

Course: Domain Driven Design & Microservices for Architects

Section: Microservices Data Patterns

<http://acloudfan.com/>

Pragmatic Paths Inc © 2021

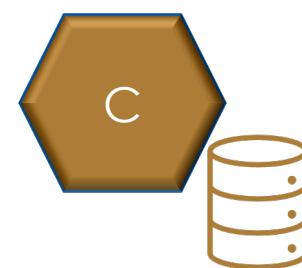
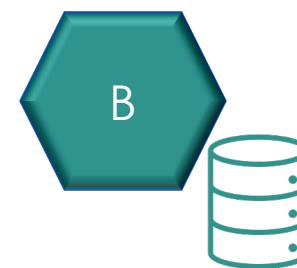
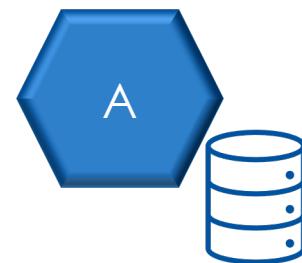
Contact: raj@acloudfan.com

Discount Link to course:

<https://www.udemy.com/course/domain-driven-design-and-microservices/?referralCode=C5DCD3C4CC0F0298EC1A>

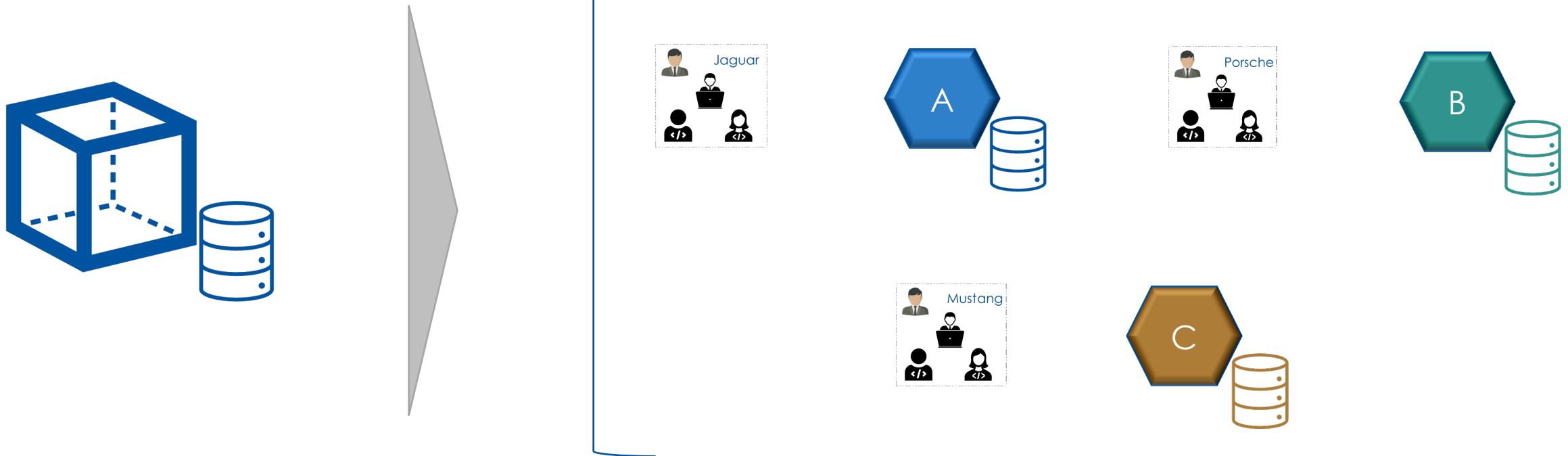
Microservices Data Patterns

Databases are an essential part of ALL applications.

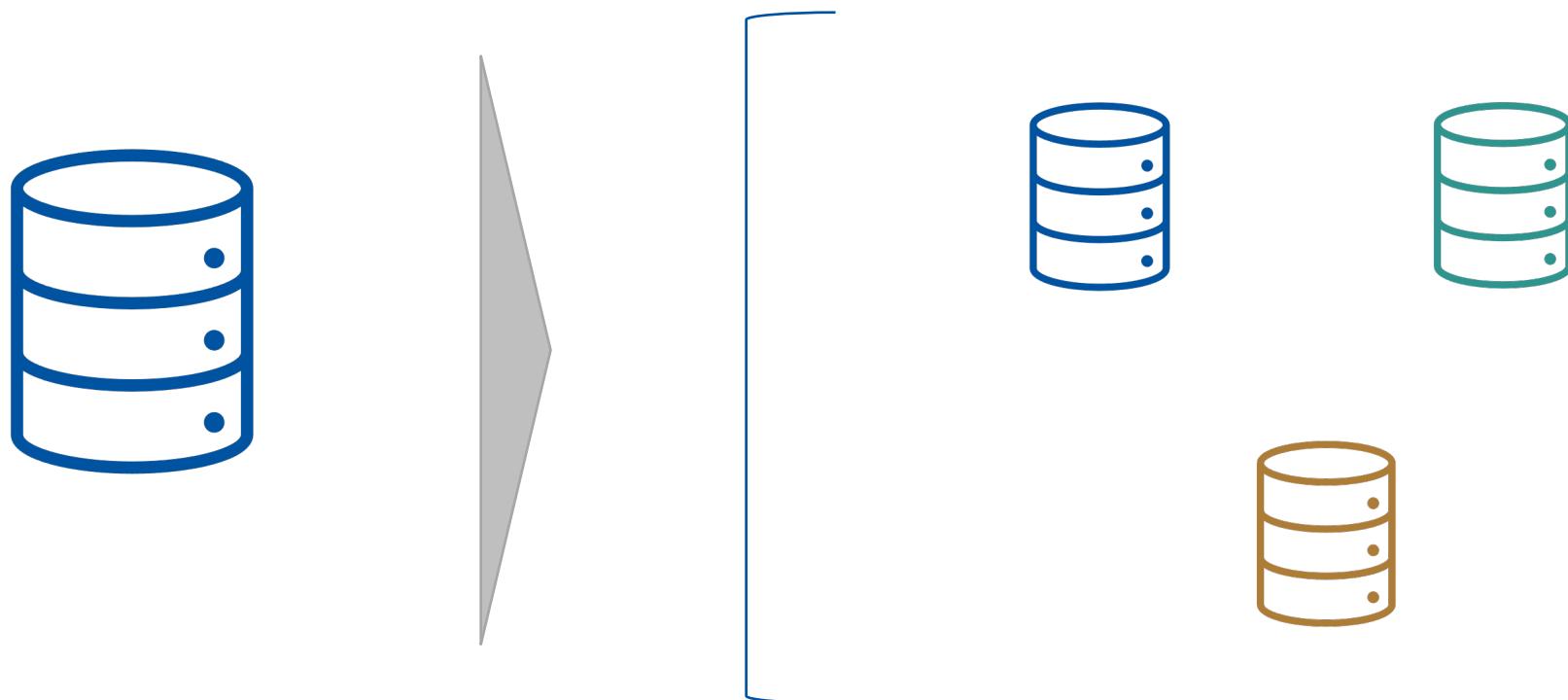


Microservices Data Patterns

Monolithic app to Microservices



Breaking the Database is a **CHALLENGE**



Data Patterns

- Used for designing the new Microservices
- Conversion of Monolithic apps to Microservices

- 
- 1 Shared Database Pattern
 - 2 Separate Database Pattern
 - 3 Strangler Pattern
 - 4 Converting monolith to microservices
 - 5 Options for breaking shared DB into multiple DB instances

Persistence in Shared Databases

Data storage in legacy applications



- 1 Shared Database pattern
- 2 Pros of shared database
- 3 Cons of shared database

Data Storage

Persistence for application data

- Managed in raw files
- Databases

ORACLE



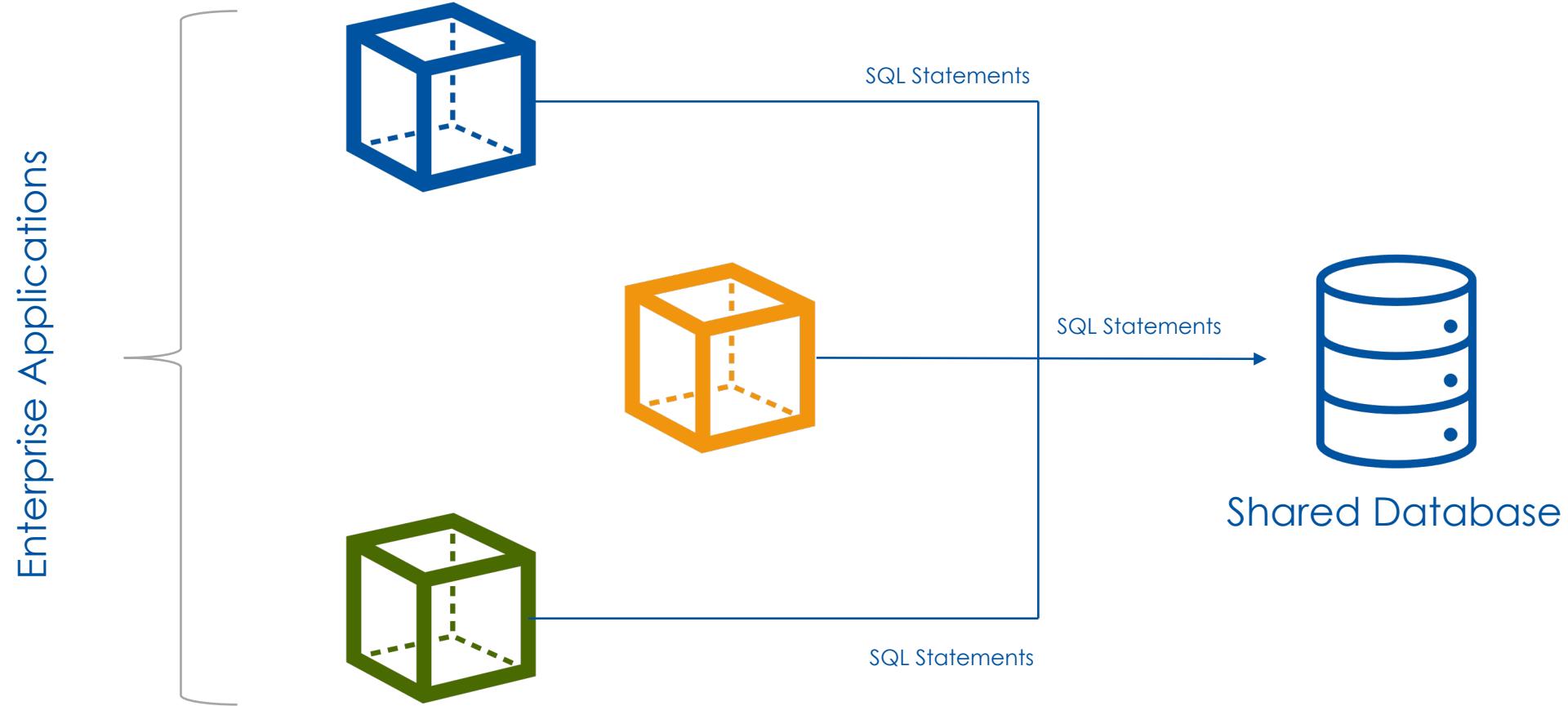
mongoDB®



Relational DBs commonly used in legacy apps for all types of data !!

Use of RDBMS in Legacy Apps

Shared Database(s) for multiple applications



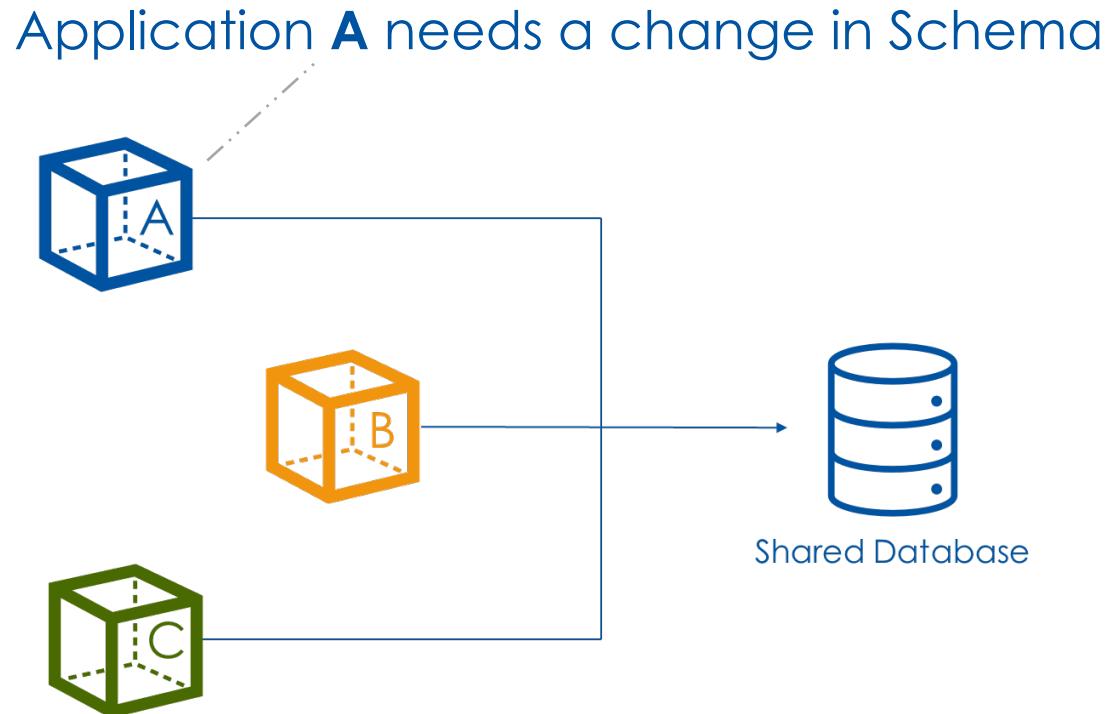
Shared Database - Good things

- 1 Simplified data management
- 2 Cost savings on Database (licensing, servers ...)
- 3 Centralized database administration

Challenges with Shared Database

1

DB Changes need to be managed carefully



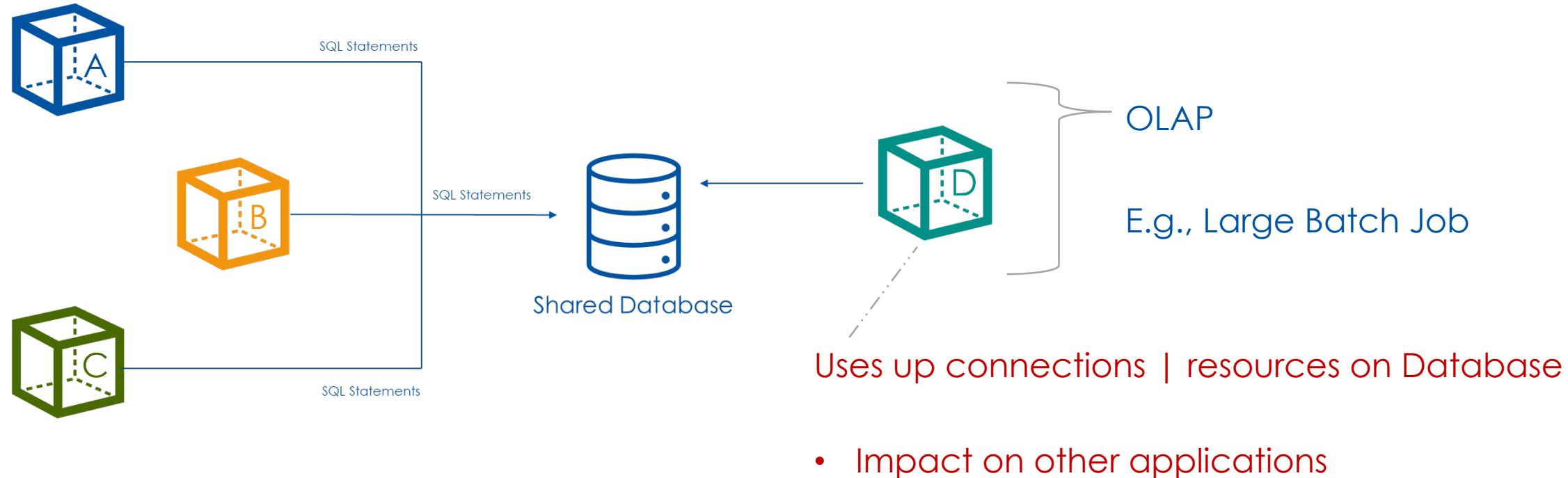
- High Cost
- High Risk
- Longer time to value

All application owners will need to assess the impact on their App !!!

Challenges with Shared Database

2

One application may negatively impact all applications

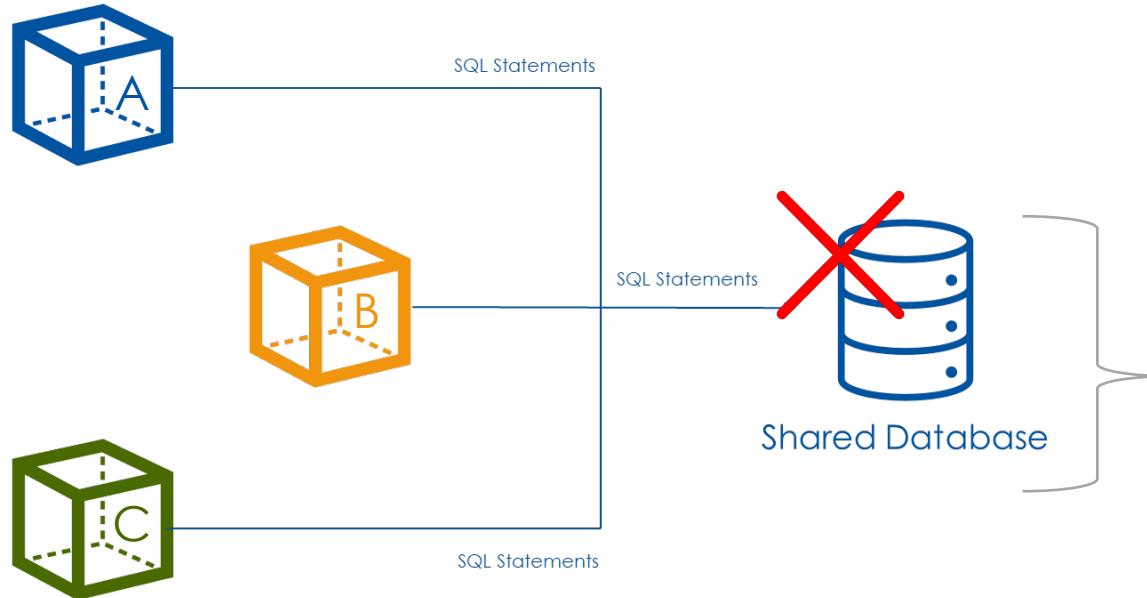


One App can use too much resource and impact all other Apps !!

Challenges with Shared Database

3

Single Point of Failure



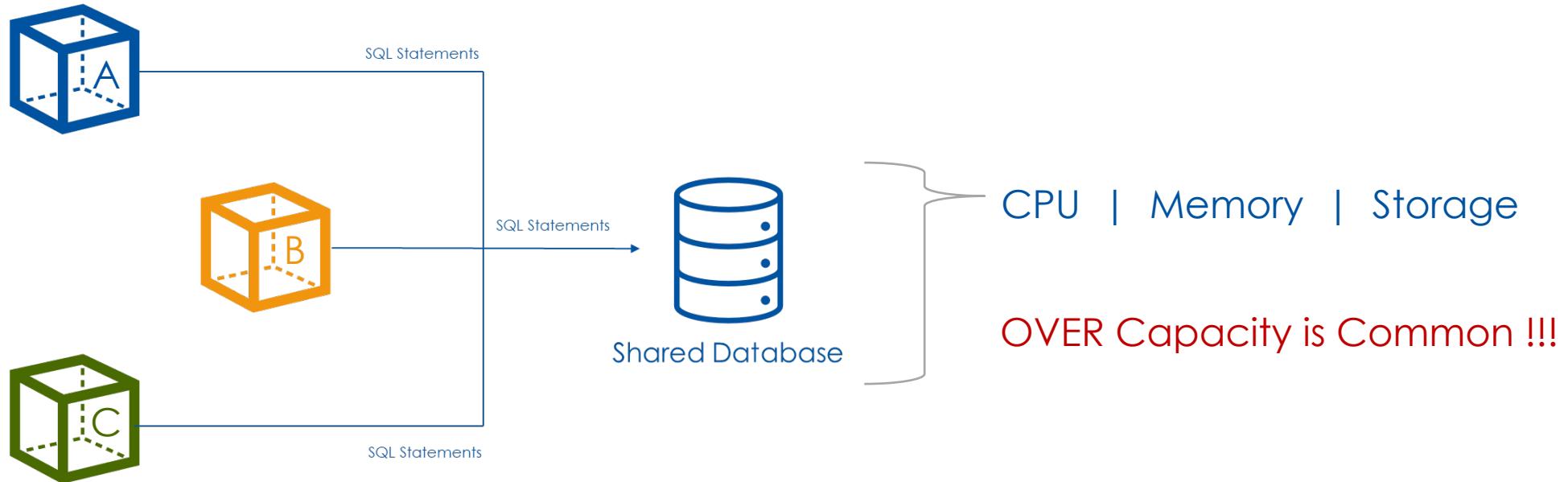
- Complex HA solutions → High cost

All Applications dependent on the DB will be unavailable !!!

Challenges with Shared Database

4

Capacity planning for the Database



ALL teams MUST provide estimates on their usage !!!

Challenges with Shared Database

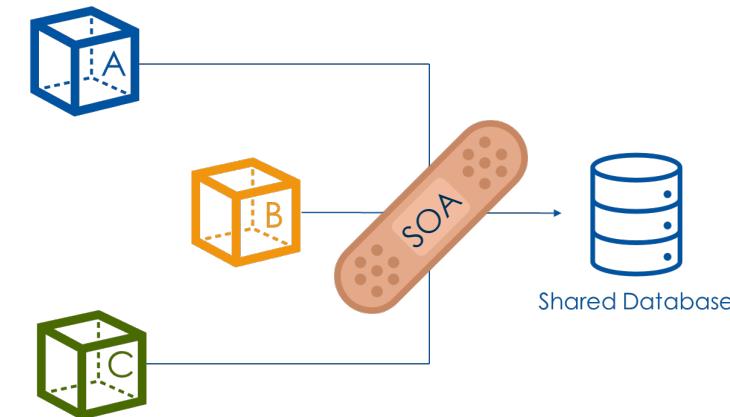
- 
- 6 Same Database types for all applications

Application teams does not have a choice of DB type !!!

Monolithic applications

Shared Database(s) is an anti-pattern BUT

- Many enterprises are still dealing with such applications
- Service Oriented Architecture | API for rescue

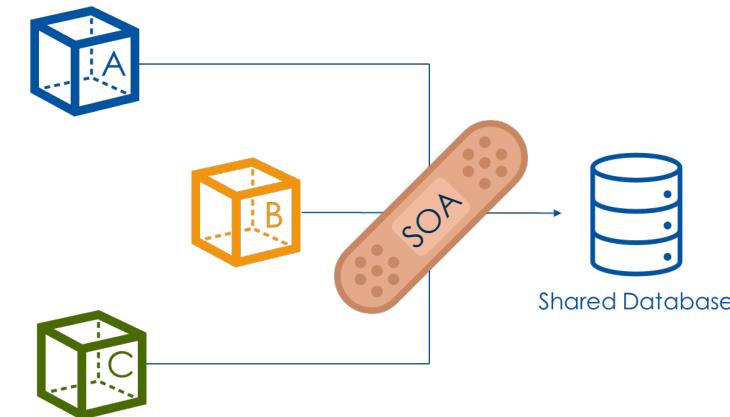




Quick Review

Shared Database(s) is an anti-pattern BUT

- Many enterprises are still dealing with such applications
- Service Oriented Architecture (SOA) | API for rescue



Service Oriented Architecture (SOA)

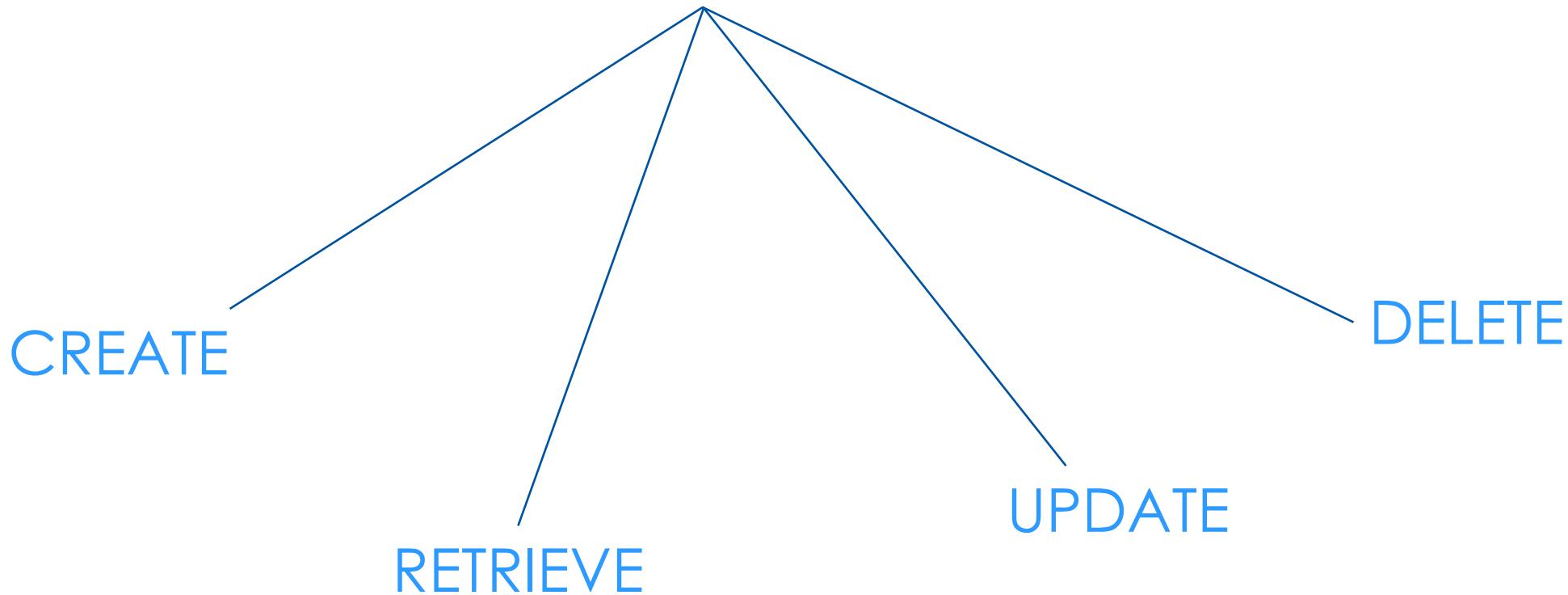
Insulated applications from underlying database(s)



- 1 Introduction to SOA
- 2 SOA & Shared Database pattern

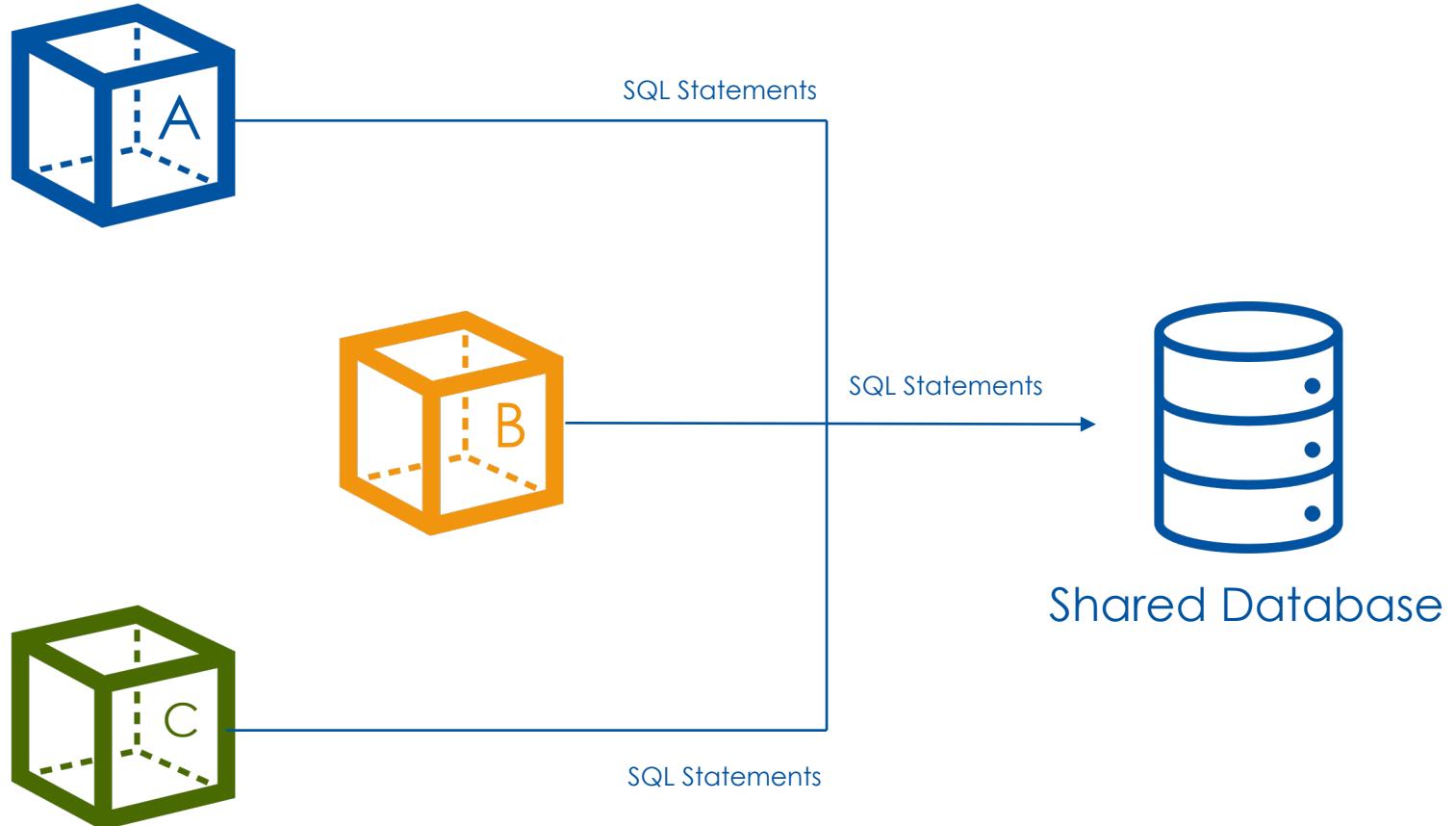
Service Oriented Architecture

Data exposed by way of CRUD services | API

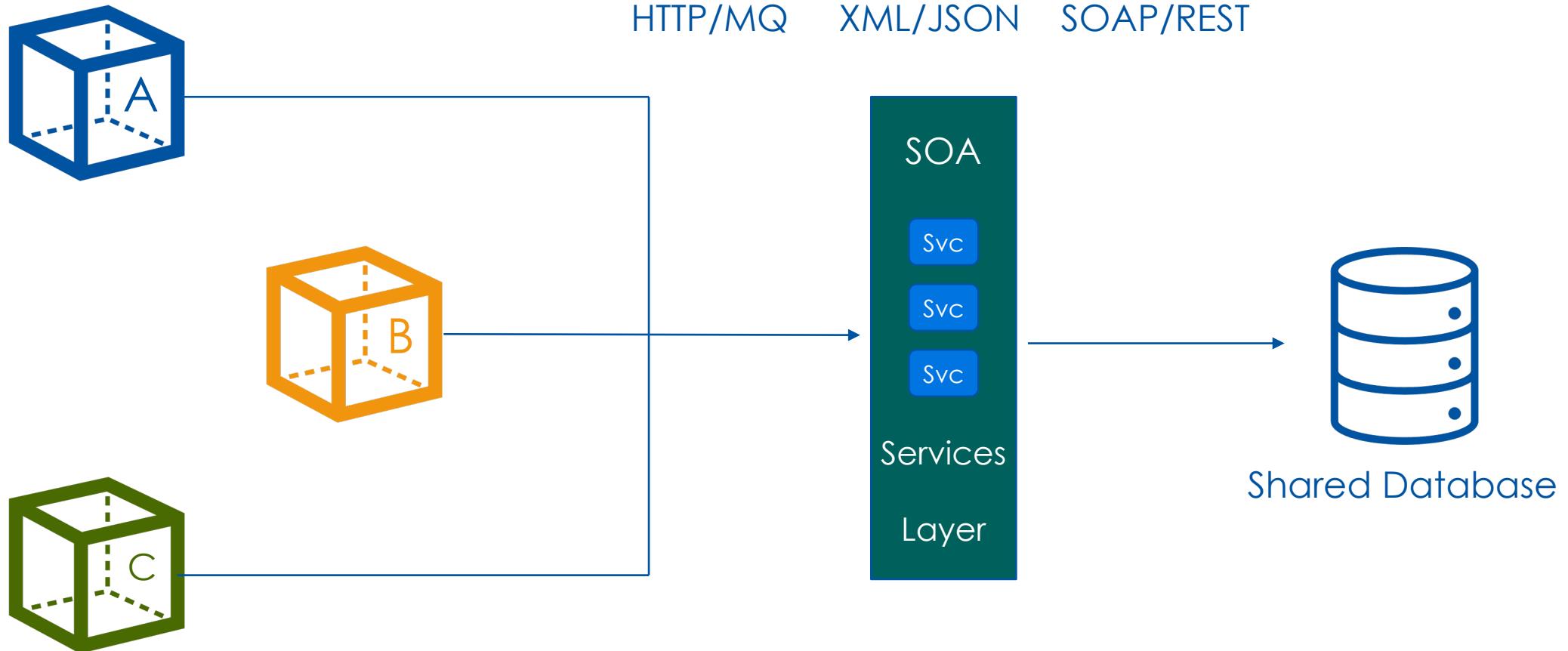


Hides the Database structure behind services !!!

SOA Layer placed between Apps & DB



SOA Layer placed between Apps & DB



Services provided CRUD and Higher-level business operations

SOA addressed some Challenges

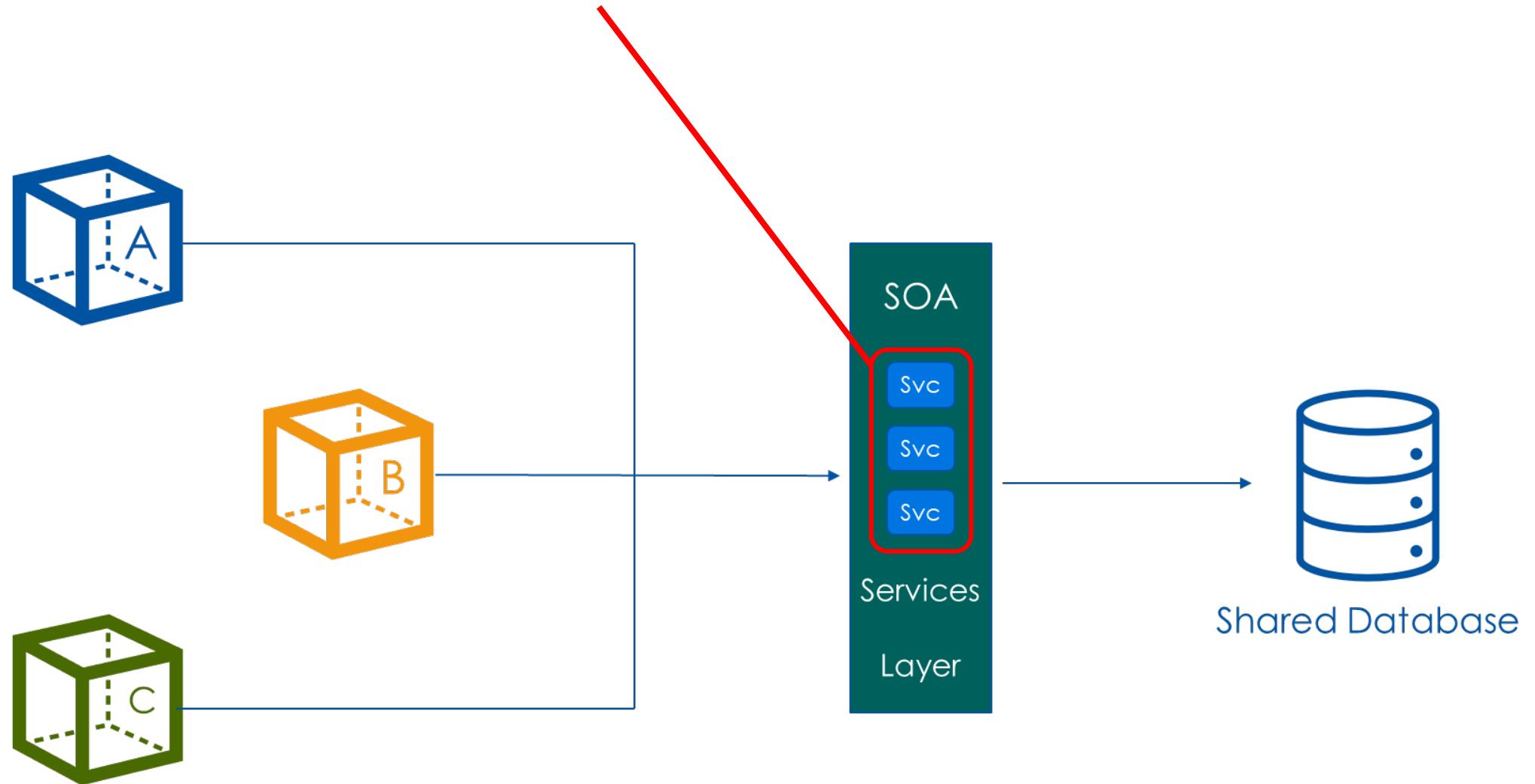
- 1 ► Hides the structure of DB i.e., no more SQL statements
- 2 ► Services were designed to be re-usable
- 3 ► Changes to DB become manageable

Doesn't address *many* of the shared DB challenges !!

Misconception



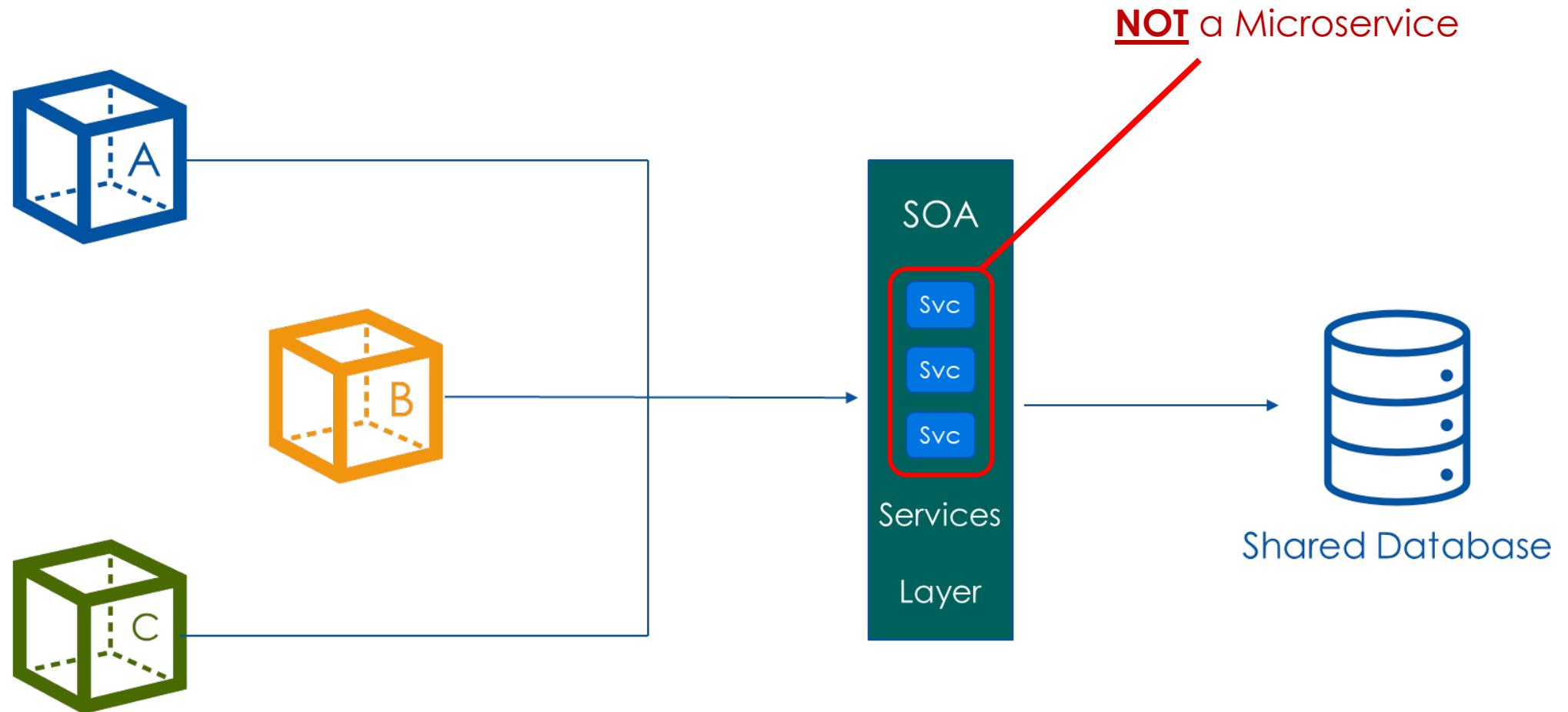
A small SOA Service is a Microservice



Misconception



A small SOA Service is a Microservice





Quick Review

SOA services insulated the application from Database changes

- BUT did not address other challenges of Shared Database(s)

Separate Database Pattern

Recommended for Greenfield Microservices Initiative



- 1 Greenfield versus Brownfield
- 2 Separate Database Pattern
- 3 Advantages of separate DB

Microservices Projects

Greenfield



New application with no constraints from technical debt perspective

Brownfield



Existing monolithic app to be converted to Microservices architecture

Microservices : Recommendation

"Separate DB" recommended for Microservices

Brownfield



Is it even possible to break the existing DB?

Is it worth the effort i.e., do a Cost-Benefit Analysis?

What will be the Business benefits?

Microservices : Recommendation

"Separate DB" recommended for Microservices



Are the teams skilled for managing their own DB?

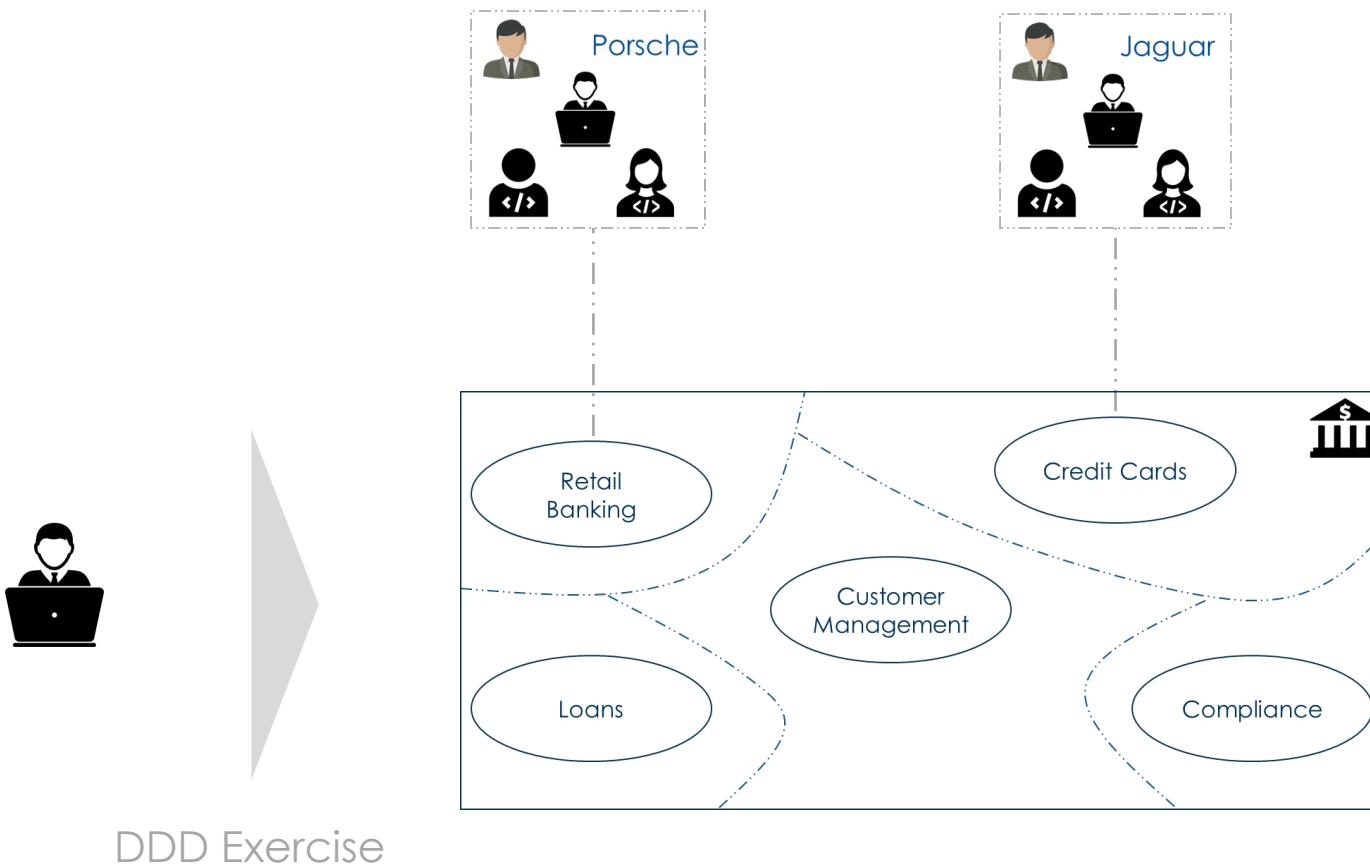
Does the organization have tools to manage multiple database?

Are the teams ready to manage design complexities?



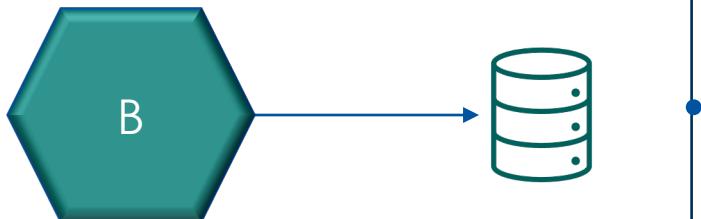
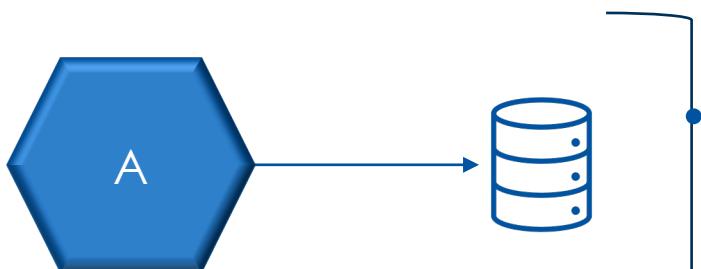
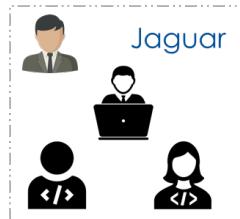
Microservices from ground up

Microservices teams assigned the Bounded Context





Microservices from ground up

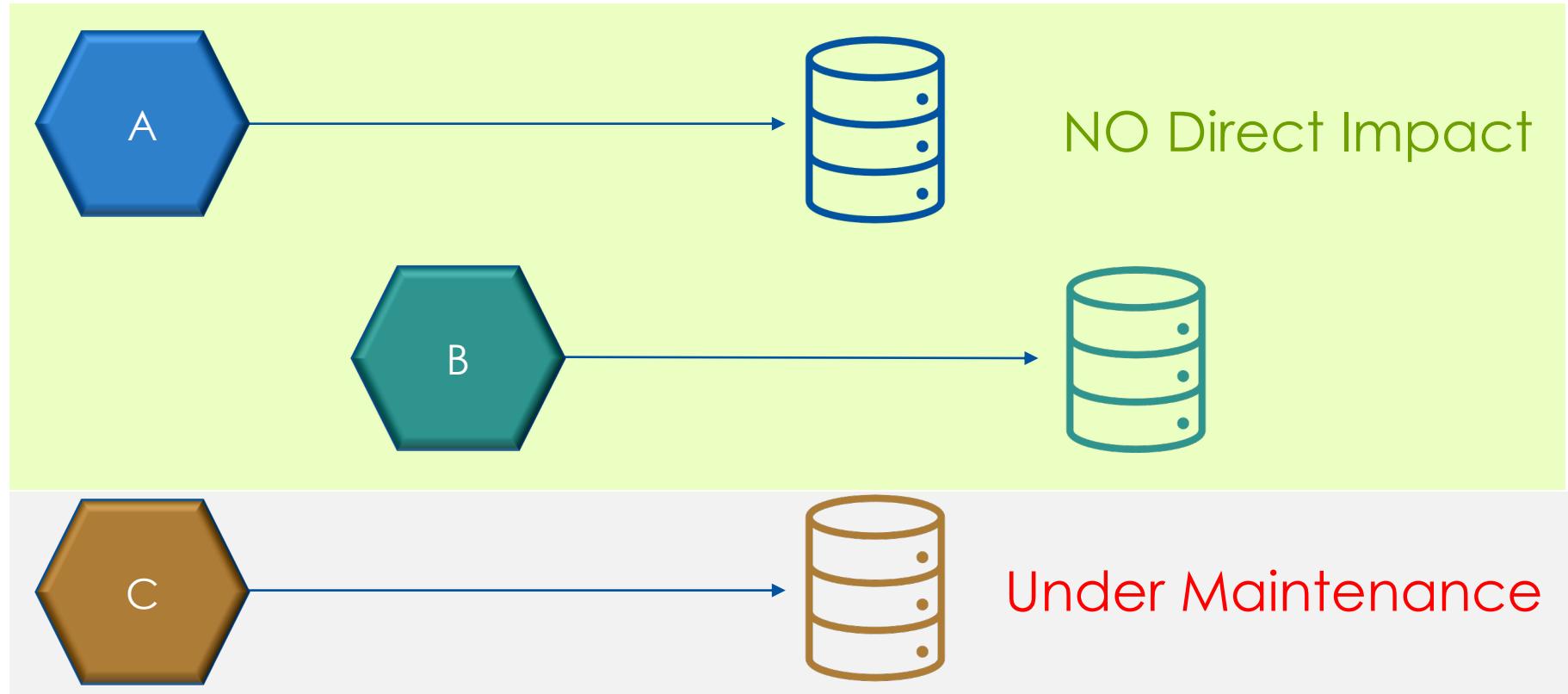


- No interdependency between teams
- Each team decide on their tech stack
- Service interactions via defined interfaces
- No direct access to data

Each microservice has its own Database !!!

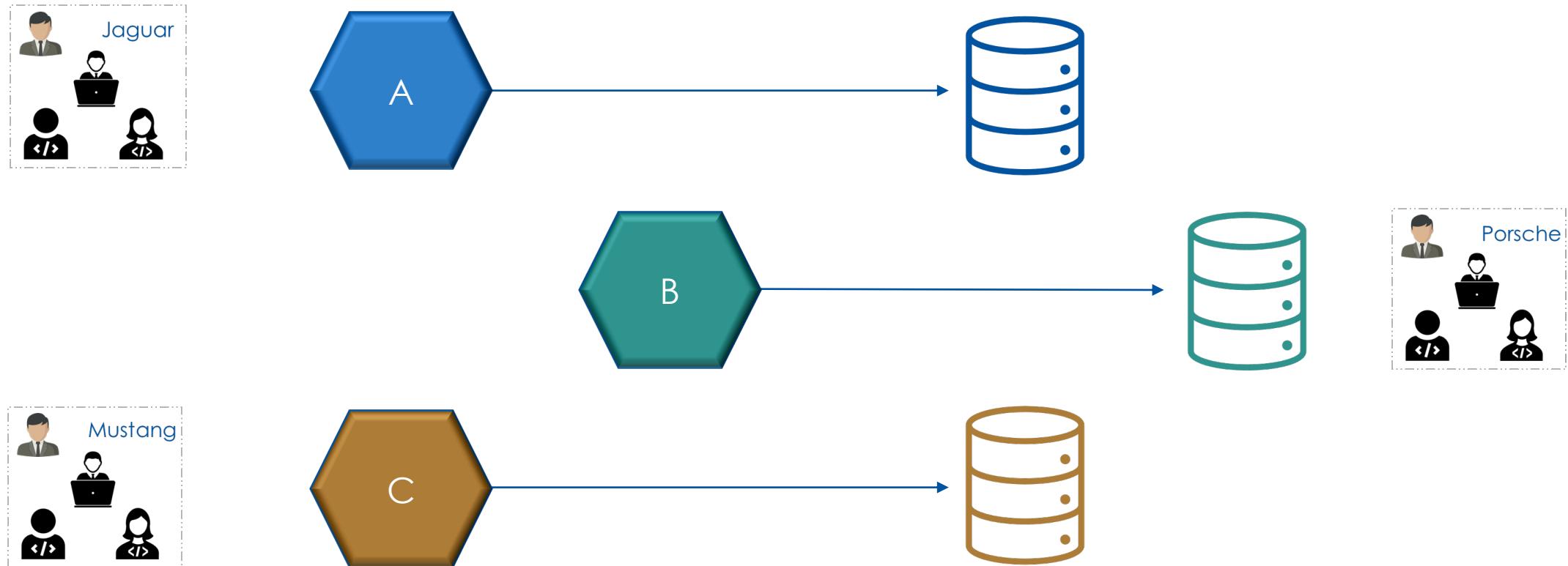
Benefits : Separate Database Pattern

Simpler change management



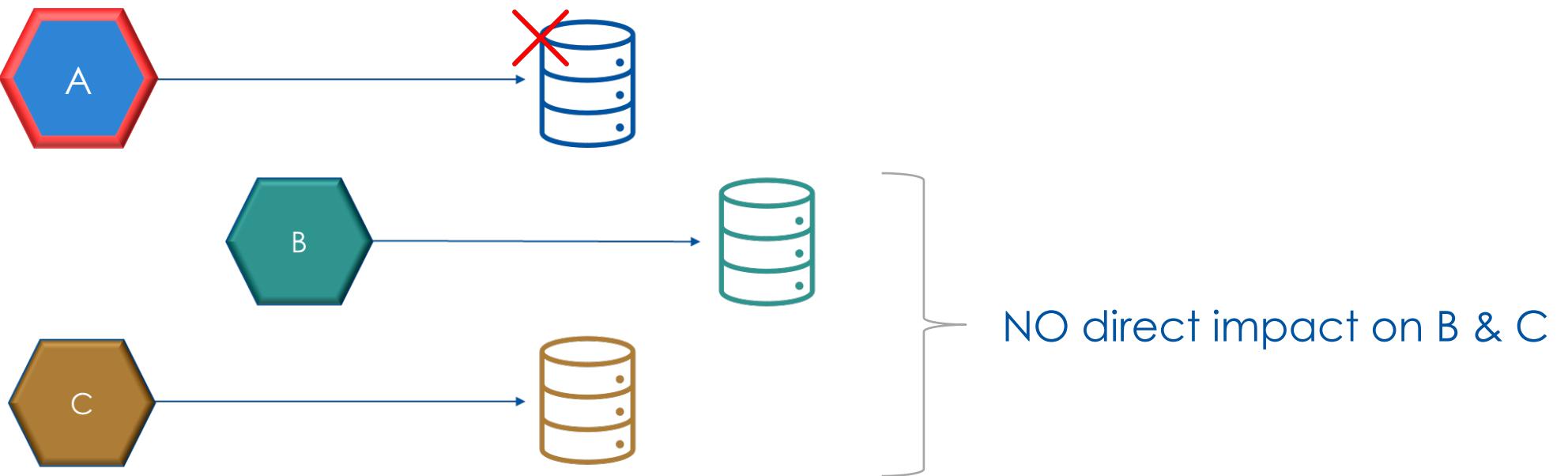
Separate Database Pattern

Each microservice team owns & manages their database



Benefits : Separate Database Pattern

Reduced blast radius on Database Failure



Benefits : Separate Database Pattern

Capacity planning | Scaling at DB level becomes simpler



- No new capacity for next year



- Additional 5 TB for storage



- 1 TB Storage
- CPU +2 cores

Benefits : Separate Database Pattern

Each team can decide on Database i.e., doesn't have to RDBMS



- RDMBS



- NoSQL



- RDBMS



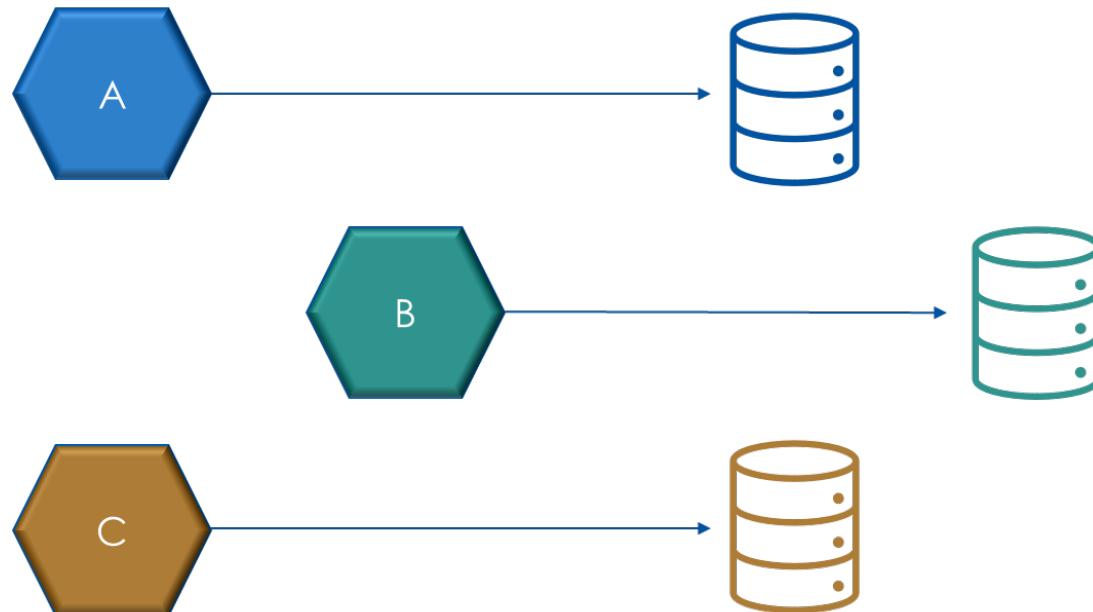


Quick Review

Greenfield



Separate Database Pattern is recommended



Brownfield & Databases

Converting a Monolith to Microservices

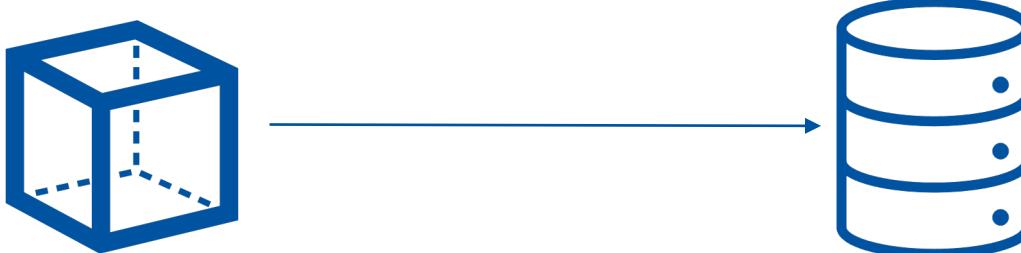


- 1 Converting Brownfield to Microservices
- 2 Shared Database = Anti pattern
- 3 Strangler pattern



Conversion to Microservices

A monolith is refactored to multiple microservices



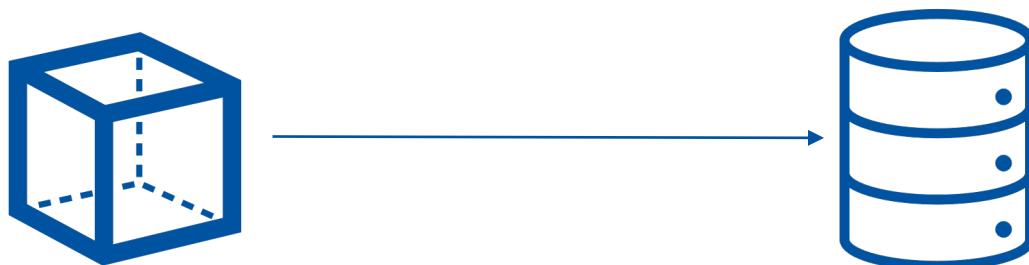
Shared Database

- 1 Apply Separate DB Pattern
- 2 Database Refactoring
- 3 Logical separation of database



Conversion to Microservices

A monolith is refactored to multiple microservices



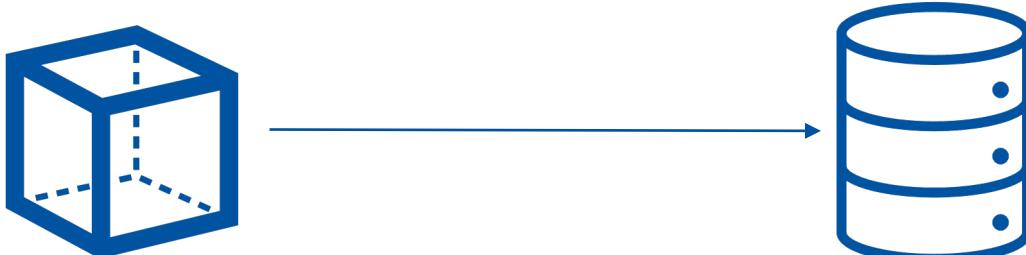
Apply Separate DB Pattern

Using the Strangler Pattern



Conversion to Microservices

A monolith is refactored to multiple microservices



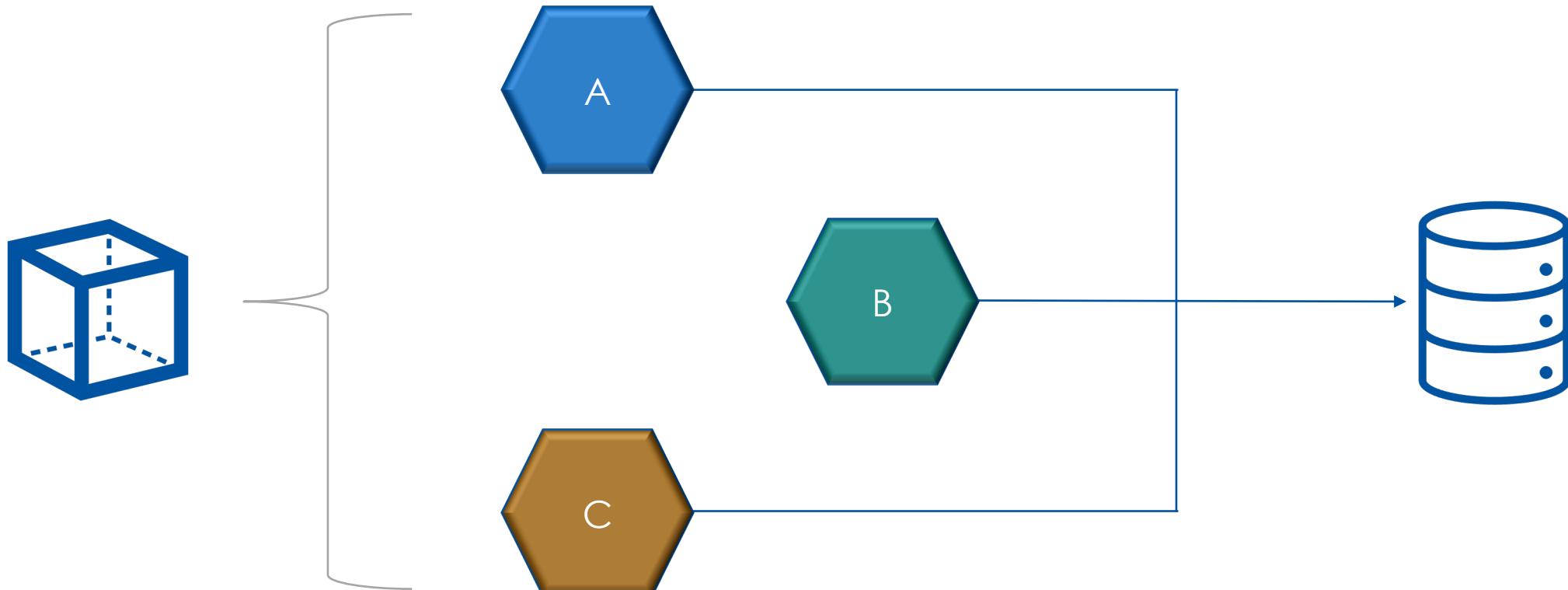
System of records in RDBMS

- Hundreds of tables
- Complex relationships
- Stored Procedures | Triggers



Conversion to Microservices

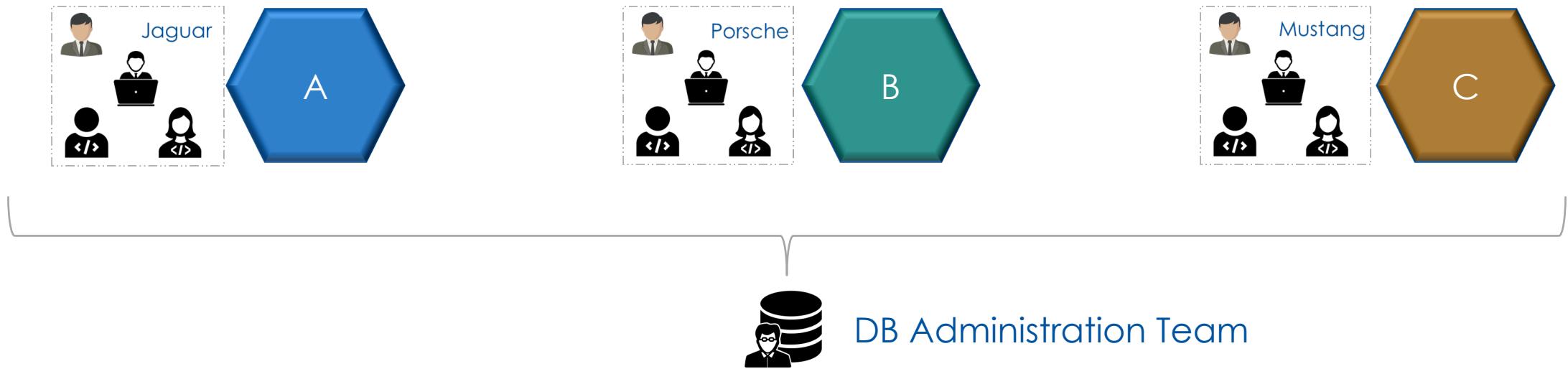
ONLY application refactored; DB stays in place



Microservices will suffer from the same challenges as applications !!!

Shared Database Pattern

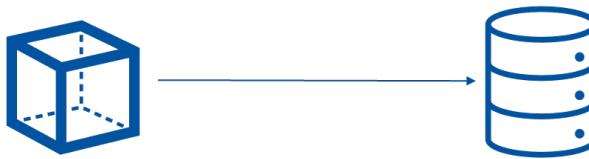
It is an Anti-Pattern as it leads to inter-dependencies



- Higher need for coordination
- Slow speed to value
- Increased testing effort
- No independent scaling

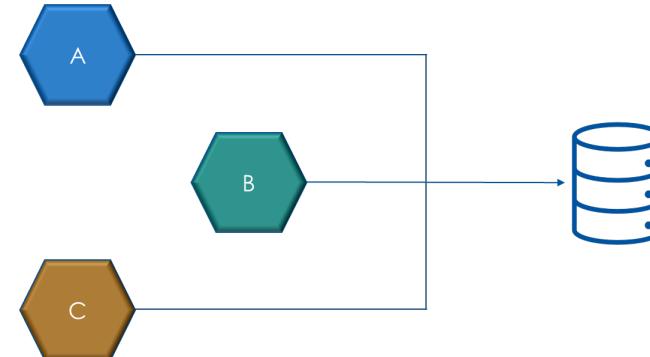
Shared DB : Transition State Architecture

Positioning the application for Microservices architecture



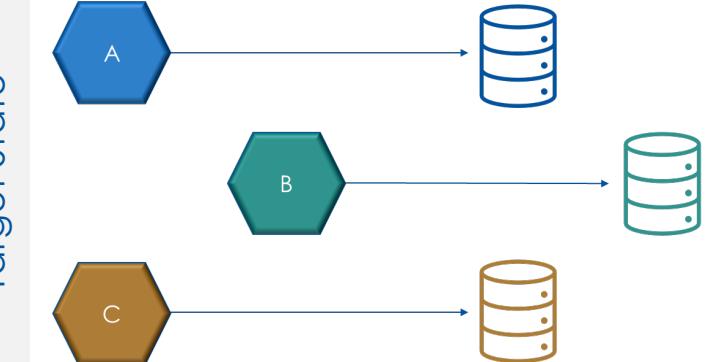
- Focus on Application

Transition State



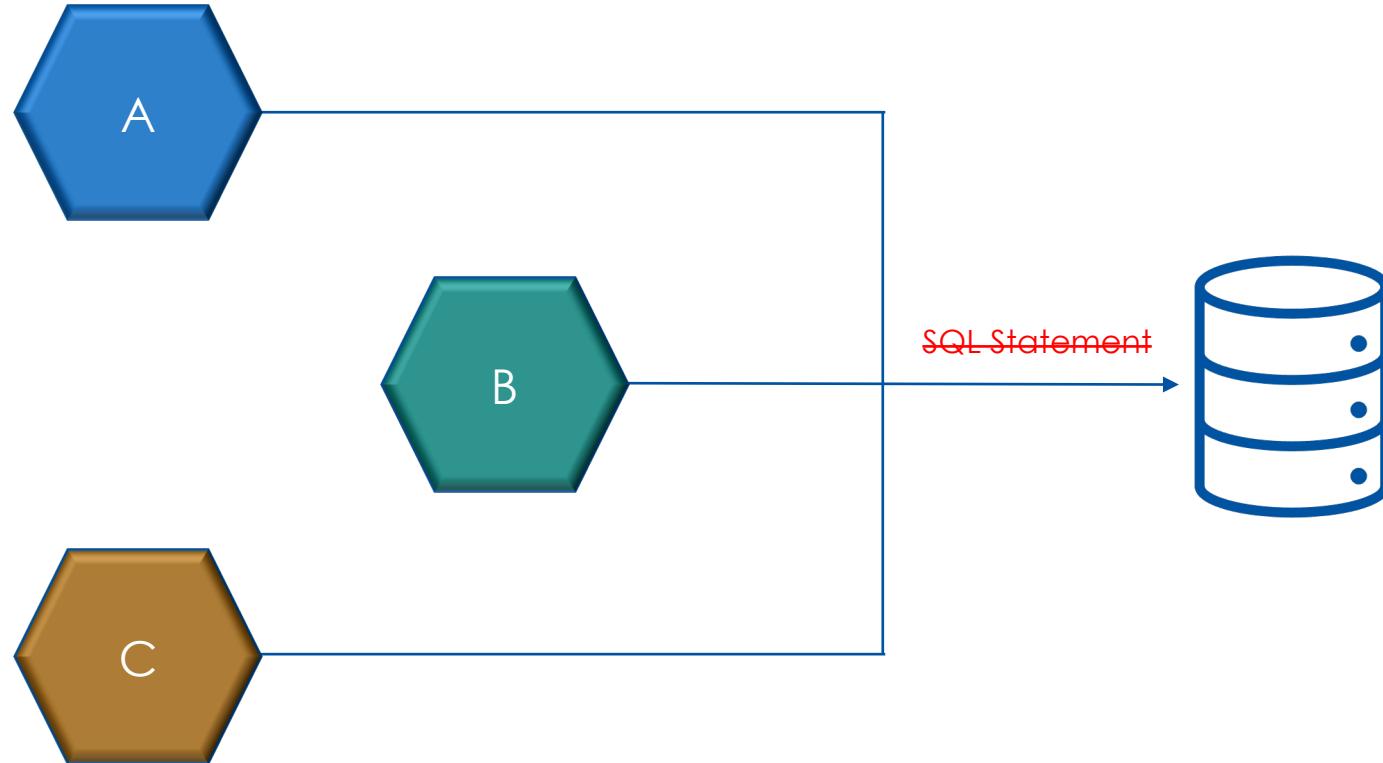
- Focus on Data

Target State



Shared DB : Transition State Architecture

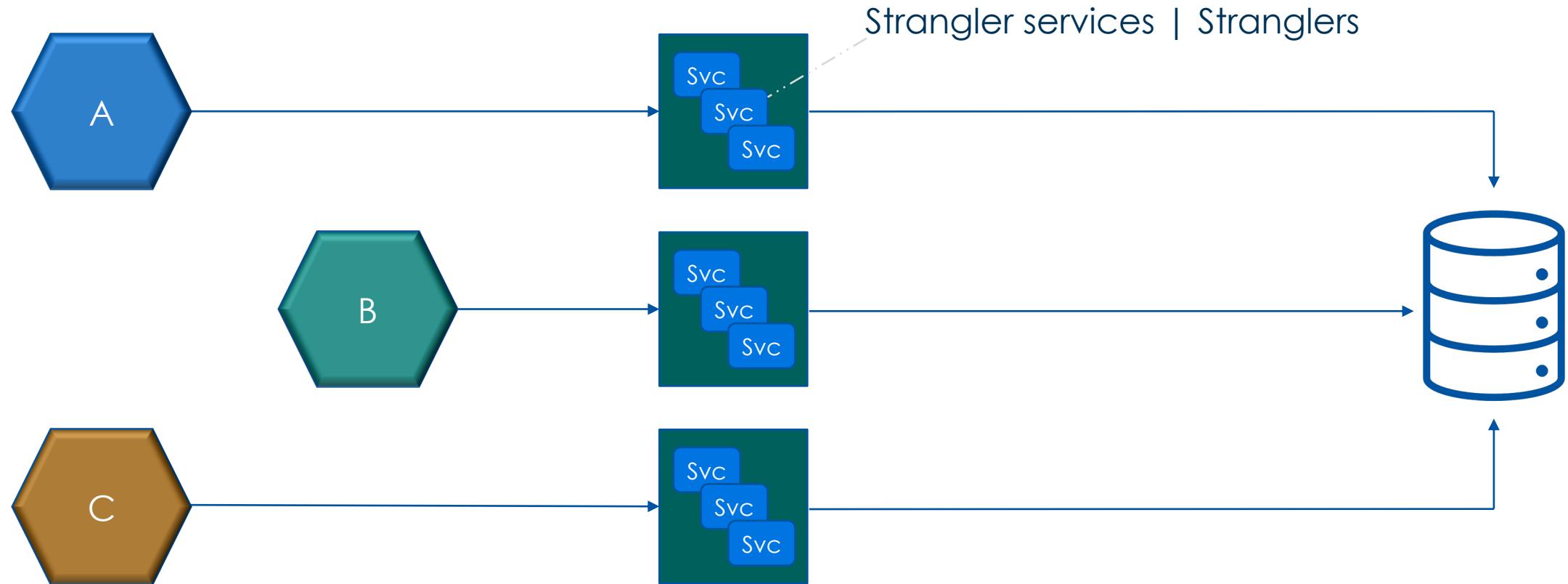
Keep the Microservices code independent of the Database



Use of SQL in code will make it difficult to switch the DB !!!

Strangler Strategy

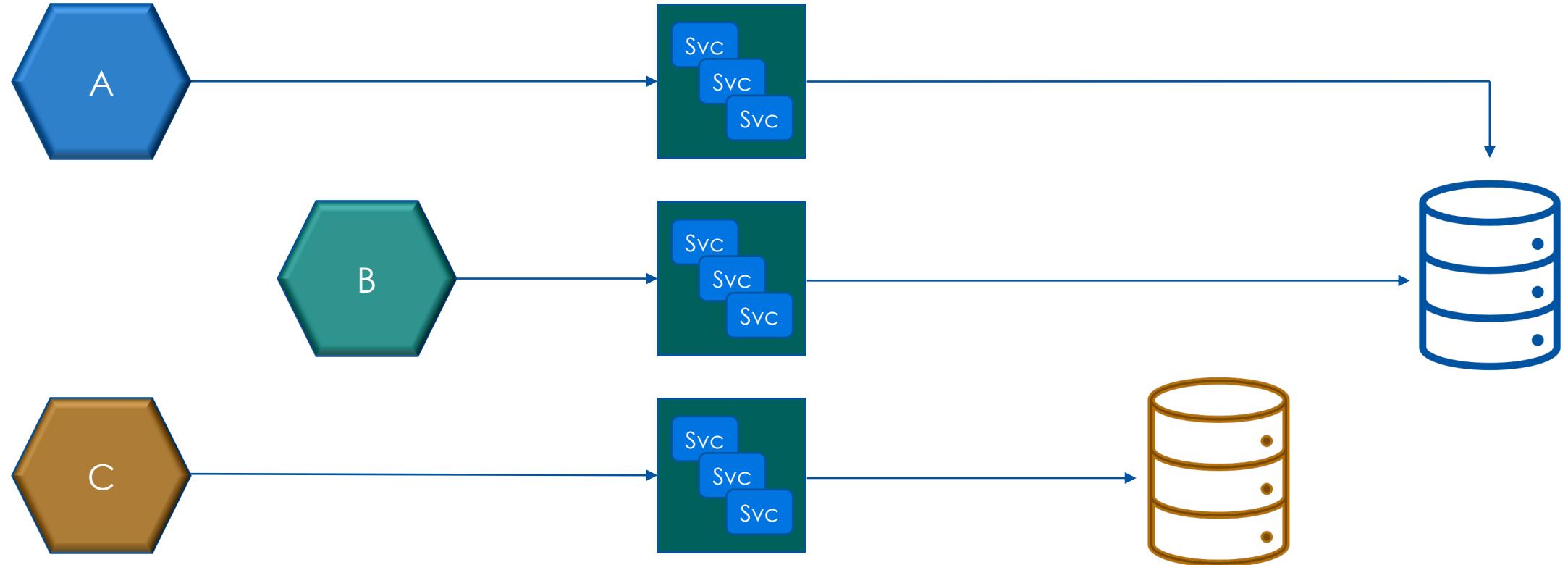
Access the data via services/API in the transition stage



Switching the DB will not affect the Microservice

Strangler Strategy

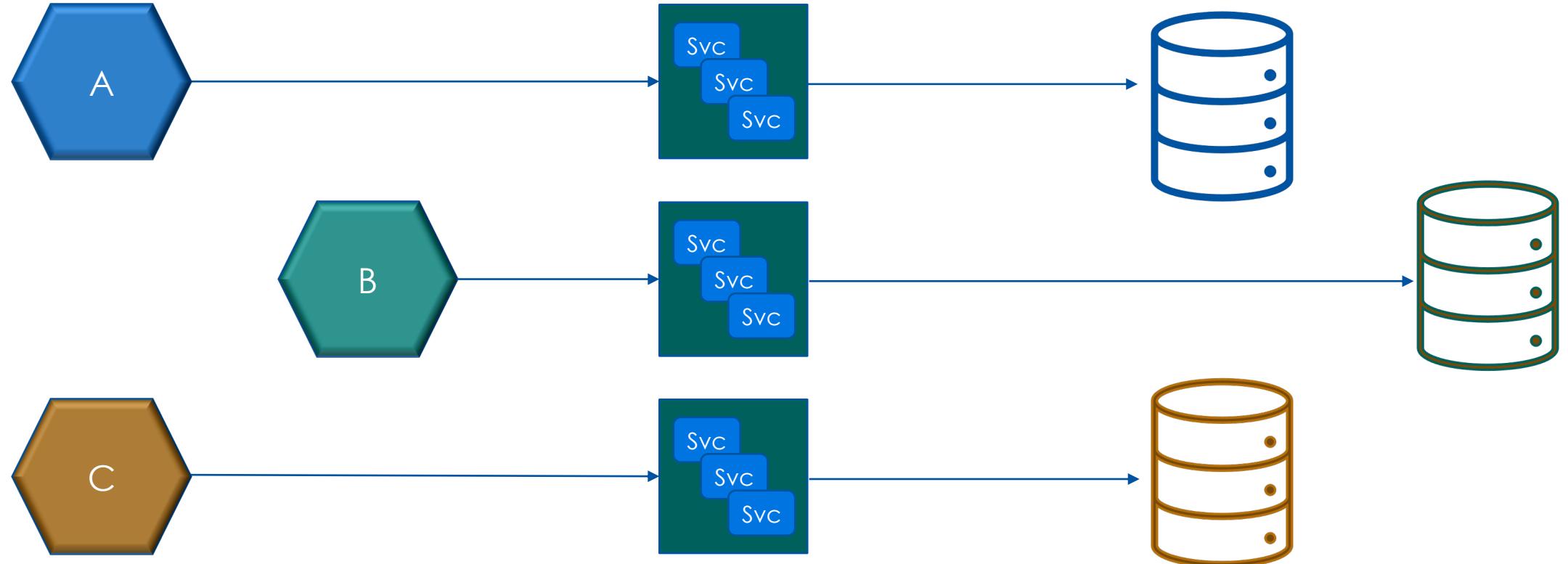
Access the data via services/API in the transition stage



Teams decide their priorities and work independently (almost)

Strangler Strategy

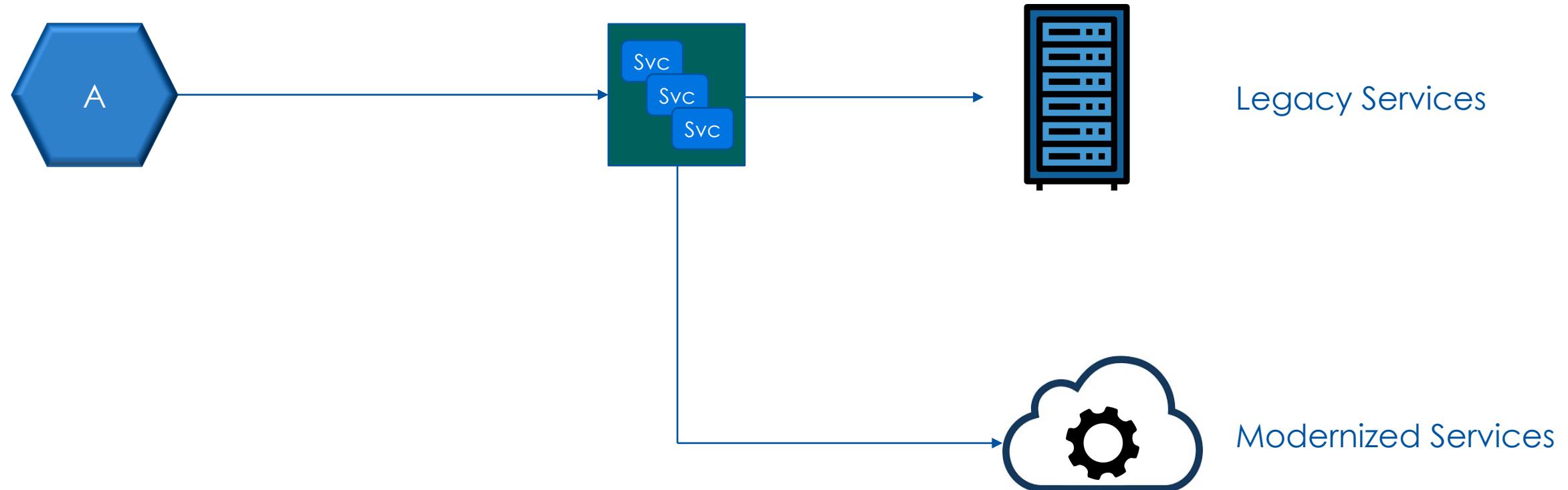
Access the data via services/API in the transition stage



Eventually the target state is achieved !!!

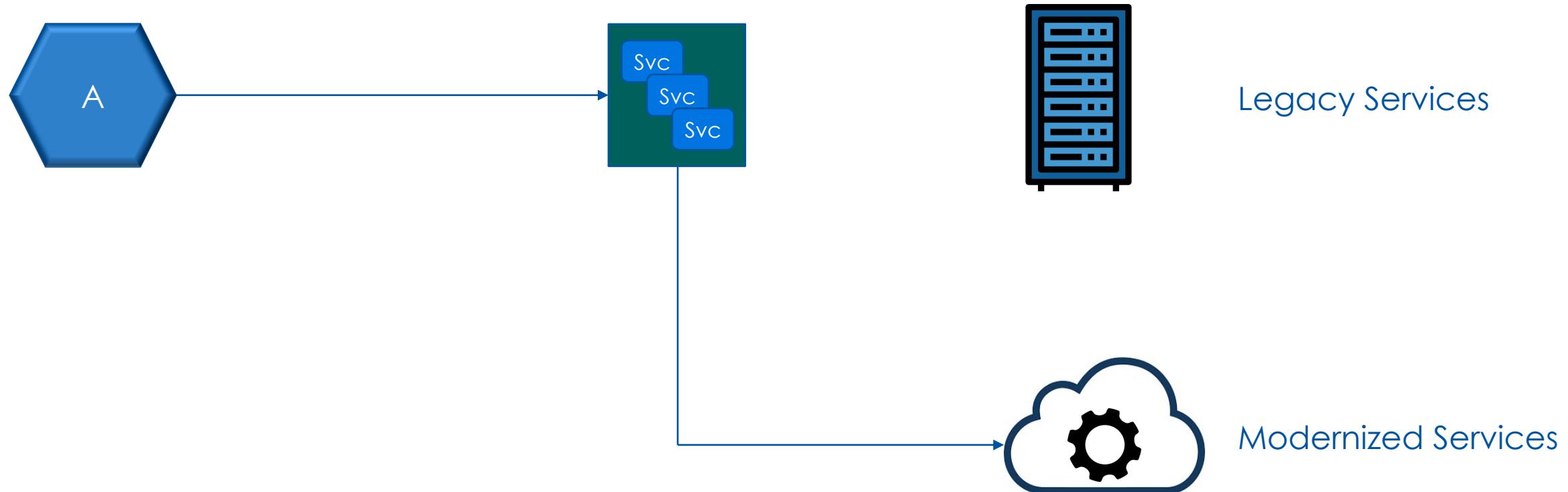
Strangler Pattern

May be used for other legacy services as well



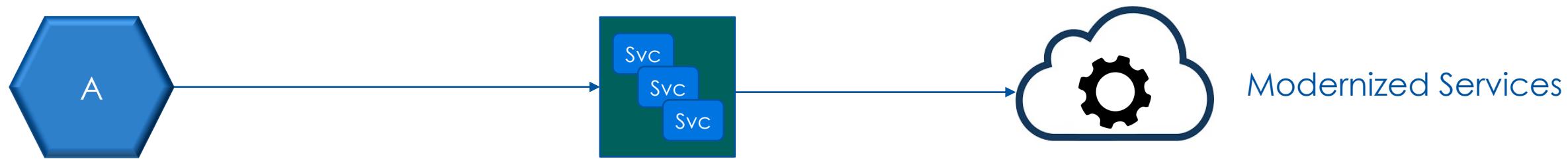
Strangler Pattern

Over a period, modernized service will replace the legacy service



Strangler Pattern

The strangler services will be changed to point to new services



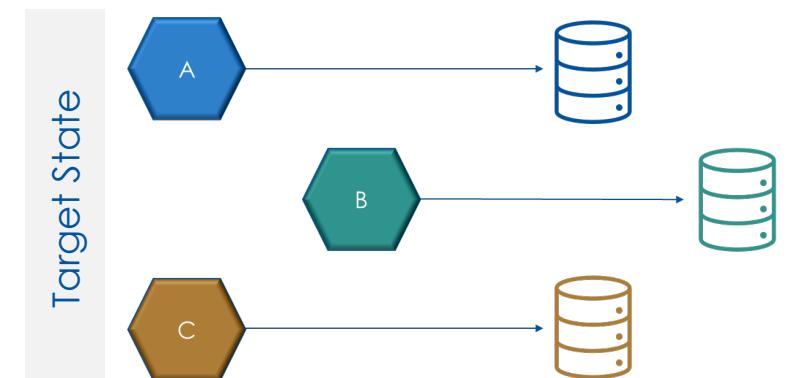
Microservice code insulated from backend changes !!!



Quick Review

Shared Database pattern = Anti pattern for Microservices

Strangler pattern used for replacing the backend | databases



Microservices : Shared DB Pattern

Converting Brownfield monolith to Microservices



- 1 Objective & Challenges
- 2 Database Refactoring
- 3 Logical Database Separation

Note: Discussion apply ONLY to an RDBMS



Reality check

One may not have the flexibility of using "Separate DB"

For multiple reasons:

- Time constraints
- Cost | Budget constraints
- Lack of skilled resources
- Higher risk
- Legacy technology
- ...

Objective

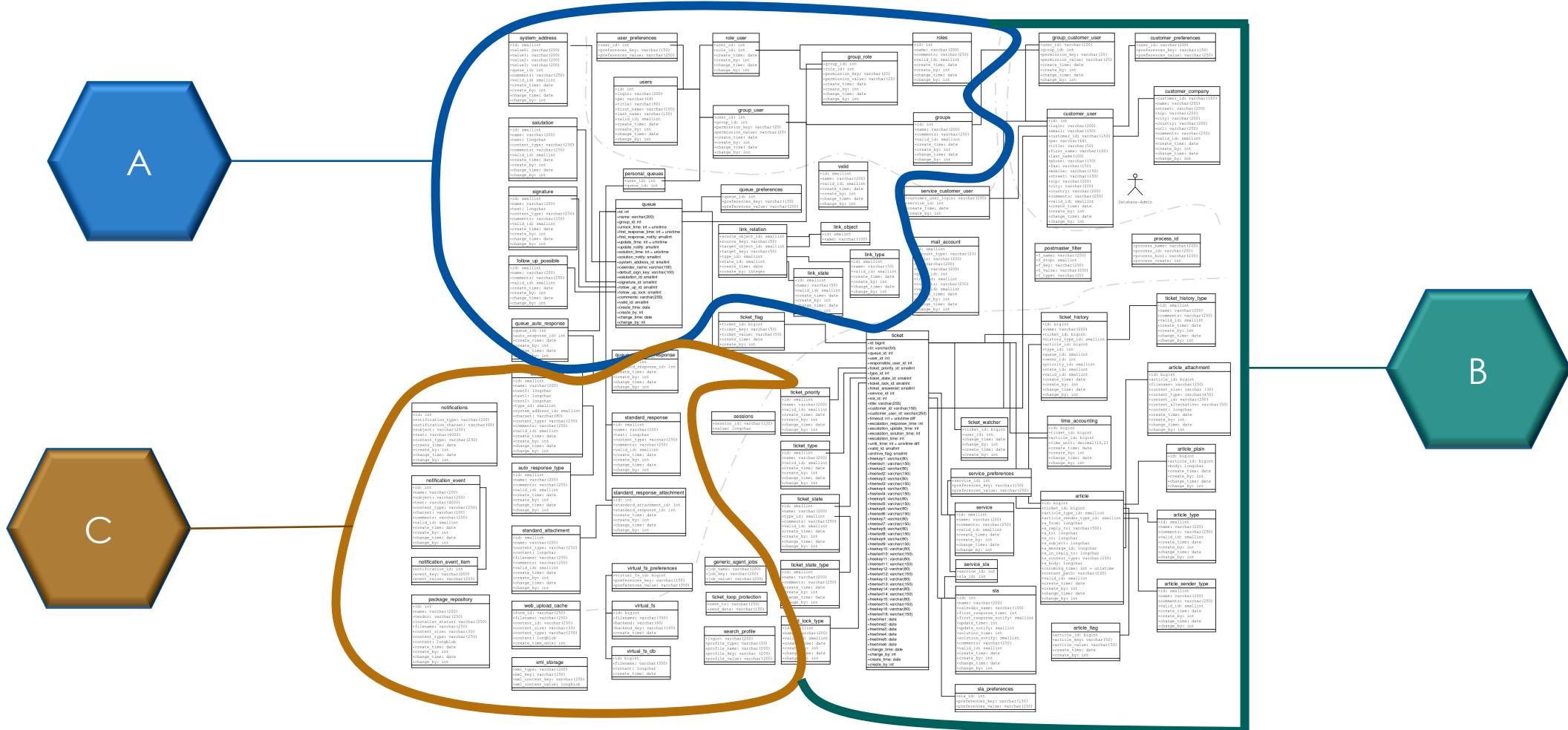
Achieve isolation of data within the same Database instance

- Each Microservice owns part of the data in the Database
- Each Microservice have direct access to ONLY its own data



Challenges with breaking the DB

Large Databases are NOT easy to separate 😞

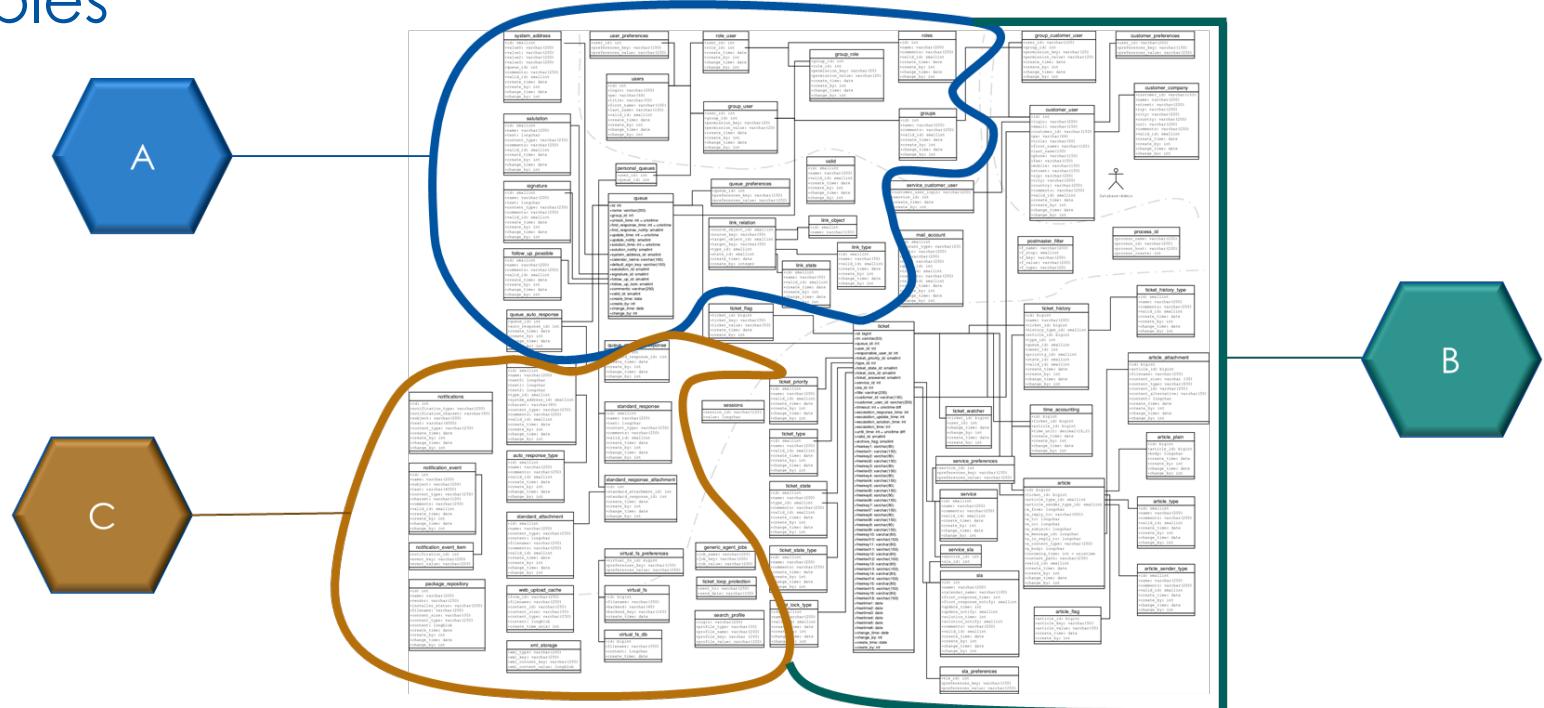




Challenges with breaking the DB

Separation doesn't end with segregating the tables

- Shared data across Microservices
- Relationships between tables
- Stored Procedures
- Triggers



Dealing with Shared Database



Database Refactoring

- Changes to underlying database

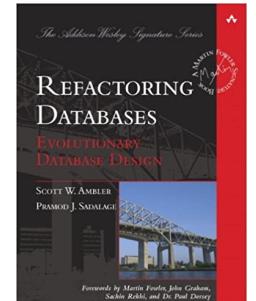


Logical separation of database

- Use the database as-is

“

A small change to the database which improves its design without changing its semantics



by Scott W & Pramod S

Database Refactoring

There are 6 change categories suggested for DB Refactoring

Structural

Changes to definition of tables, views, and columns

Architectural

Changes to methodology on how apps interact with DB

Referential
Integrity

Changes to the Primary Key, Foreign Keys, Triggers

Database Refactoring : Change Categories

There are 6 categories of change suggested for DB Refactoring

Methods

Code changes to Stored Procedure like adding and removing parameters

Transformations

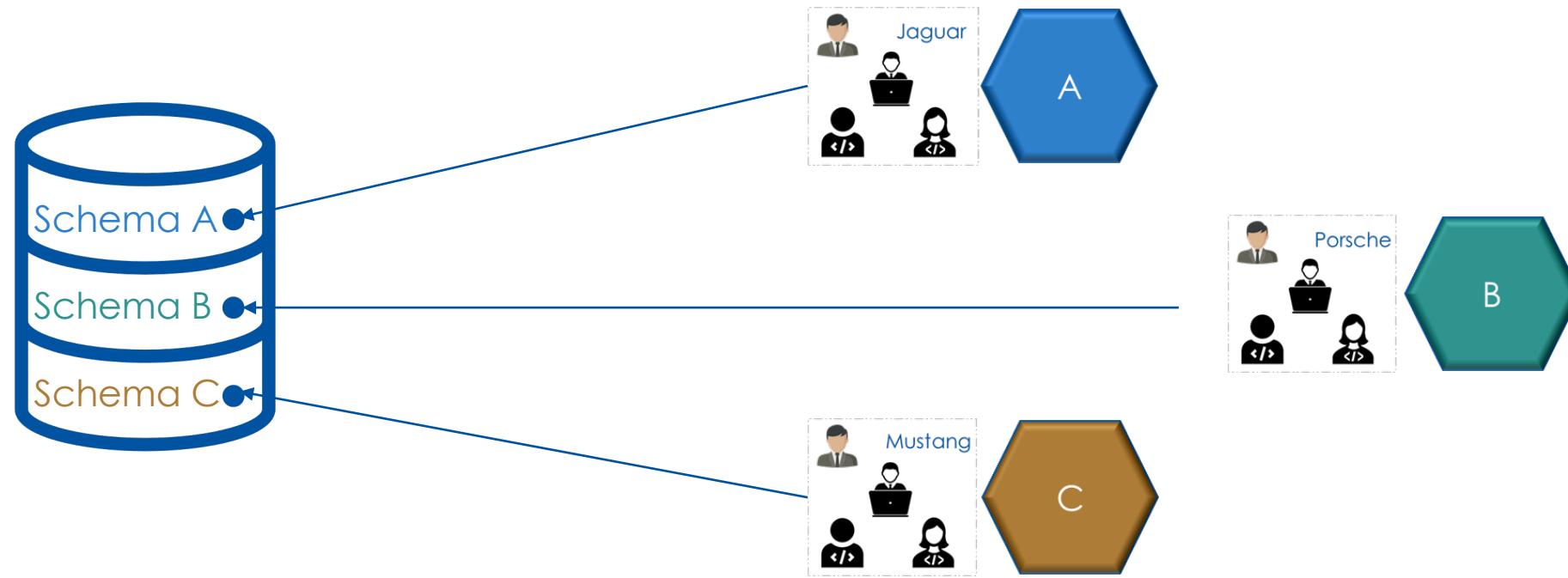
Changes to the database schema

Data Quality

Changes for improvement to data quality

Example : Structural Refactoring

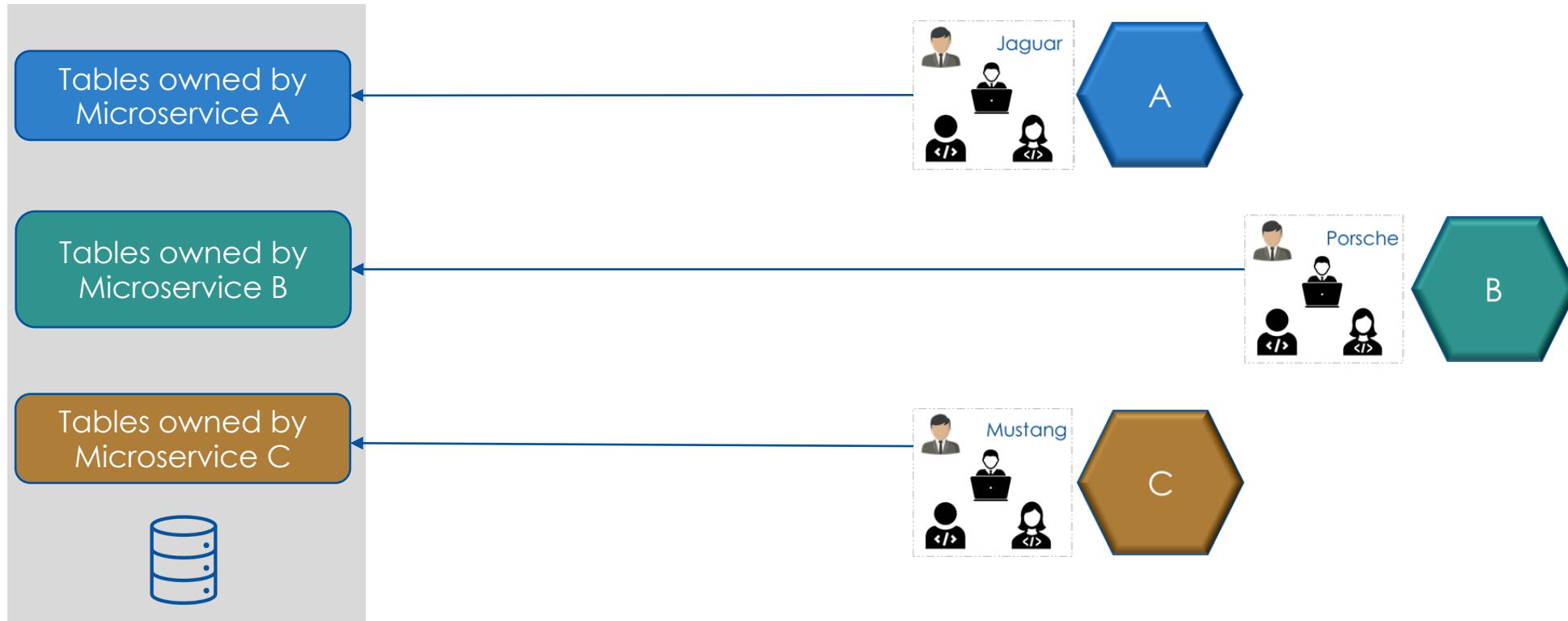
Separate schemas and put in access control for microservices



How is it done? Depends on the specific RDBMS !!!!

Logical separation of the Database (RDBMS)

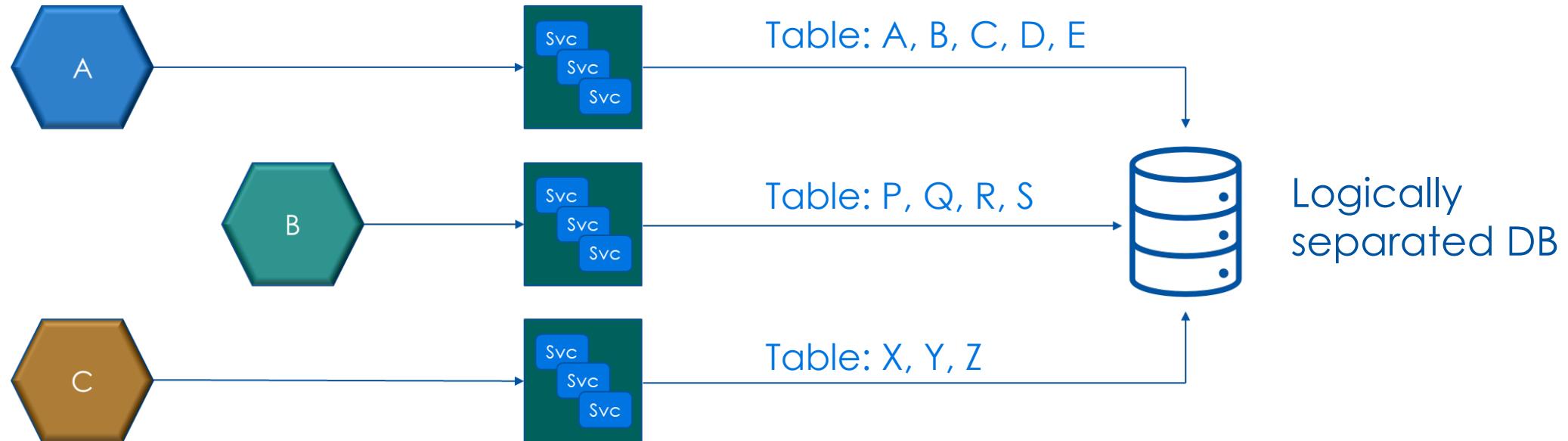
Divide the related tables among Microservices



Teams need to be disciplined; not to access other MS data directly

Using services for data access

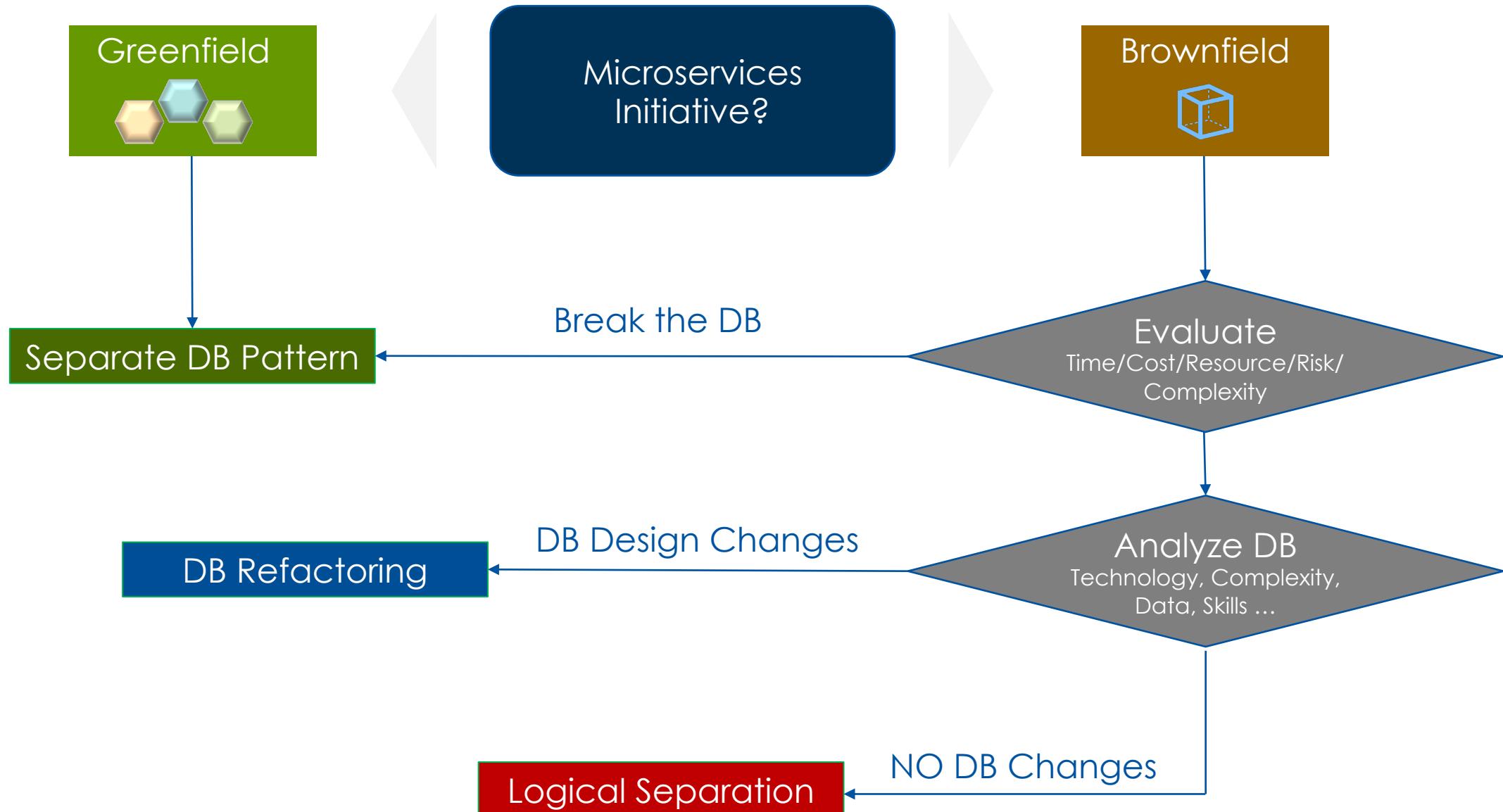
All data access via services to minimize risk of direct access



- Governed services
- Provides control over only the assigned tables/data



Quick Review



Downside of Separate Databases

But there are solutions !!!



- 1 Challenges
- 2 Solution to challenges
- 3 Introduction to CQRS, Event Sourcing & SAGA patterns

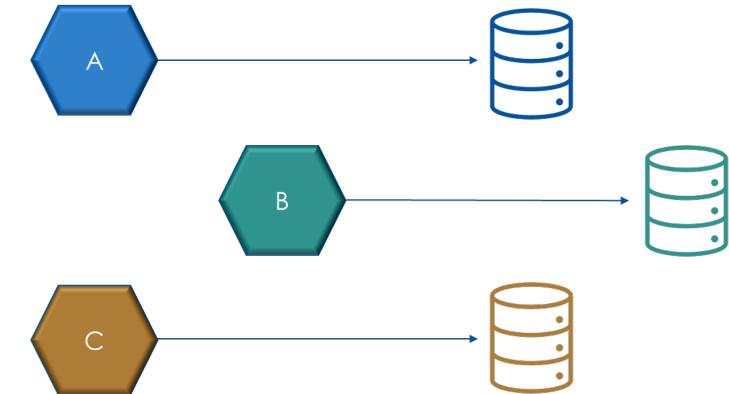
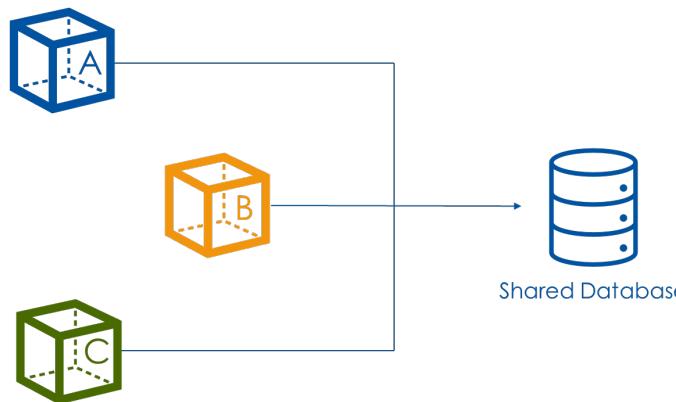
Downside : Separate Database Pattern



HIGHER cost of the Database for the solution

Licensing | Hardware

Operational | Maintenance

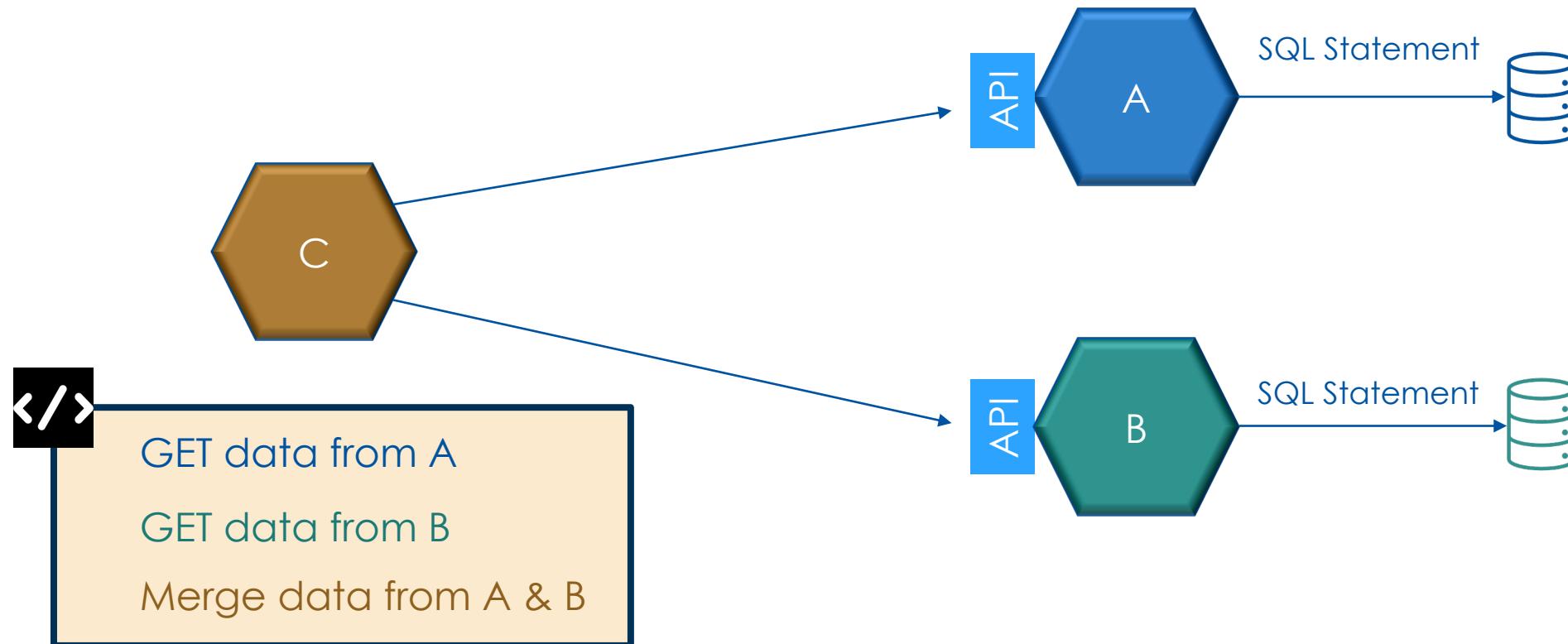


\$\$\$ spent on Databases may go up by up to 3 times !!

Downside : Separate Database Pattern

2

Degraded Read Performance due to distribution of data



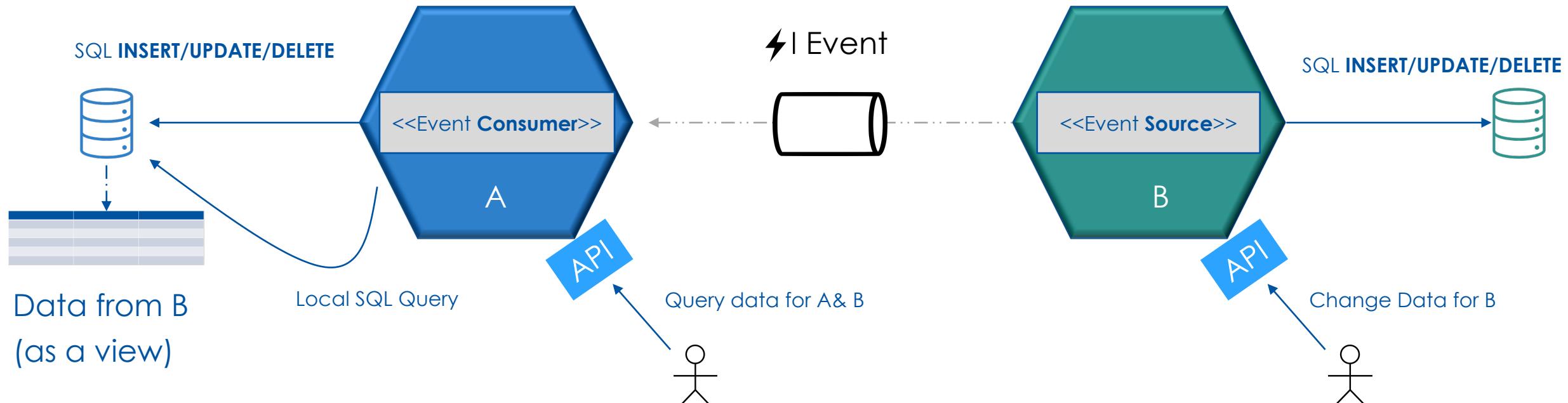
In a Shared DB - we could have used a JOIN !!!

Addressing the challenge

2

Degraded Performance due to distribution of data

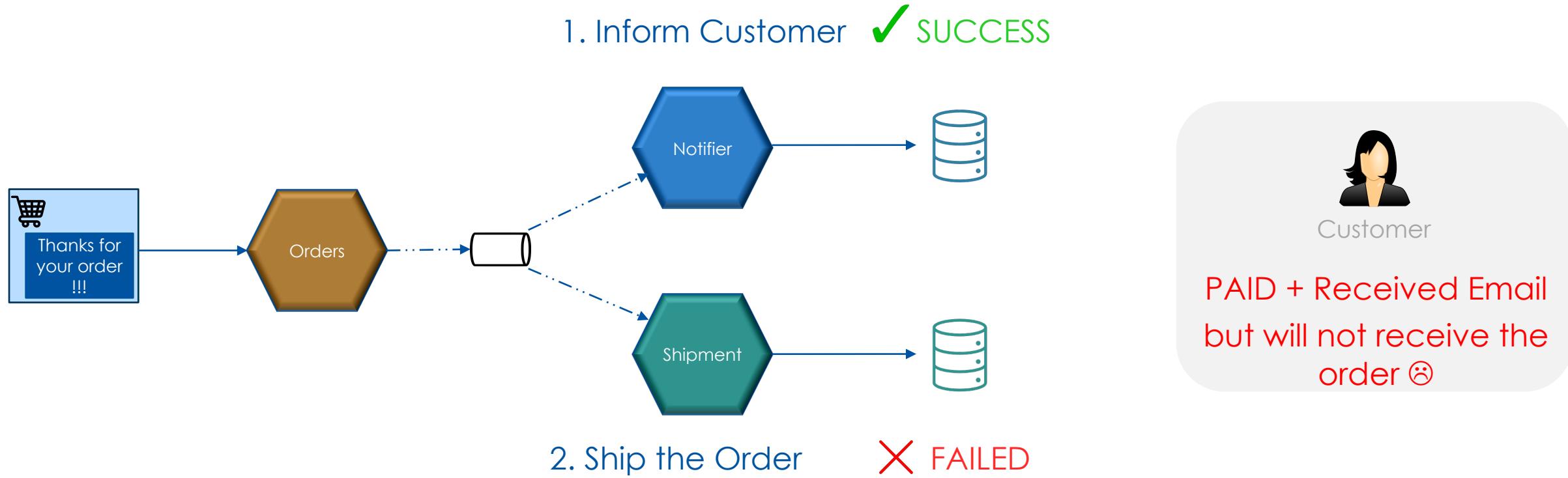
► Manage local copy of the data thru asynchronous replication



Downside : Separate Database Pattern

3

Complexity in managing Transaction | Data integrity



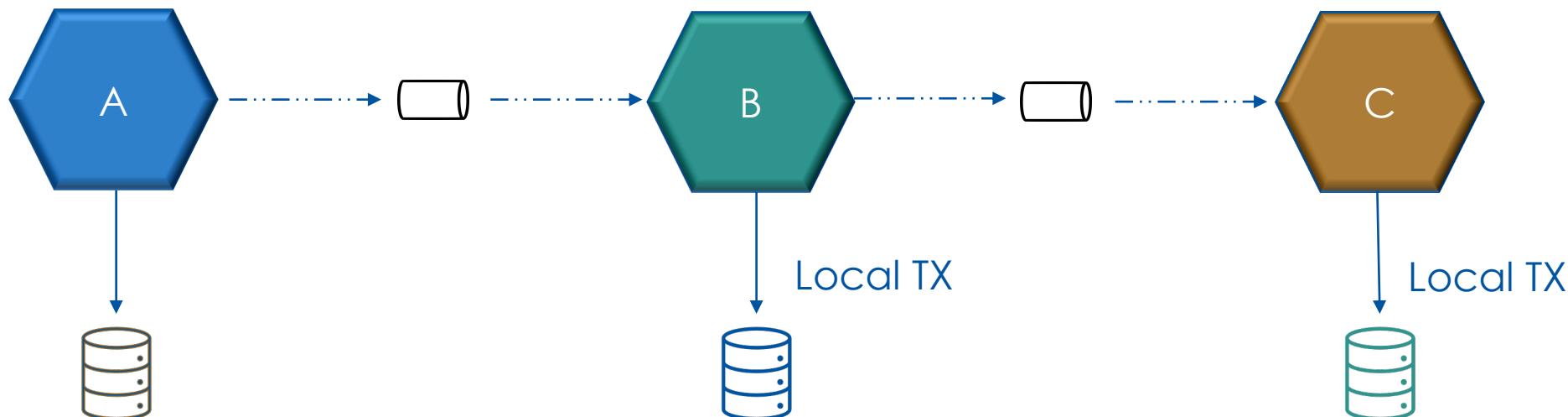
Shared DB - Could have used a local Transaction !!!

Addressing the challenge

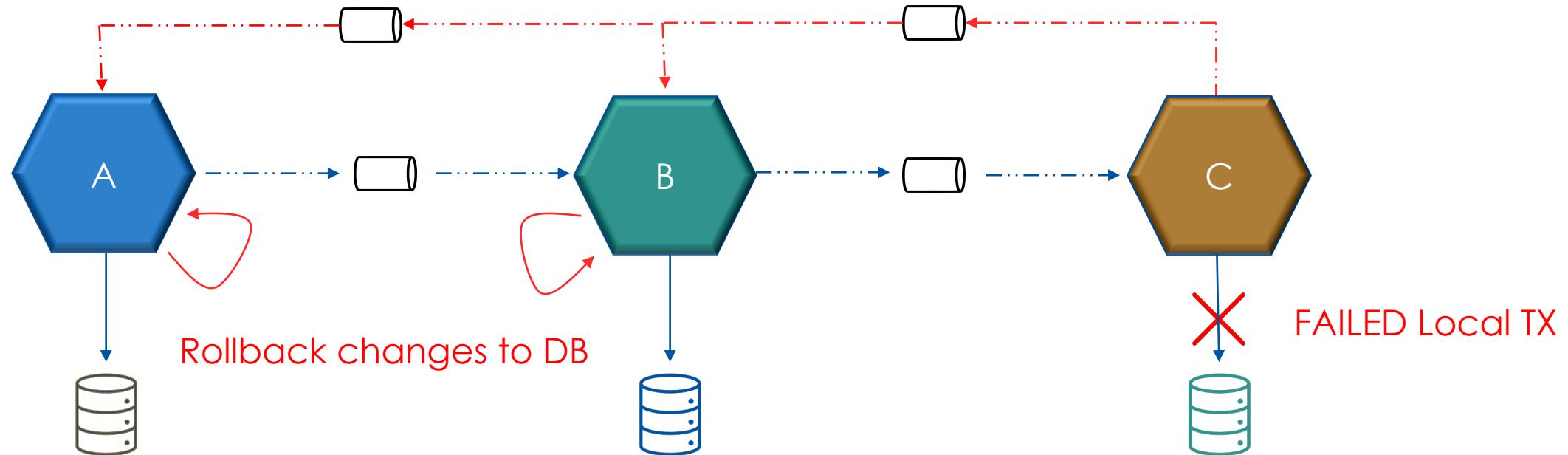
3

Complexity in Data integrity | Transaction management

- ▶ Use a sequence of local transactions with distributed rollback



- ▶ Use a sequence of local transactions with distributed rollback





Quick Review

- 1 HIGHER cost of the Database for the solution
 - Open Source, Cloud Native Databases
- 2 Degraded Performance due to distribution of data
 - CQRS Pattern & Event Sourcing
- 3 Complexity in Data integrity | Transaction management
 - SAGA Pattern & Reliable messaging