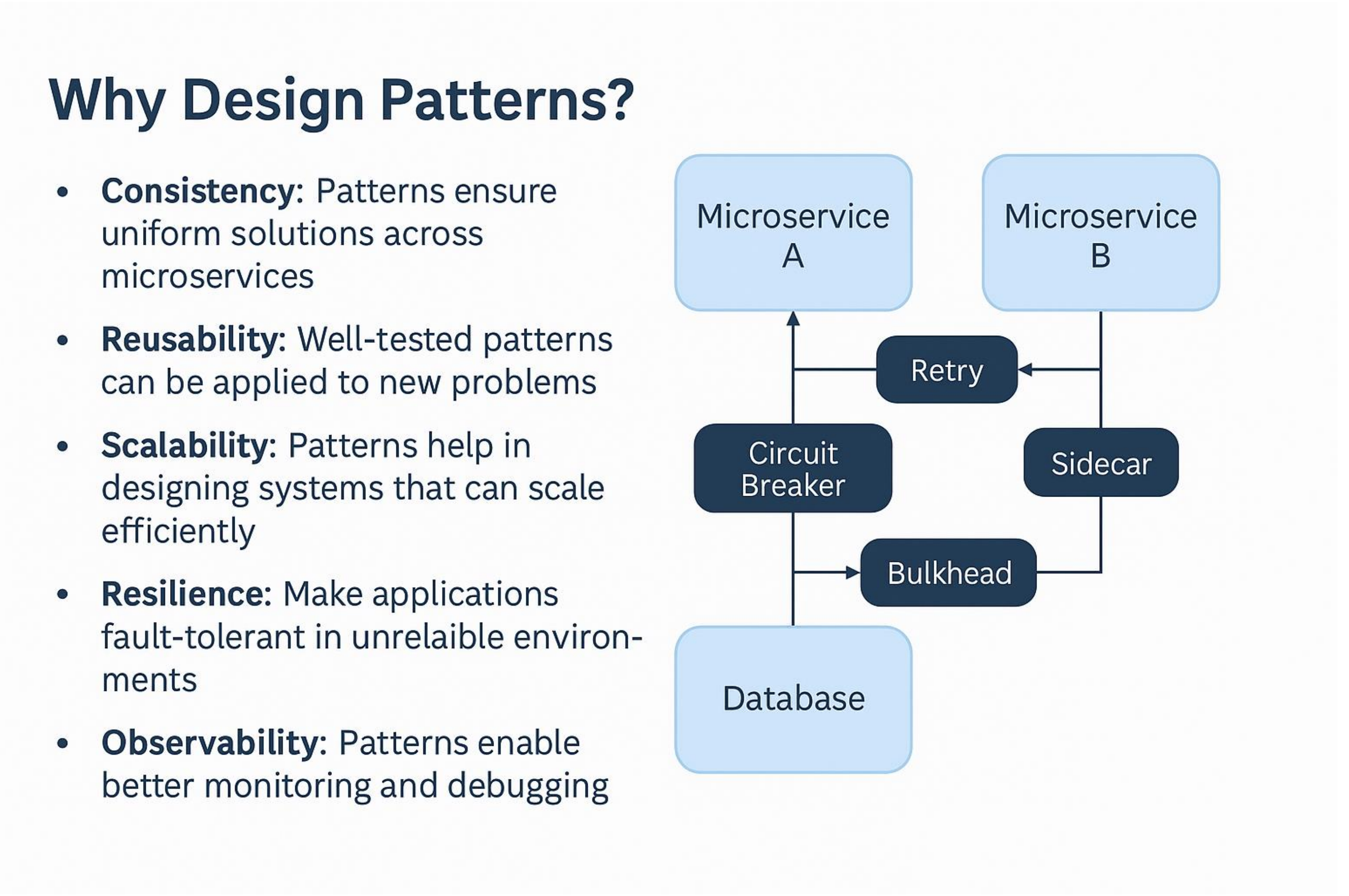
- Cloud-native = apps built to thrive in dynamic, distributed, scalable environments
- Characteristics:
  - Microservices
  - Containers
  - Orchestration
  - CI/CD
  - DevOps and observability

What are Design Patterns?

- Reusable templates for solving common software architecture problems.
- Originally popularized in OOP (Gang of Four), but now essential in cloud-native architecture.
- Provide a shared vocabulary for architects and developers.

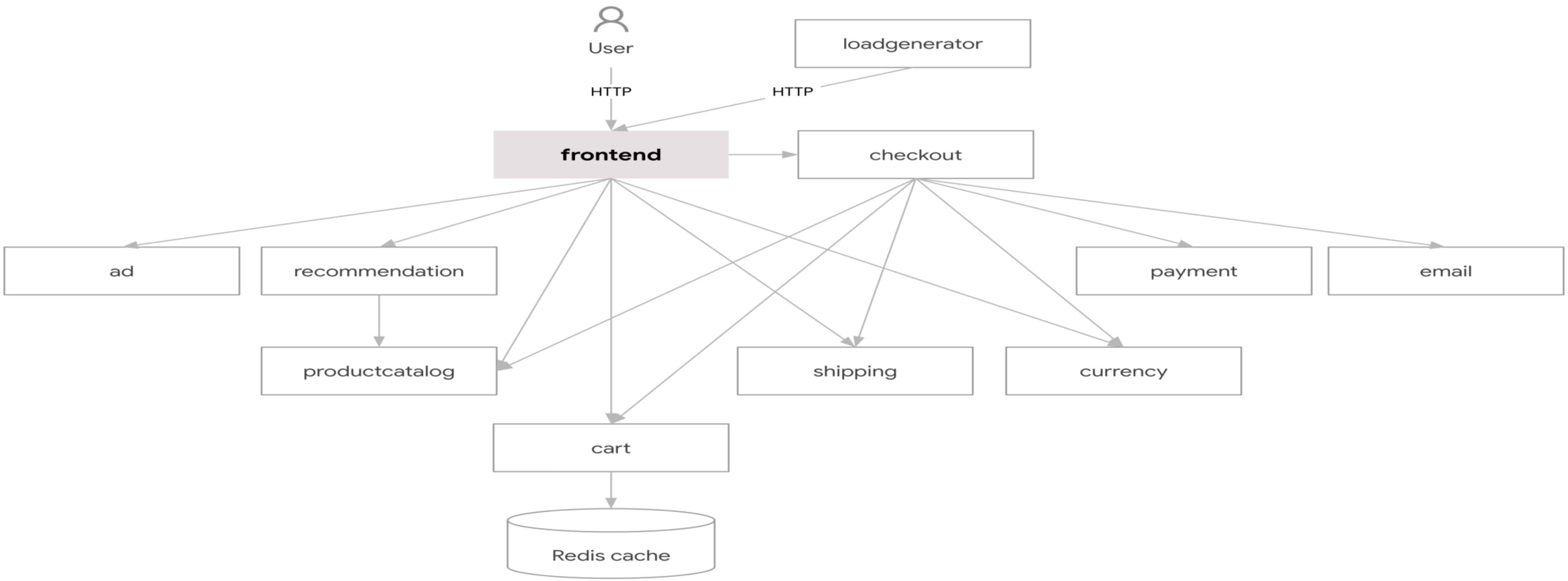
Why Design Patterns?

- Patterns = proven solutions to common architectural problems



Microservice Pattern

- The **Microservice Pattern** breaks down an application into a collection of small, independently deployable services—each responsible for a single business capability.



Core Principles of Microservices

- Single Responsibility: Each service owns a distinct business function.
- Independent Deployment: Services can be updated without redeploying the entire app.
- Decentralized Data: Each service manages its own database schema.
- Polyglot Technology: Services can use different languages, frameworks, or databases.

Microservices Architecture

Microservices architecture structures applications as a collection of small, loosely coupled services.

Each service is responsible for a specific business capability.

Benefits:

- Scalability
- Fault Isolation
- Faster Deployment
- Technology Diversity

Core Patterns in Microservices

- Decomposition Patterns
  - These patterns focus on how to break down an application into services.
- Communication Patterns
  - These define how services interact with each other.
- Data Management Patterns
  - These patterns address how data is handled in a microservices architecture.
- Deployment Patterns
  - These relate to how services are deployed and managed.

DECOMPOSITION

Decomposition Patterns

- By Subdomain
  - This approach follows Domain-Driven Design (DDD) principles.
  - For instance, a logistics company could have subdomains for "Package Tracking" and "Route Optimization.
- By Business Capability
  - Services are structured around business domains.
  - For example, an e-commerce platform might have a "Payment Service," a "User Service," and an "Inventory Service".
- By Strangler Fig Pattern or Strangler Vine Pattern
  - migrating a legacy system to a new system (e.g., microservices architecture) incrementally and safely

Communication Patterns

- **Synchronous Communication**
  - Synchronous communication is a communication pattern where a service sends a request and waits for a response.
- **Asynchronous Communication**
  - Asynchronous communication allows services to interact without having to communicate sequentially, fostering a loosely coupled, event-driven architecture

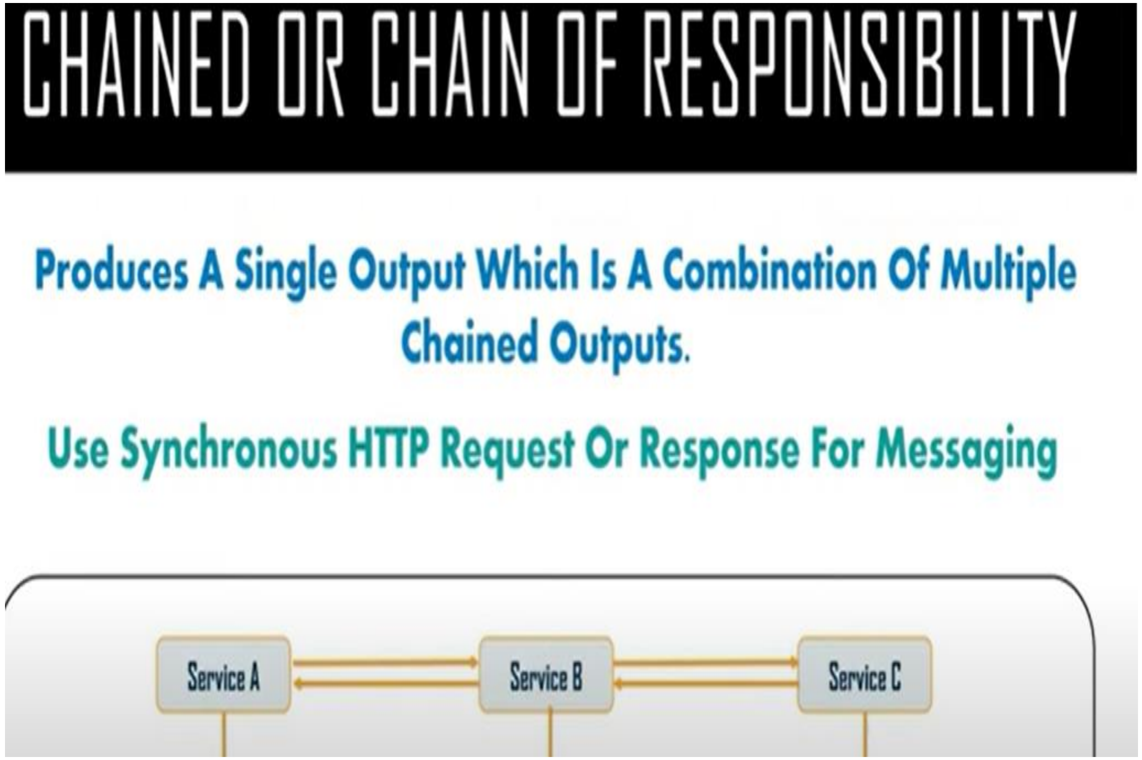
Synchronous Communication

RESTful APIs

- Example Scenario: A "User Service" fetches user details synchronously using a REST API when a "Notification Service" needs to send alerts.

gRPC

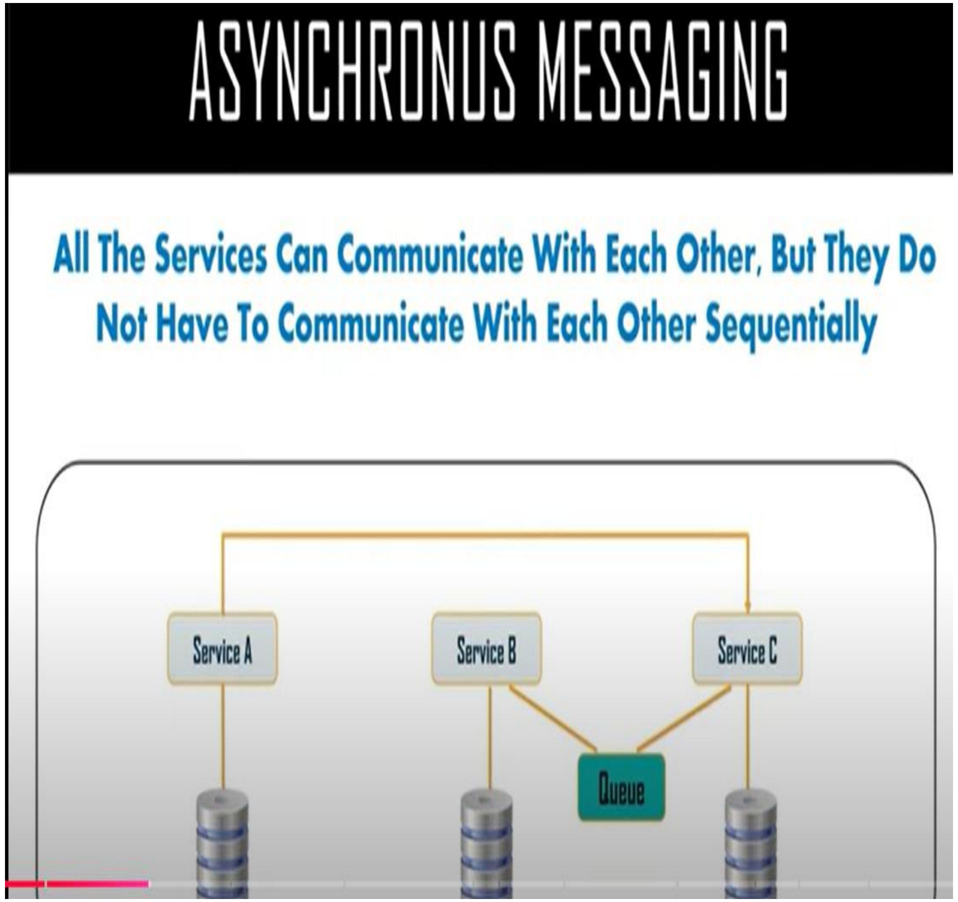
- Example Scenario: A high-performance "Analytics Service" communicates with a "Data Collection Service" using gRPC.



Asynchronous Communication

Event Broker

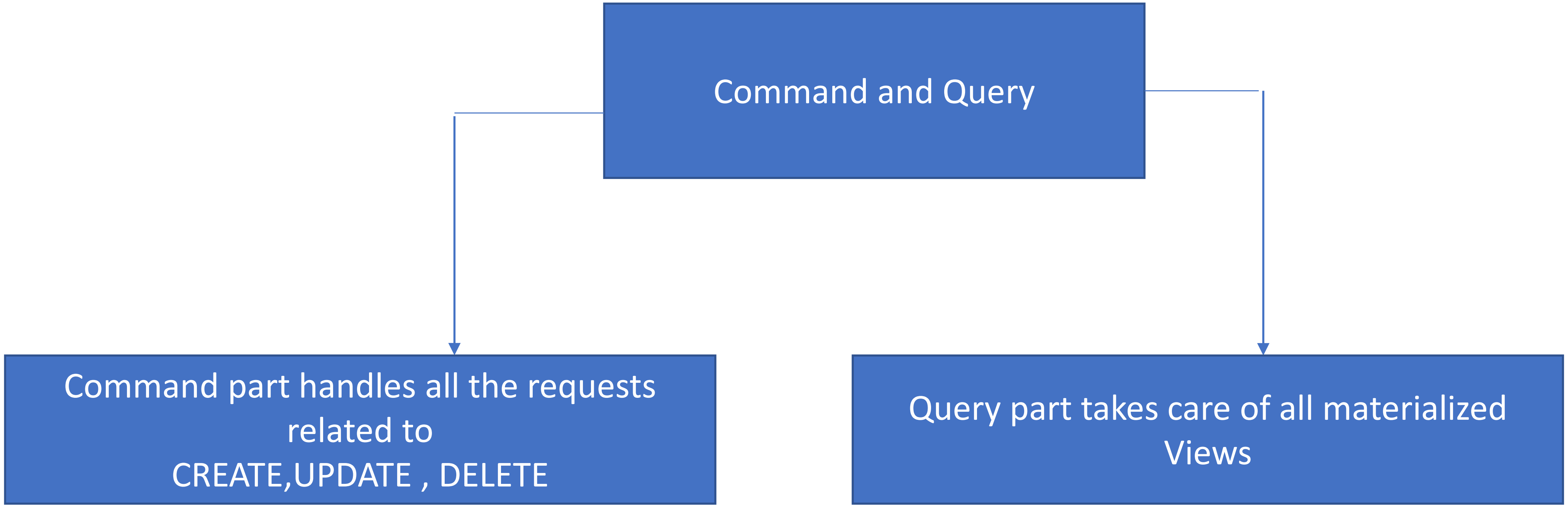
- Event brokers like Kafka or RabbitMQ.
- Example Scenario: An "Order Service" publishes an event when an order is placed, triggering the "Inventory Service" and "Shipping Service" to process the order asynchronously.
- Event-driven architecture for loosely coupled communication



Data Management Patterns

- **Database per Service**
  - Ensures independence and avoids tight coupling.
  - Example Scenario: A "Product Service" uses MongoDB for flexibility, while an "Order Service" uses PostgreSQL for transactional consistency.
- **CQRS (Command Query Responsibility Segregation)**
  - Separates read and write operations for better performance.
  - Example Scenario: A "Banking Service" records all account transactions as events to ensure a reliable audit trail.
- **Event Sourcing:**
  - Captures state changes as events.
  - Example Scenario: A "Customer Support Service" uses separate databases for querying customer interactions and updating issue statuses

CQRS (Command Query Responsibility Segregation)



CQRS (Command Query Responsibility Segregation)

- **The Scenario: An E-Commerce Product Page**
- Consider a popular product page on an e-commerce website. This page has a very high number of reads compared to writes.
- **Writes (Commands):** Updating the product's price, changing its description, or adding a new unit to inventory. These are infrequent.
- **Reads (Queries):** Thousands of users viewing the product page, its details, stock level, and reviews. These are extremely frequent.

Let's assume a typical read-to-write ratio of **100:1**. For every one time a product's details are updated, it is viewed 10,000 times.

Traditional Approach (Without CQRS)

In a traditional system, a single database, often normalized for data integrity, handles both reads and writes.

- **Write Operation Cost (Cw):** A write operation must maintain transactional integrity, potentially updating multiple tables (e.g., Products, Inventory, PriceHistory). This makes it relatively slow.
- Let Tw (Time for one write) = **80 ms**
- **Read Operation Cost (Cr):** A read operation on this normalized database must perform complex JOINs to gather all the data for the page (product details, inventory count, review scores, etc.).
- Let Tr (Time for one read) = **40 ms**

The total computational load (LT) on the single database is the sum of the load from all write and read operations.

$$LT = (1 \times Tw) + (10000 \times Tr)$$
$$LT = (1 \times 80 \text{ ms}) + (10000 \times 40 \text{ ms})$$
$$LT = 80 \text{ ms} + 400000 \text{ ms}$$
$$LT = 400080 \text{ ms}$$

In this model, the thousands of expensive read operations create a massive load and contend for resources with the write operations, potentially slowing the entire system down

CQRS Approach

With CQRS, we have two separate models, each optimized for its task.

Write Model (Command): The database is still normalized for data integrity. The cost of a write operation remains the same.

T_{w-cqrs} = **80 ms**

Read Model (Query): This is the key difference. The read database is a **denormalized** copy of the data, specifically designed for fast reads. All the data needed for the product page is pre-joined and stored in a single document or a wide table. This makes read operations extremely fast.

Let T_{r-cqrs} (Time for one optimized read) = **4 ms**

Now, the load is split between two independent systems.

Load on Write System (Lw):

$$L_w = 1 \times T_{w-cqrs} = 80 \text{ ms}$$

Load on Read System (Lr):

$$L_r = 10000 \times T_{r-cqrs} = 10000 \times 4 \text{ ms} = 40000 \text{ ms}$$

Comparison

Metric	Traditional Model	CQRS Model	Performance Gain	
Time per Read	40 ms	4 ms	10x Faster	
Total Read Load	400,000 ms	40,000 ms	10x Less Load	
Scalability	Must scale a single, complex database for both reads and writes.	Can scale the read and write systems independently. Can deploy 10 read-optimized servers and only 1 write server, matching the load perfectly and saving costs.	Highly Optimized	

Deployment Patterns

- **Single Service per Container:** Simplifies scaling and isolation.
  - Example Scenario: A "Notification Service" runs in its own container, allowing easy scaling during promotional campaigns.
- **Serverless Deployment:** Uses cloud services like AWS Lambda.
  - Example Scenario: An "Image Processing Service" runs as a serverless function to resize and compress images on demand.
- **Orchestrators:** Kubernetes for managing containers.
  - Example Scenario: An "API Gateway" and multiple microservices are managed and scaled using Kubernetes clusters.

Resilience and Fault-Tolerance Patterns

- **Circuit Breaker**
  - Stops repeated failed requests to prevent cascading failures.
  - Example Scenario: A "Payment Gateway Service" triggers a circuit breaker if a third-party payment provider becomes unresponsive.
- **Retry**
  - Retries failed operations after a delay.
  - Example Scenario: A "File Upload Service" retries uploading files to a cloud storage service if there are temporary network issues.

Resilience and Fault-Tolerance Patterns

- **Bulkhead isolation**
  - Allocates resources to prevent overloading critical services.
  - Example Scenario: A "Booking Service" isolates resources for "Flight Booking" and "Hotel Booking" to prevent one from impacting the other.
- **Timeout**
  - Limits how long a service waits for a response from another service.
  - Example Scenario: A "Search Service" sets a timeout for fetching recommendations from an external "Recommendation Service."

Observability and Monitoring Patterns

- Centralized Logging
  - Aggregates logs using tools like ELK Stack.
  - Example Scenario: A centralized logging system captures logs from "Order Service," "Inventory Service," and "Payment Service" for troubleshooting.
- Distributed Tracing
  - Tools like Jaeger or Zipkin to track requests across services.
  - Example Scenario: Tracing a customer order request across "API Gateway," "Order Service," and "Shipping Service."
- Health Checks
  - Monitors the health of individual services.
  - Example Scenario: An "API Gateway" periodically checks the status of backend services to ensure availability.