Database per Service: A Core Microservices Design Pattern

Written By: *Saikat Goswami*

---

1. Introduction

Microservices architecture splits a monolith application into a set of modules, each owning a distinct business capability. But breaking up an application isn't just about code. Data must be decentralized too.

That's where the **Database per Service** pattern comes in. It's a foundational principle of microservices that ensures **each service owns its own data**, with no direct access from other services.

This article explores what this pattern is, why it matters, how to implement it correctly, and the trade-offs you need to consider.

---

2. What Is the "Database per Service" Pattern?

Definition

In this pattern, **each microservice has its own private database** that only it can access directly. No other service is allowed to read or write to that database.

The service is the only interface to the data. External access must go through the service's API.

Each microservice manages its own schema, storage engine, and database logic, ensuring **data encapsulation** and **independence**.

---

3. Why It Matters in Microservices

A. Service Independence

If services share a database, they're tightly coupled. Schema changes or performance issues in one service can impact others. Owning the database lets each service evolve independently.

B. Scalability

With separate databases, each service can scale independently—both in terms of compute and storage.

C. Polyglot Persistence

Different services may benefit from different database technologies (SQL, NoSQL, graph, time-series). This pattern allows each team to choose the best fit.

D. Security and Data Isolation

Data boundaries align with service boundaries. Only the owning service can enforce access rules, reducing accidental data leaks.

---

4. Anatomy of a Database-per-Service System

Consider an e-commerce application split into:

- **User Service** → PostgreSQL
- **Order Service** → MySQL
- **Catalog Service** → MongoDB
- **Shipping Service** → Cassandra

Each service:

- Connects only to its own database
- Exposes APIs for other services to access data
- Can be deployed, versioned, and migrated independently

---

5. Advantages of This Pattern

1. Loose Coupling Between Services

Without shared databases, changes to a schema or table won't ripple through other teams' services.

2. Autonomy for Development Teams

Each team can manage their database as they see fit—indexing, scaling, backups, migration strategy, etc.

3. Improved Availability and Fault Isolation

A failure in one database or service doesn't bring down the entire application.

4. Better Alignment with Domain-Driven Design

The data model closely follows the service's domain logic. Bounded contexts stay intact.

5. Technology Freedom

One service can use PostgreSQL for relational consistency, while another uses MongoDB for document flexibility.

---

6. Trade-Offs and Challenges

Despite its benefits, this pattern brings complexity. Here's what to watch for:

A. Data Duplication

To avoid cross-service DB access, services may copy data between each other (e.g., customer profile info). This leads to duplication and potential staleness.

B. Distributed Transactions

ACID guarantees across multiple services become difficult. Traditional distributed transactions (e.g., two-phase commit) are complex and fragile.

Solution: Use eventual consistency and patterns like Saga or event-driven workflows.

C. Querying Across Services

You can’t run a `JOIN` across services. To answer complex queries (e.g., "Show all orders with customer names"), you need to aggregate via APIs or maintain pre-joined views in a read model.

D. Data Governance and Ownership Confusion

Who owns shared data like customer addresses or user profiles? Clear domain boundaries and data contracts are critical.

E. Increased Operational Overhead

More databases mean more infrastructure to manage, secure, monitor, and back up.

---

7. Patterns That Support Database per Service

1. API Composition

Build a service that aggregates data by calling multiple microservices in parallel.

Use case: Building a UI that needs customer info, order status, and shipping location.

```
[Client] → [Aggregator API] → [User + Order + Shipping services]
```

Pros: Fast, decoupled
Cons: Adds latency, complexity

2. CQRS (Command Query Responsibility Segregation)

Separate the write model (domain services and their DBs) from the read model (precomputed views or projections).

Use case: A dashboard needing rich, joined data that’s hard to compute at runtime.

3. Event Sourcing / Change Data Capture

Use event logs or CDC tools to publish changes between services asynchronously.

Example: User Service emits "UserCreated" event → Order Service updates its local cache.

Pros: Enables eventual consistency
Cons: Adds complexity in event versioning and replay

---

8. Implementing the Pattern Effectively

A. Enforce Boundaries

Ensure no service accesses another's database—even read-only. Use firewall rules, credentials, and code reviews.

B. Define Ownership Clearly

Each piece of data should have one owner. If multiple services need the data, they should fetch or subscribe to updates from the owner.

C. Set Up Monitoring and Backups per DB

Each service should have its own backups, alerts, and performance metrics for their database.

E. Make Data Explicit in APIs

When exposing data from one service to another : add versioning, caching rules, and documentation.

---

9. When to Use (and Avoid) Database per Service

Use When:

- You want strong service boundaries
- Services are independently deployable
- Teams are autonomous and cross-functional
- The system is large and will evolve over time

Avoid When:

- You're building a small app with a few services
- The overhead of multiple databases isn't justified
- All teams work closely and data changes infrequently

---

10. Real-World Examples

Uber

Uses event streams to replicate key data across services, maintaining autonomy and eventual consistency.

Amazon

Every service owns its data. Order history, cart data, user profiles—each lives in its own database. This isolation allows each team to deploy daily without fear of breaking someone else's system.

Netflix

Microservices at Netflix each own their state. For example, the recommendations engine may store its data in a graph database, while billing uses a traditional RDBMS.

---

11. Future Trends and Technologies

- **Distributed SQL databases** (e.g., CockroachDB) offer a hybrid model: logical separation, shared infra
- **Change Data Capture tools** (e.g., Debezium) simplify syncing between services
- **Serverless databases** reduce operational overhead of managing many DBs
- **Data mesh** concepts extend the idea of data ownership and domain alignment to analytics platforms

---

Key Takeaways:

- **Don't share databases across services.** Share data via APIs or events.
- **Expect duplication** and design for it.
- **Choose the right tools** for versioning, syncing, and querying.
- **Make ownership explicit**—every data field should have a responsible service.
