How would you handle database design in a microservices architecture? Suppose you have a monolithic ecommerce system , how you can convert to microservice based architecture , what DB you prefer and why ? list the strategies to enable fast access of data from DB. How will you integrate your new system ? Design the system architecture and data flow.

Migrating a monolithic e-commerce system to microservices requires a fundamental shift in data ownership. The core principle is **Database Per Service**, meaning no two services share a database table.[1]

1. Database Strategy: "Database Per Service"

In the monolith, you likely had one giant SQL database (e.g., Oracle/MySQL) with `Users`, `Orders`, and `Products` tables linked by foreign keys.
In microservices, you must break this link.

Why?

- **Decoupling:** If the `Order` service needs to change its schema, it won't break the `User` service.
- **Scaling:** You can scale the `Product Catalog` database (read-heavy) differently from the `Checkout` database (write-heavy).[1]

2. Preferred Databases (Polyglot Persistence)

Instead of forcing one DB to do everything, choose the best tool for each service.[2] [3]

Microservice	Preferred DB	Why?
Product Catalog	**NoSQL (MongoDB / Cassandra)**	Products have flexible schemas (e.g., a Shirt has size/color; a Laptop has RAM/CPU). NoSQL handles this variety better than fixed SQL columns.
Orders & Payments	**RDBMS (PostgreSQL / MySQL)**	Requires strict **ACID** properties. Financial transactions must be accurate (no money lost).
User Session / Cart	**Redis (In-Memory)**	Needs extremely fast read/write for temporary data (e.g., shopping cart items) that expires quickly.

Microservice	Preferred DB	Why?
Search	**Elasticsearch**	Specialized for full-text search (e.g., "blue running shoes") which standard SQL does poorly.

3. Strategies for Fast Data Access

Since you can't do `JOIN` queries across databases anymore, use these strategies:

- **CQRS (Command Query Responsibility Segregation):**
  - Separate the "Write" model from the "Read" model.
  - *Example:* When a user places an order, write to the optimized `Order DB`. Then, asynchronously update a separate, flat `Order History Read DB` optimized for fast viewing.[4] [5]
- **Data Replication / Caching:**
  - If `Order Service` needs user data (e.g., address), don't call `User Service` every time. Replicate critical data (like `user_id`, `address`) into the `Order DB` when the user is created, or cache it in Redis.
- **Materialized Views:**
  - Pre-calculate complex data (e.g., "Total Sales per Month") and store it as a simple table, so you don't have to compute it on the fly.

4. Integration & System Architecture (The "Strangler Fig" Pattern)

Don't rewrite everything at once. Use the **Strangler Fig Pattern** to migrate piece by piece.[6] [7]

Architecture Design & Data Flow

1. **API Gateway (The Entry Point):** All client requests hit the Gateway (e.g., Kong, Zuul).
2. **The "Strangler" Facade:**
   - The Gateway checks the URL.
   - New features (e.g., `/api/v2/cart`) go to the new **Cart Microservice**.
   - Legacy features (e.g., `/api/v1/orders`) go to the **Old Monolith**.
3. **Migration Steps:**
   - **Identify a Domain:** Pick a low-risk module like "Product Reviews".
   - **Build Service:** Create the `Review Service` with its own MongoDB.
   - **Sync Data:** Use a script to copy existing reviews from the Monolith DB to MongoDB.
   - **Switch Traffic:** Point the API Gateway for `/reviews` to the new service.
   - **Repeat:** Move to the next module (e.g., Catalog, then User, finally Orders).

Data Flow Example (Placing an Order)

1. **Frontend** sends `POST /order` to **API Gateway**.
2. **Order Service** (PostgreSQL) creates a "Pending" order.
3. **Order Service** publishes an event: `OrderCreated` to **Kafka**.
4. **Inventory Service** listens to `OrderCreated`, checks stock in its own DB, and reserves items.
5. **Payment Service** listens, processes payment, and publishes `PaymentSuccess`.
6. **Order Service** listens to `PaymentSuccess` and updates status to "Confirmed".
   (This is the SAGA pattern discussed earlier).

⁂

1. https://www.geeksforgeeks.org/system-design/database-per-service-pattern-for-microservices/
2. https://www.arxiv.org/pdf/2509.08014.pdf
3. https://www.infoq.com/articles/polyglot-persistence-microservices/
4. https://appmaster.io/blog/cqrs-event-sourcing-microservices
5. https://dzone.com/articles/microservices-with-cqrs-and-event-sourcing
6. https://www.youtube.com/watch?v=gwu3lVUvlqQ
7. https://vfunction.com/blog/fig-pattern-the-solution-to-your-mono-to-microservices-modernization/
8. https://www.youtube.com/watch?v=DKQLhy9bgdk
9. https://www.linkedin.com/posts/gaurav-sharma-45800853_microservices-databaseperservice-designpatterns-activity-7375743778353192960-AzIm
10. https://www.milanjovanovic.tech/blog/breaking-it-down-how-to-migrate-your-modular-monolith-to-microservices
11. https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/database-per-service.html
12. https://circleci.com/blog/monolith-to-microservices-migration-strategies/
13. https://www.youtube.com/watch?v=Ia2q1vFA5q0
14. https://microservices.io/patterns/data/database-per-service.html
15. https://www.reddit.com/r/ExperiencedDevs/comments/qdzjoi/unpopular_opinion_database_per_service_is_just/