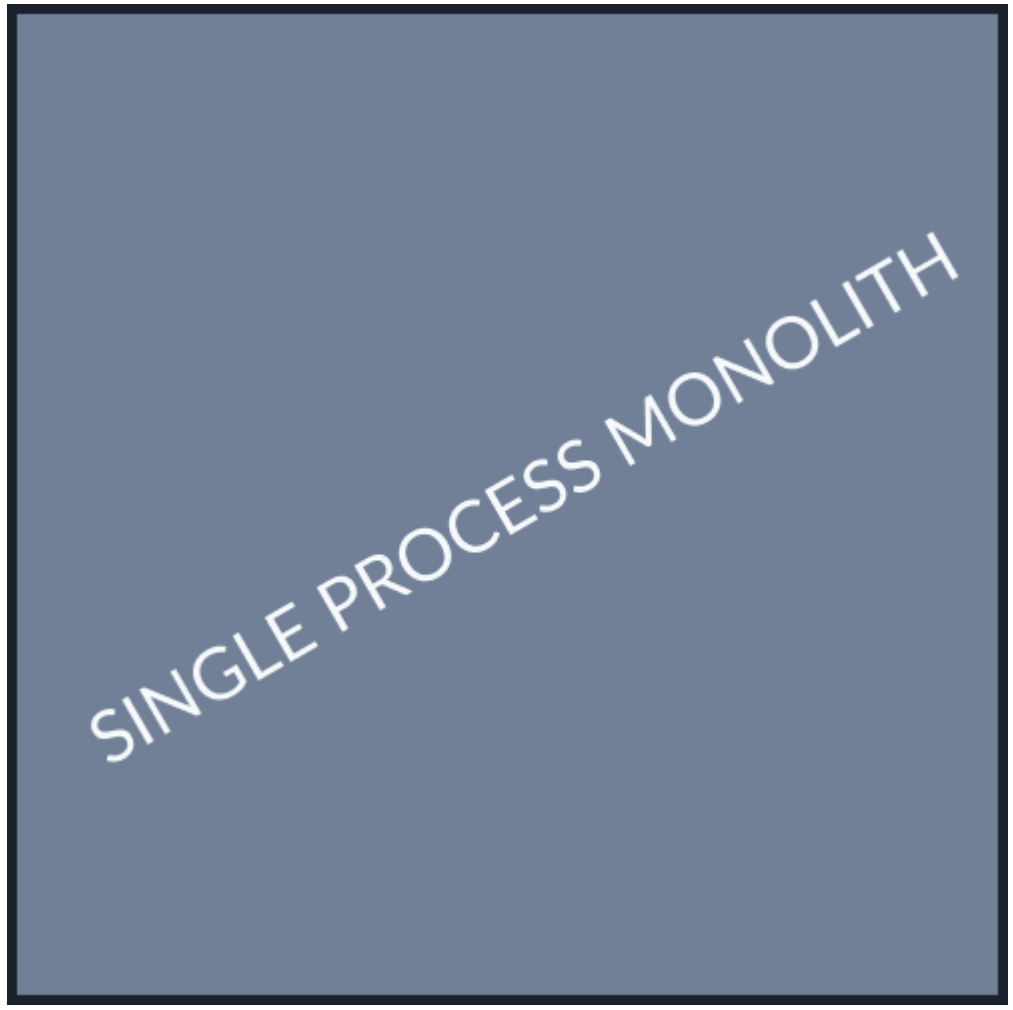
Deconstructing a Monolith

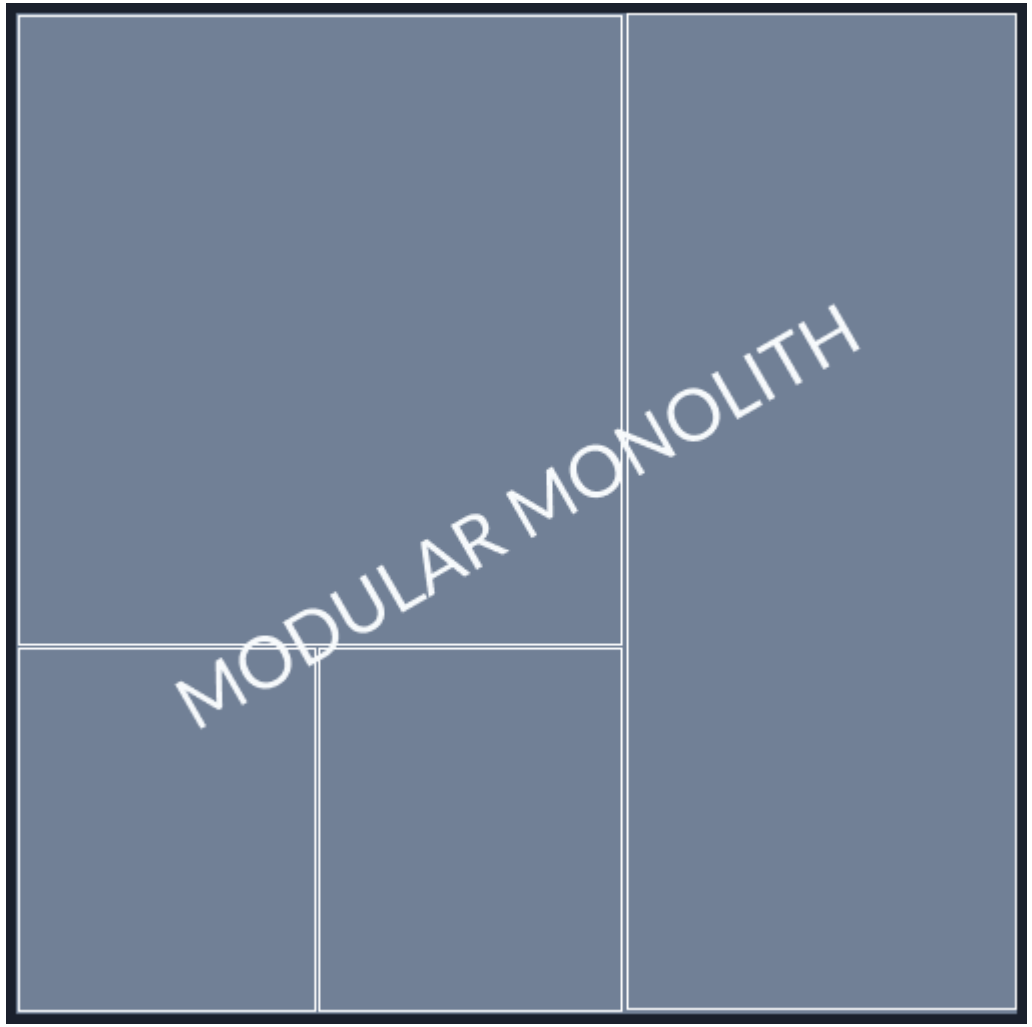
Introduction

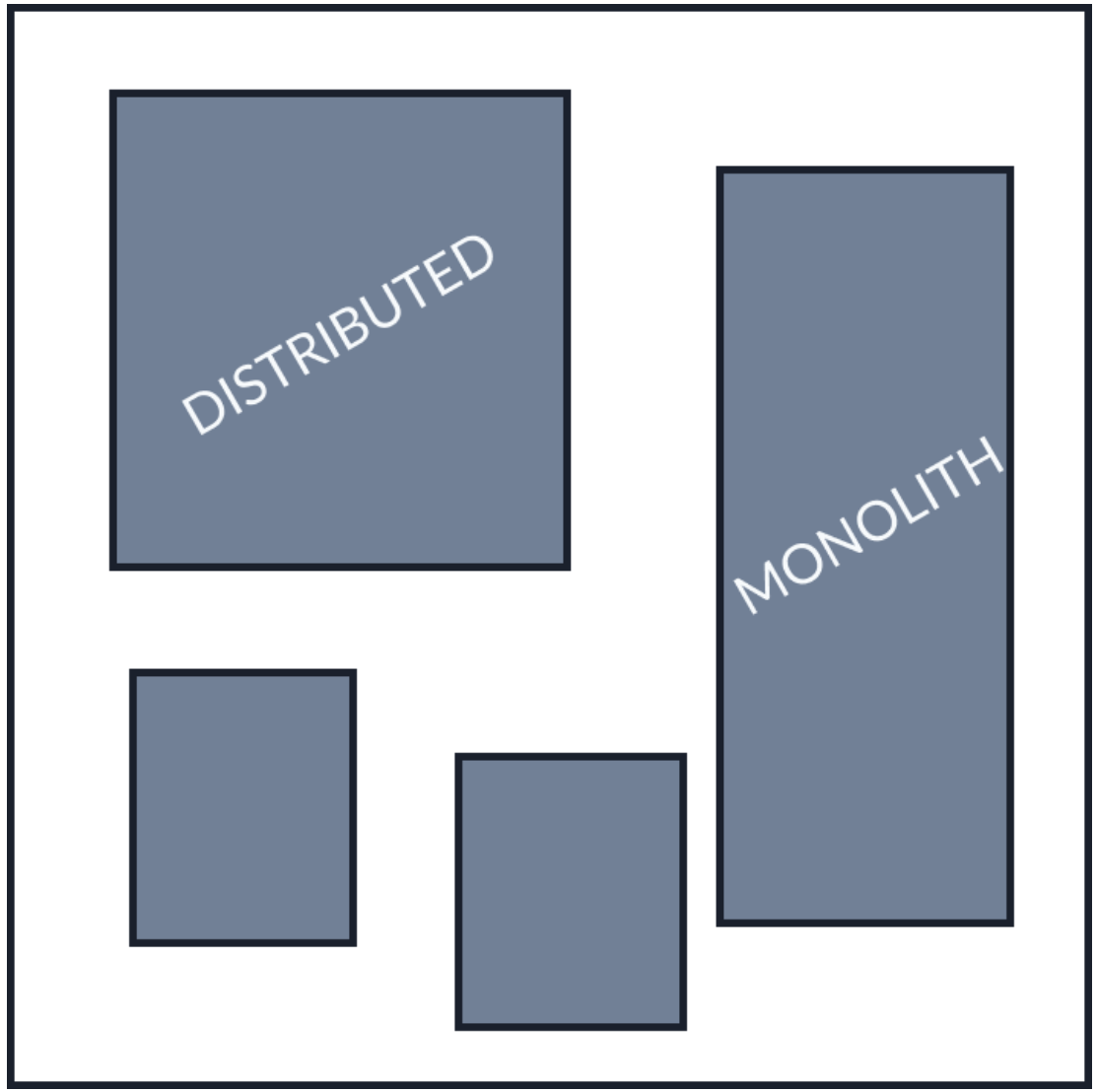
- I am Peter (@pjausovec)
- Software Engineer at Oracle
- Working on "cloud-native" stuff
- Books:
  - Cloud Native: Using Containers, Functions, and Data to Build Next-Gen Apps
  - SharePoint Development
  - VSTO For Dummies
- Courses:
  - Kubernetes Course (https://startkubernetes.com)
  - Istio Service Mesh Course (https://learnistio.com)

Agenda

1. What are Monoliths and Microservices?
2. Migration Patterns
    - Strangler Pattern
    - Branch by Abstraction Pattern
    - Parallel Run Pattern
3. Decomposing Databases
    - Database View Pattern
    - Database Wrapping Service Pattern
    - Database-as-a-Service Pattern
    - Change Data Capture
    - Aggregate Exposing Monolith
    - Change Data Ownership Pattern
4. Data Synchronization
5. Transactions


SINGLE PROCESS MONOLITH


MODULAR MONOLITH


DISTRIBUTED
MONOLITH

Monolith Challenges

- Developers getting in each others way
  - changing same piece of code
  - delaying deployments
- Ownership

Monolith Advantages

- Simpler/single deployment
- Simpler inner loop/development
  - also monitoring and E2E testing
- Code reuse

Monolithic architecture is an option

Microservices Characteristics

- Independently deployable
- Modeled around a business domain
- Own their own data

What's the single biggest advantage of microservices?

a) Scaling (like Netflix!)

b) Use any tech/language

c) Flexibility

d) Simpler deployment

FLEXIBILITY

Microservice Challenges

- Networking
  - Distributed system, CAP theorem and all that
- It's not server-side only - What about the UIs?
- Urge to use the latest and greatest
- Culture change

Recap

- Monolithic deployment
- Monoliths **can** be a good choice
- Avoid distributed monoliths
- Microservices offer **flexibility**
- A lot of network

Planning a migration

Microservices are not the goal

"Would you tell me, please, which way I ought to go from here?"

"That depends a good deal on where you want to get to"

"I don't much care where"

"Then it doesn't matter which way you go" ...

Questions to ask

- What are you hoping to achieve?
- Which alternatives did you consider?
- How do you know if migration worked?

Reasons

- Improve team autonomy
  - Amazon's two-pizza team
  - Spotify's product squads
- Reduce time to market
- Scale and robustness
- Scale the developers
- Embrace new tech

Migration Patterns

Strangler Pattern

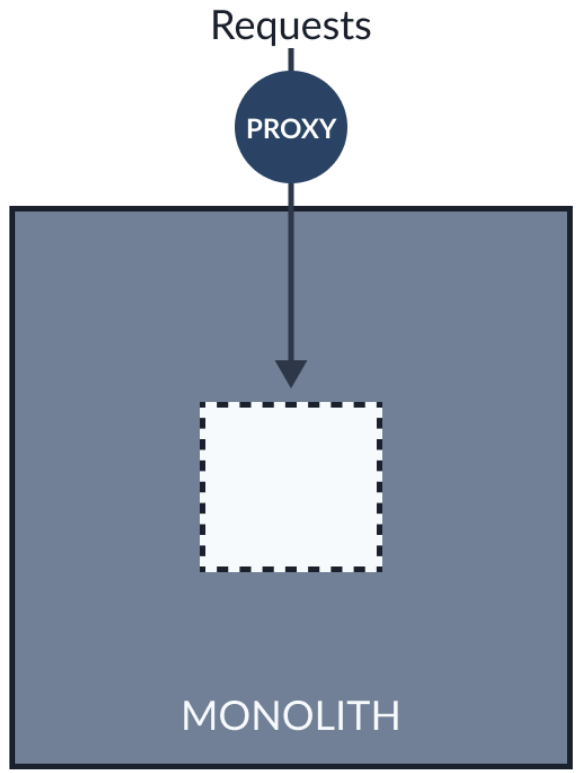
Strangler Pattern

- Used when doing system rewrites
- Both systems coexist
- Allows for incremental changes and pausing/stopping the migration

Strangler Pattern

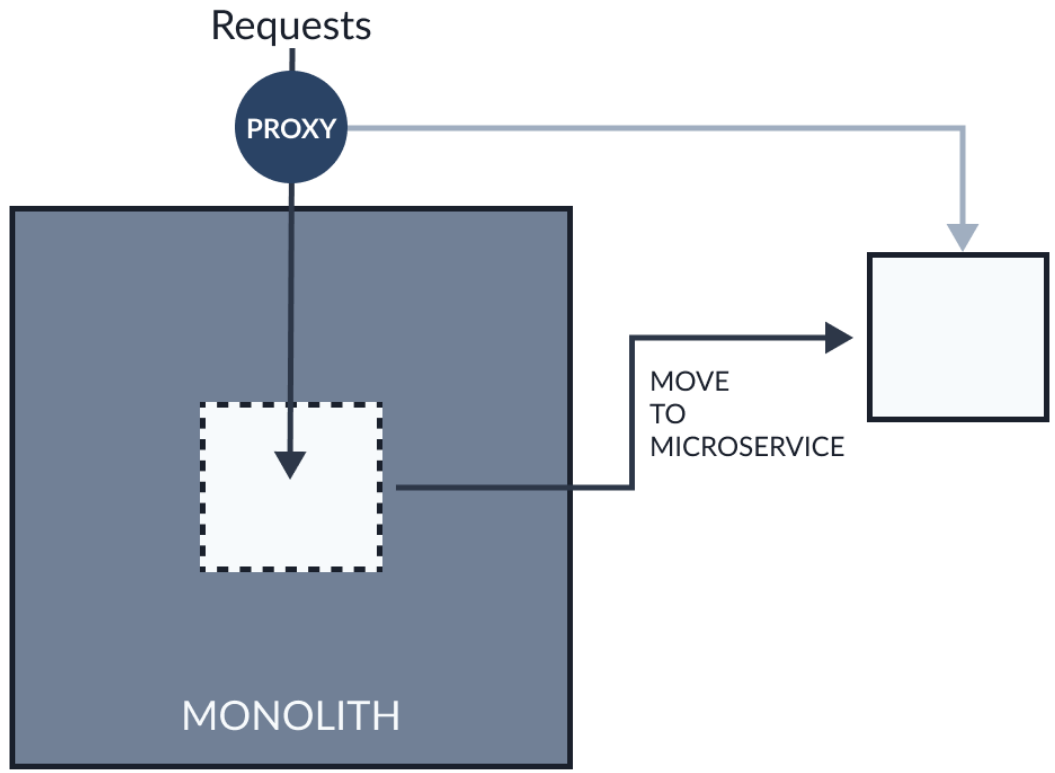
1. Identify the functionality
2. Move the functionality
3. Redirect the calls

IDENTIFY



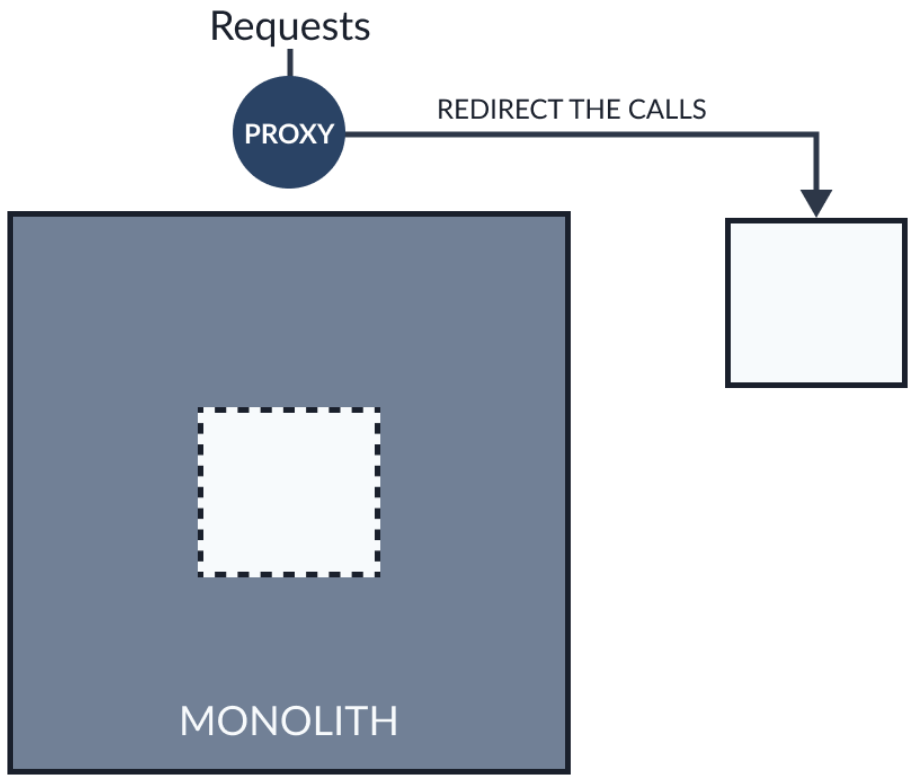
1

MOVE

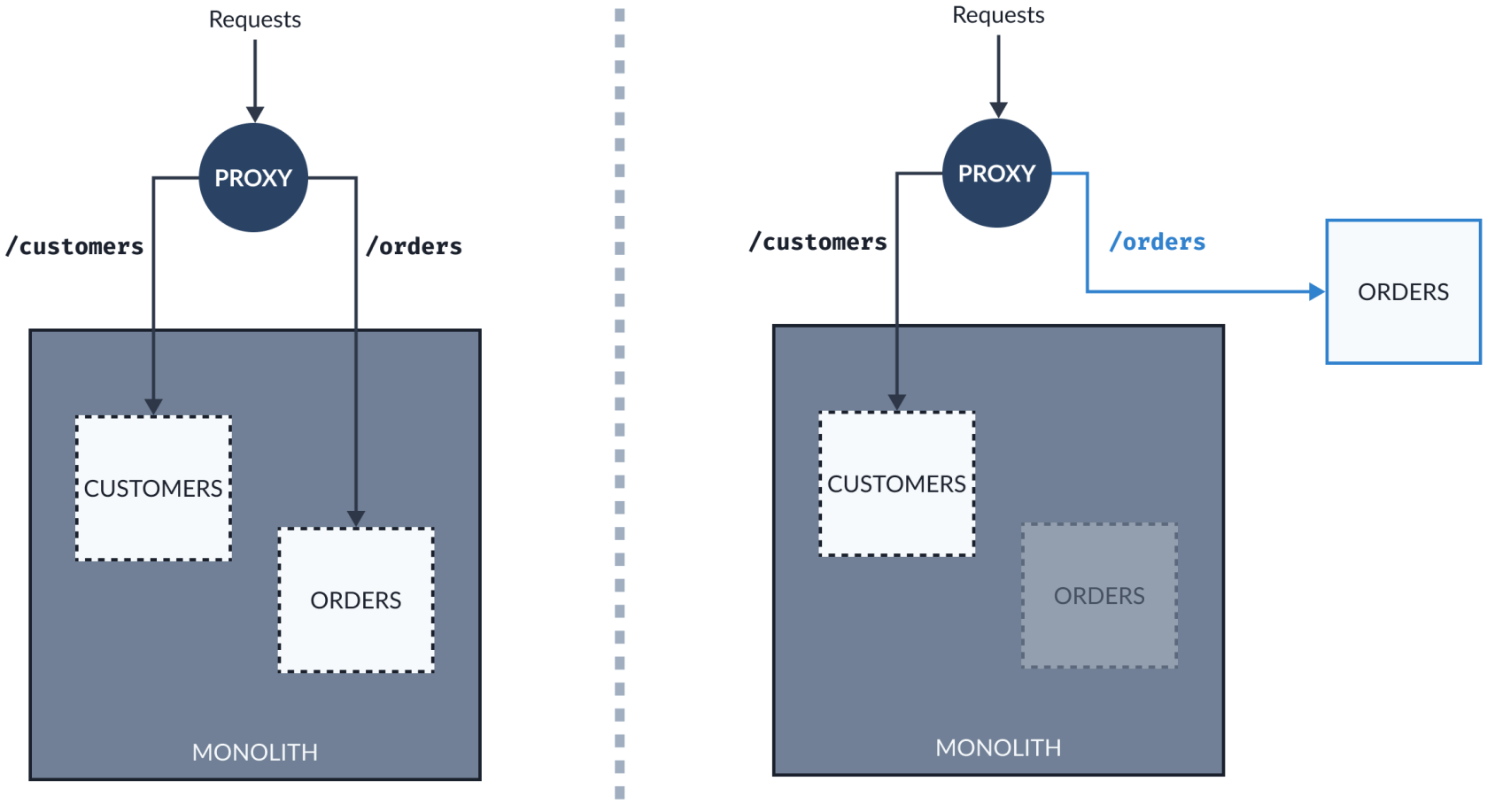


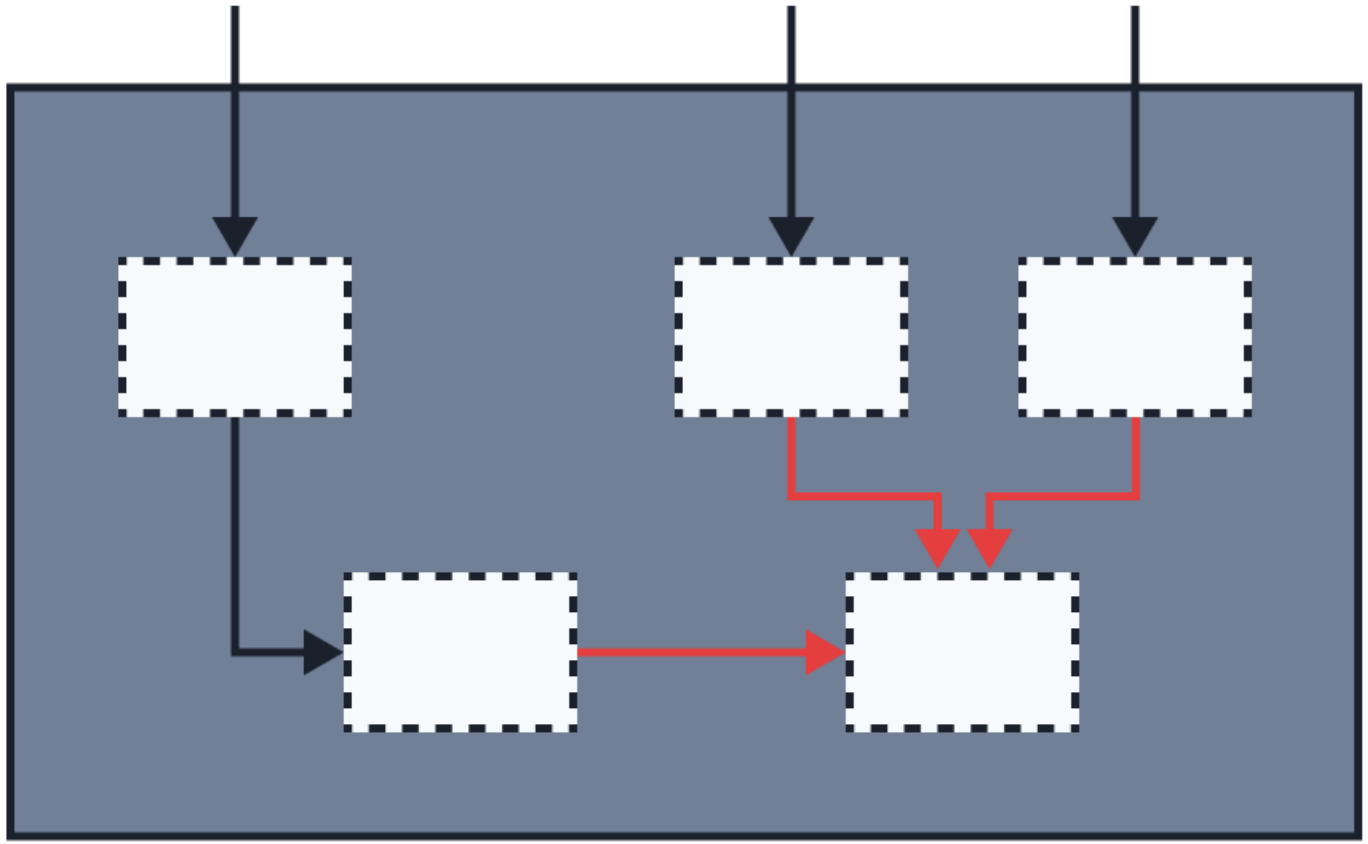
2

REDIRECT



3

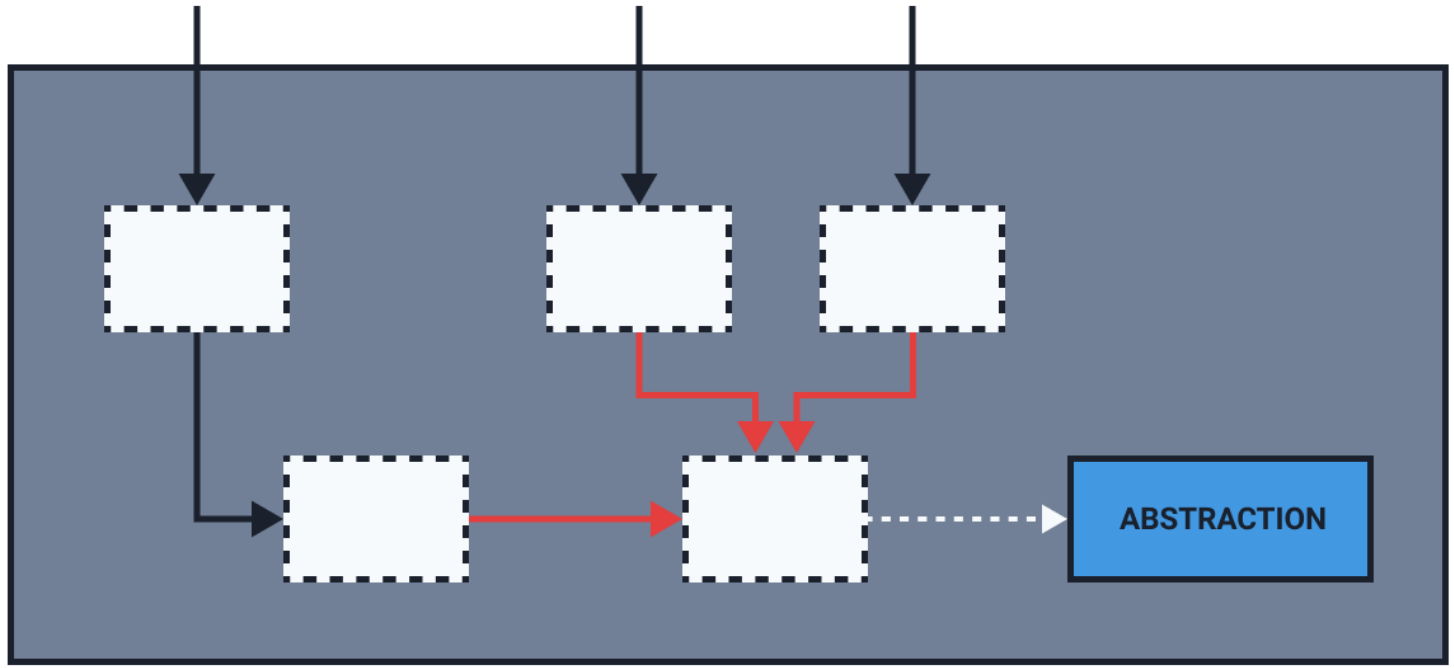

Requests
PROXY
/customers
/orders
CUSTOMERS
ORDERS
MONOLITH
Requests
PROXY
/customers
/orders
ORDERS
CUSTOMERS
ORDERS
MONOLITH



Branch by Abstraction Pattern

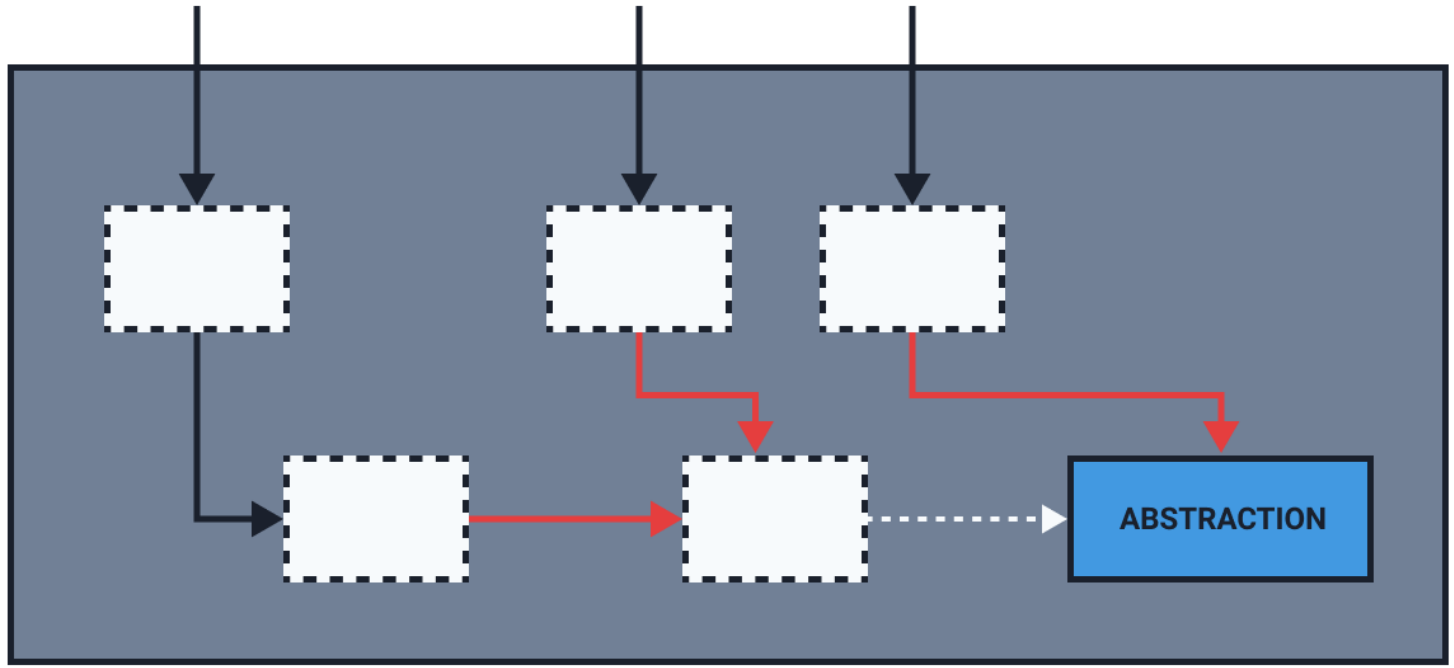
1. Create the abstraction
2. Use the abstraction (with existing implementation)
3. Implement new service
4. Switch the implementation
5. Clean up

CREATE



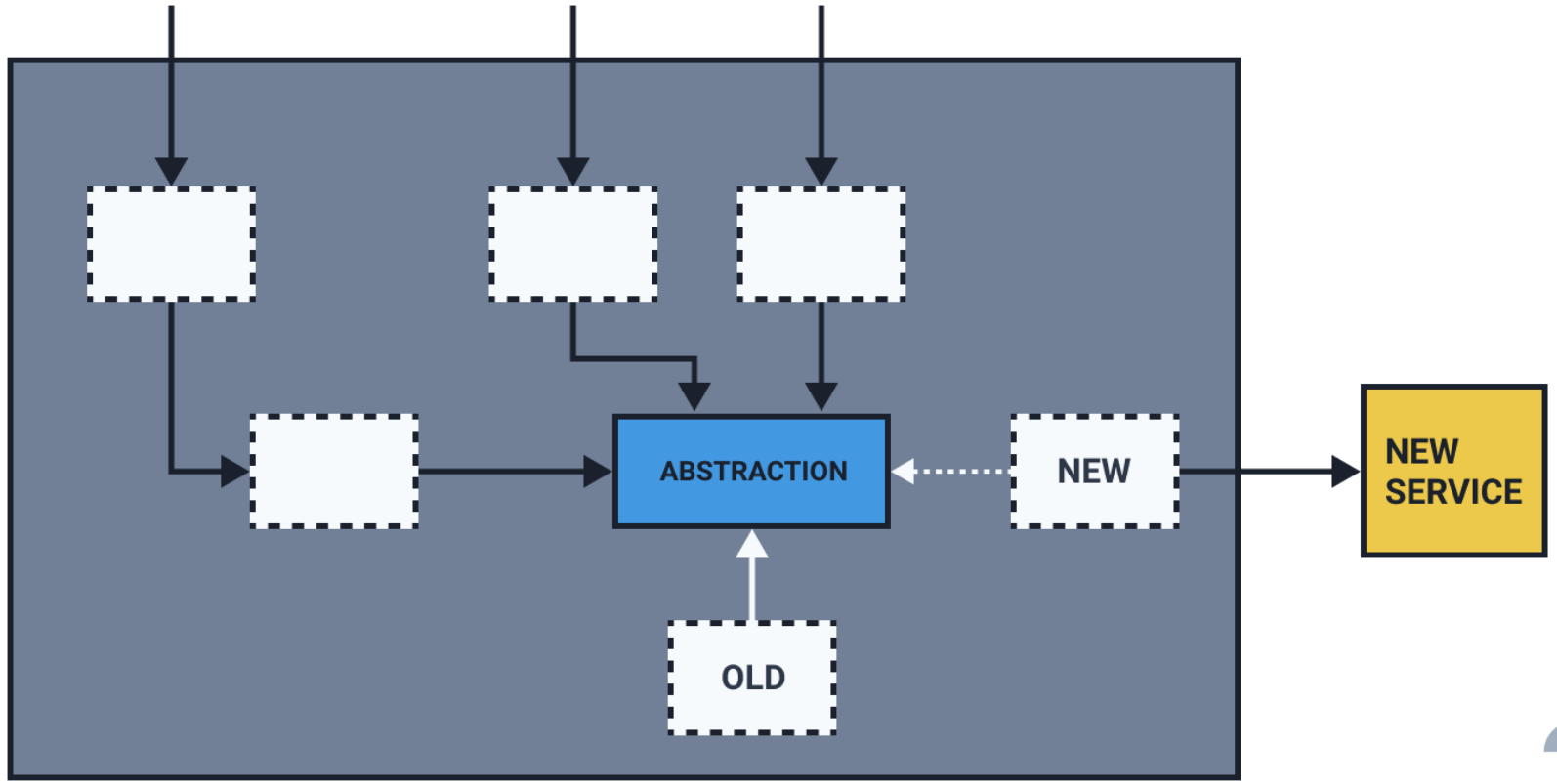
1

USE



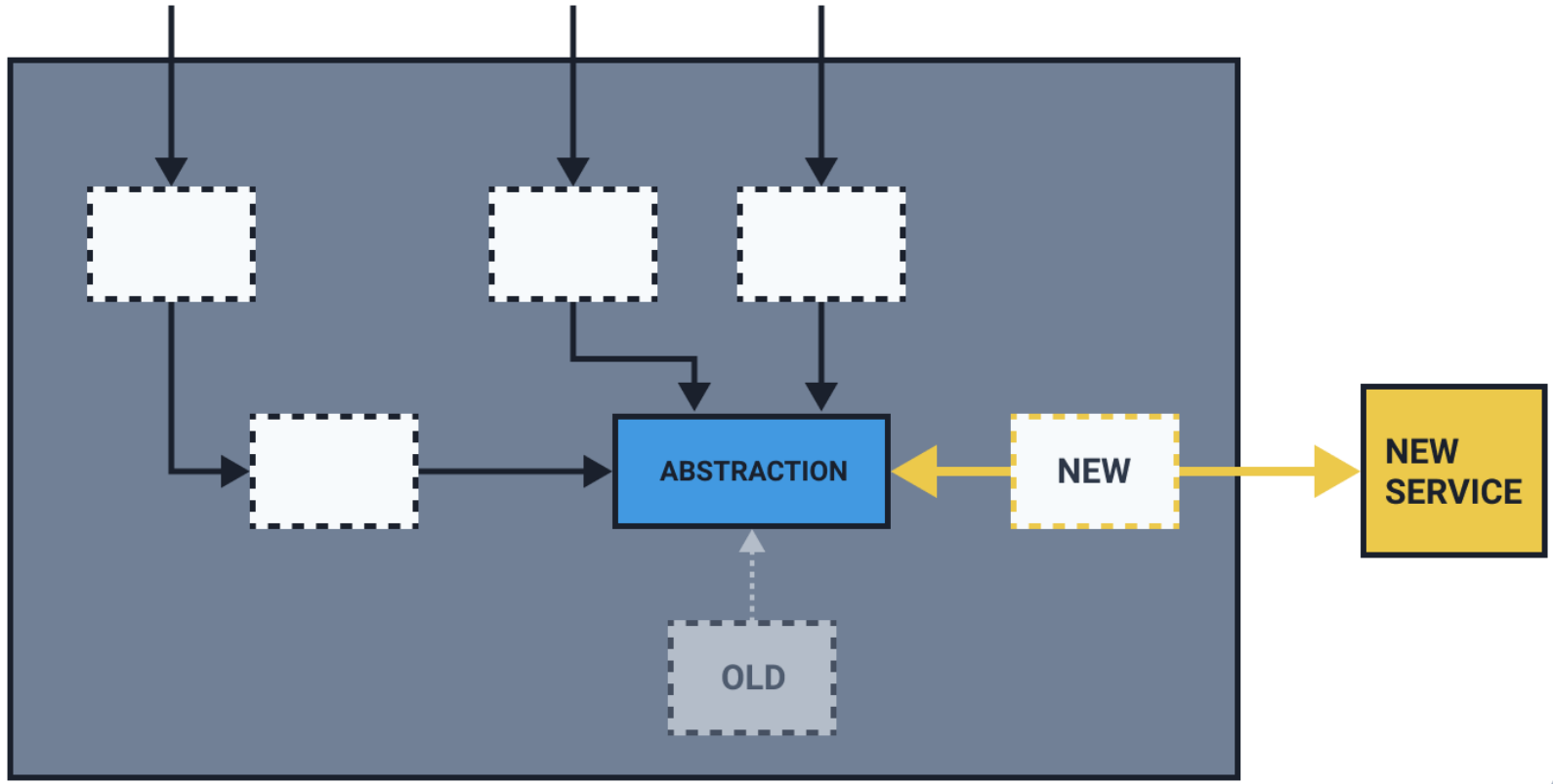
2

IMPLEMENT



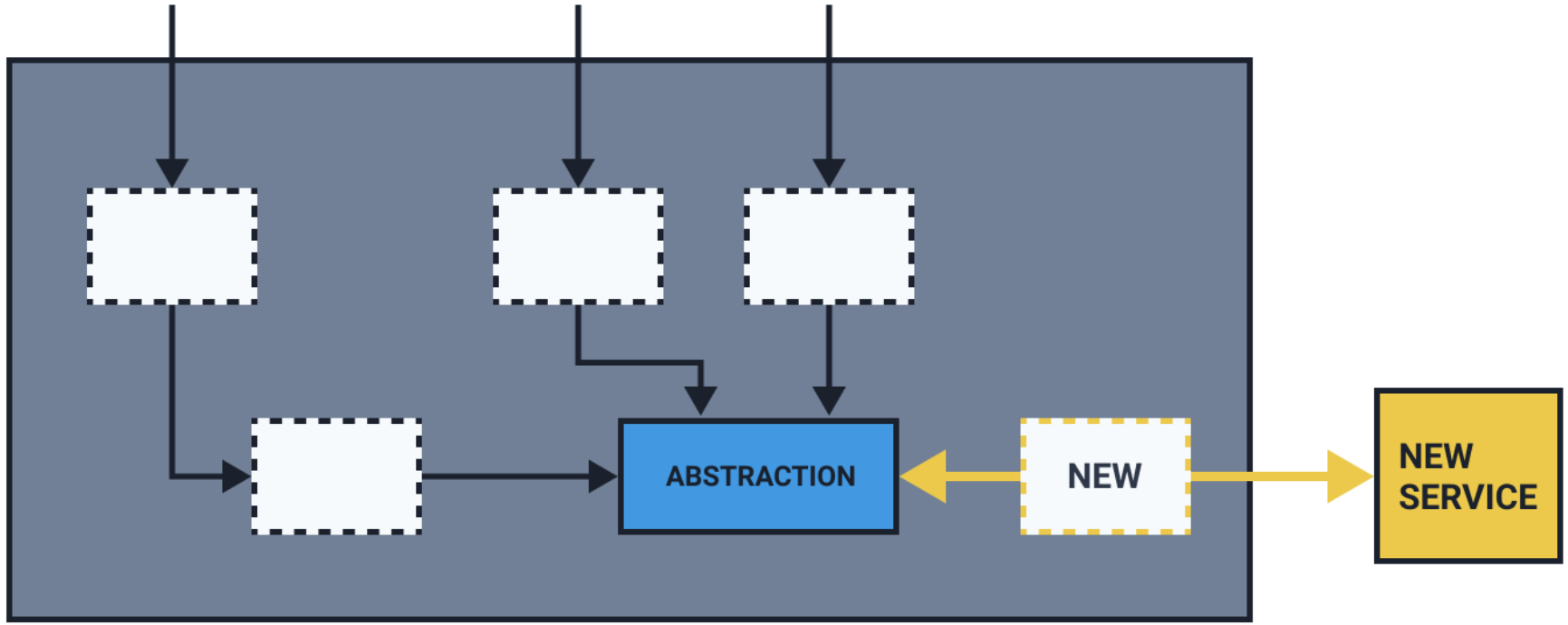
3

SWITCH



4

CLEAN UP

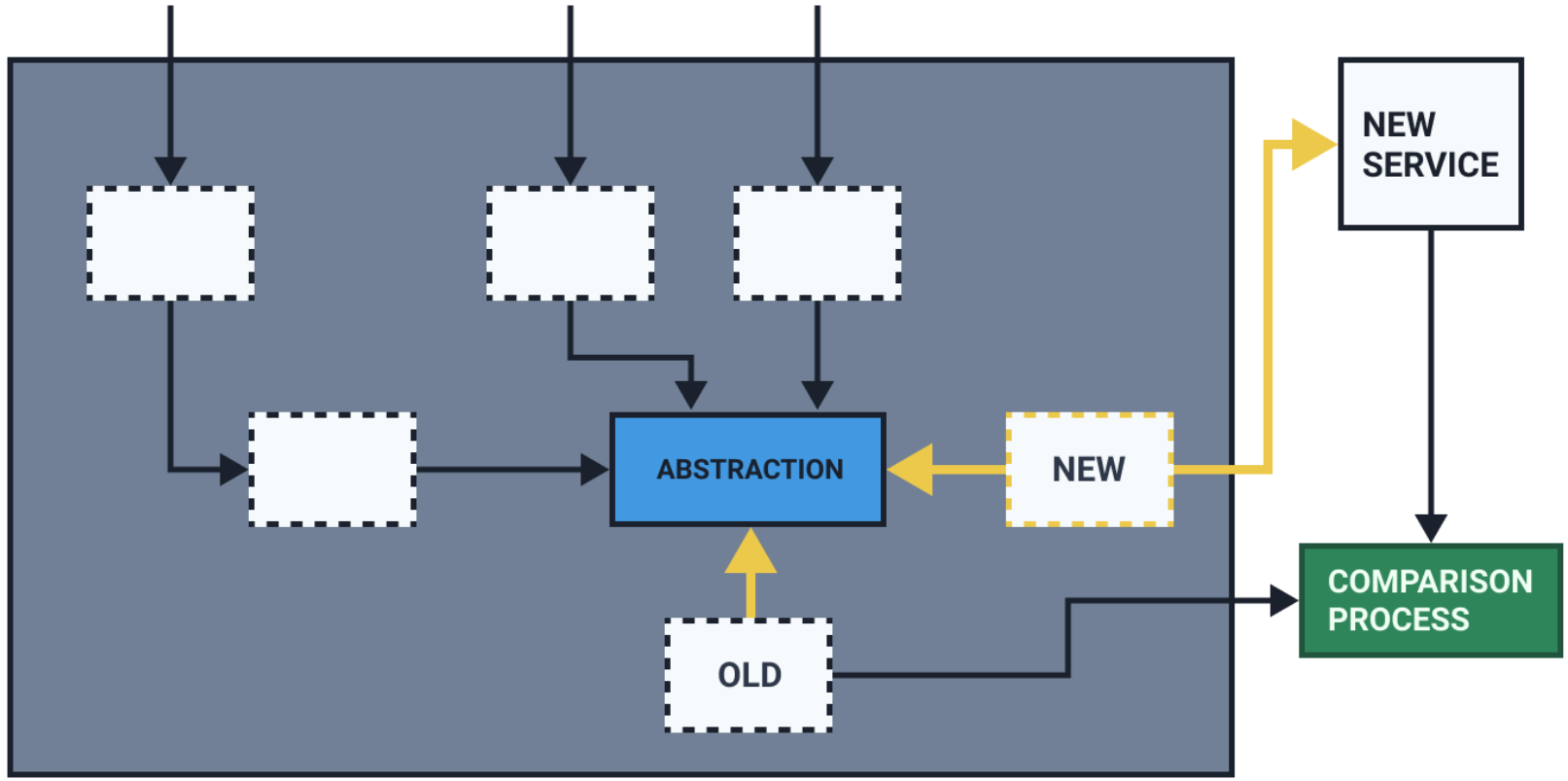


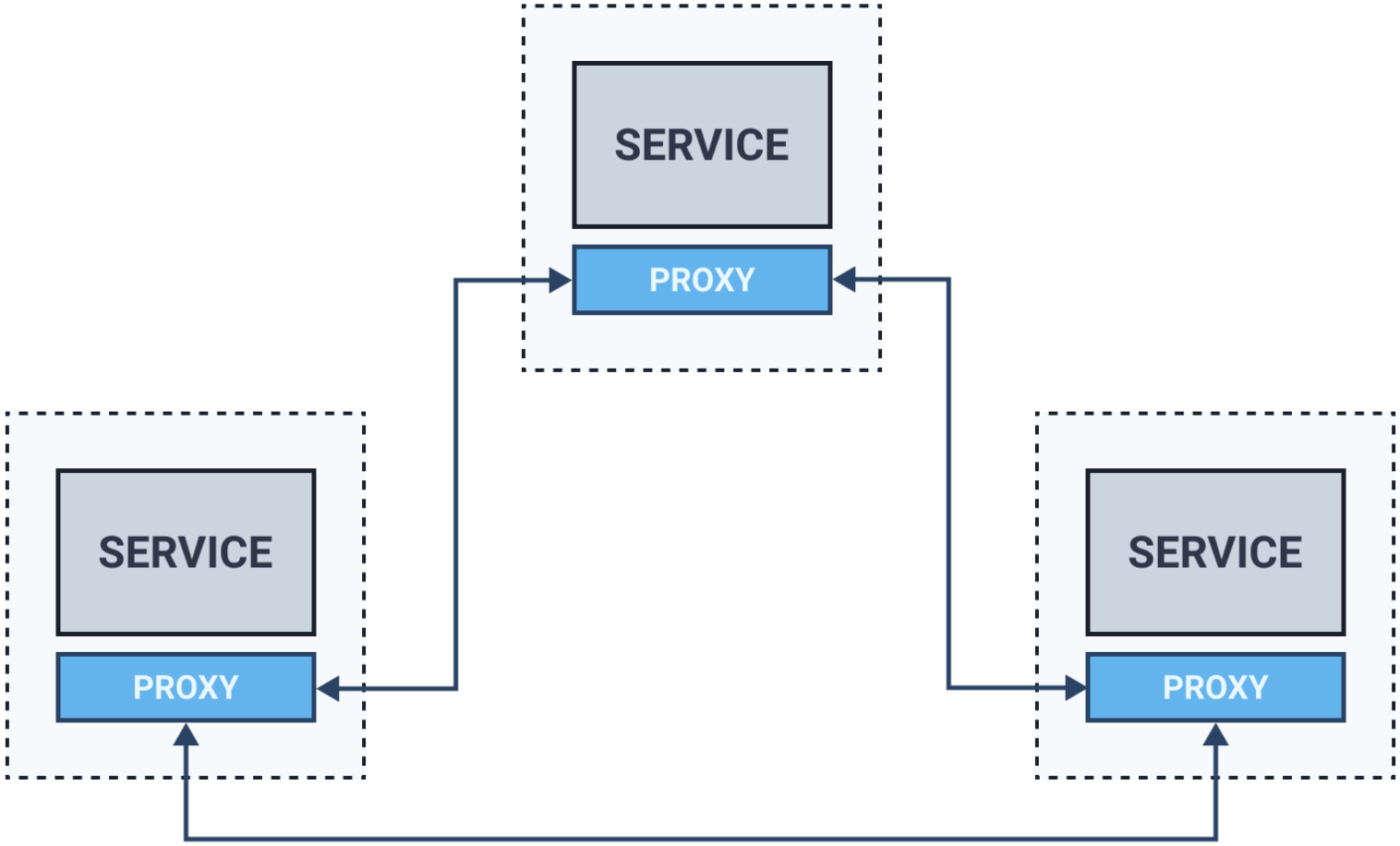
5

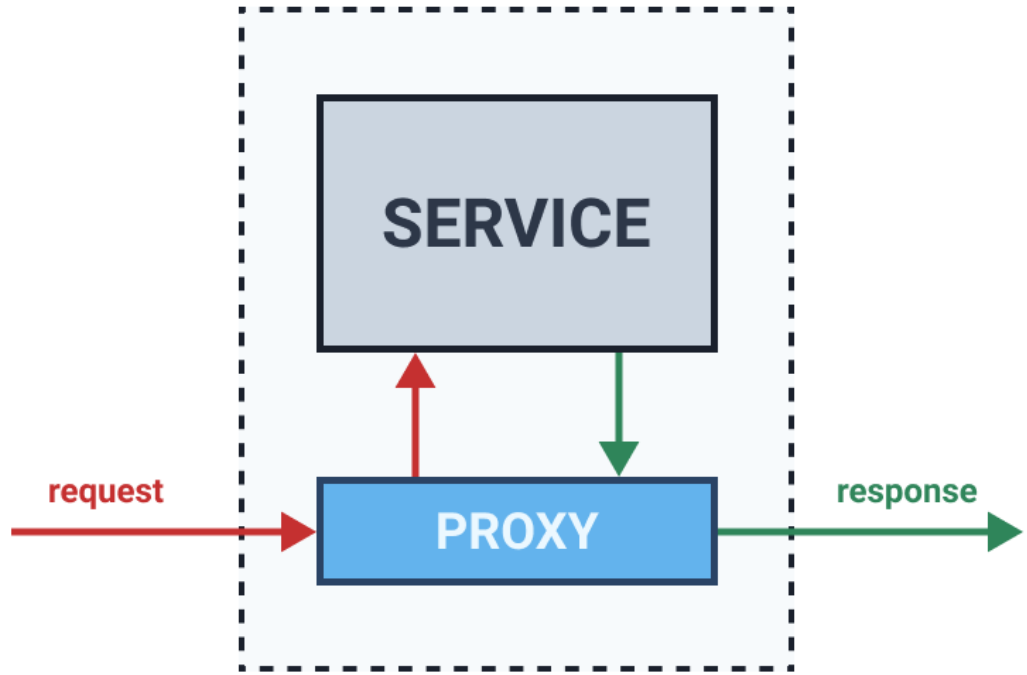
DEMO - Branch by Abstraction

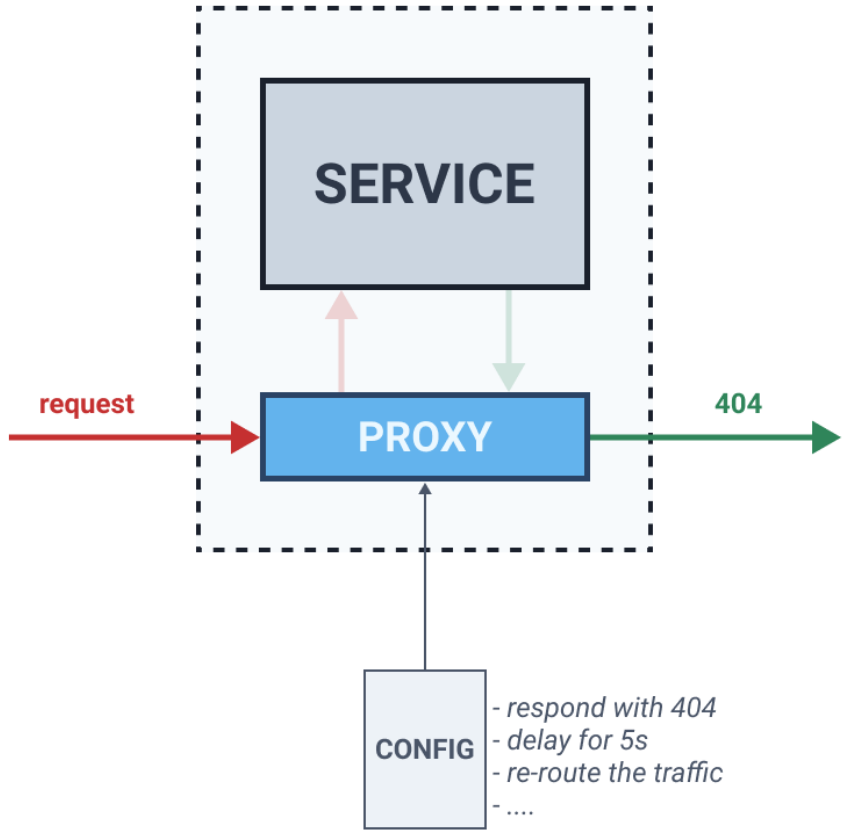
BREAK

PARALLEL RUN



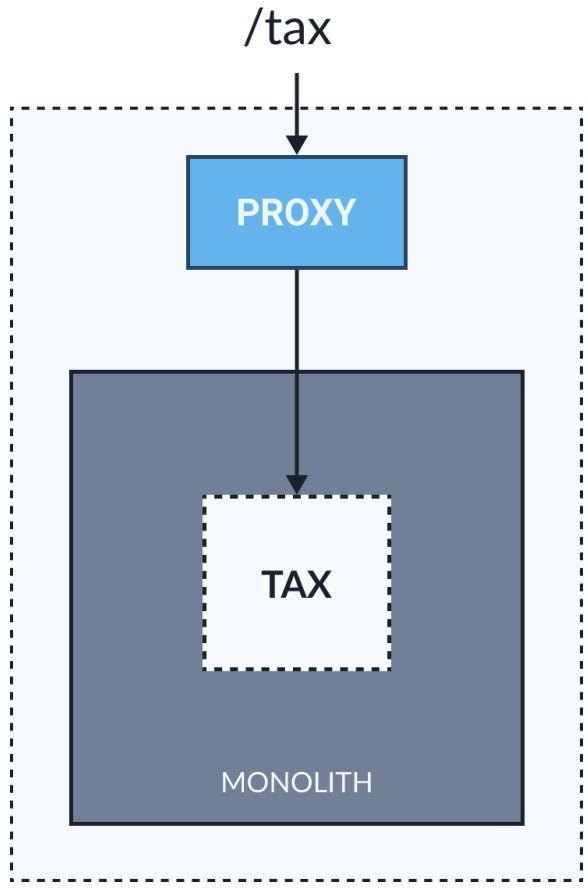

SERVICE
PROXY
SERVICE
PROXY
SERVICE
PROXY

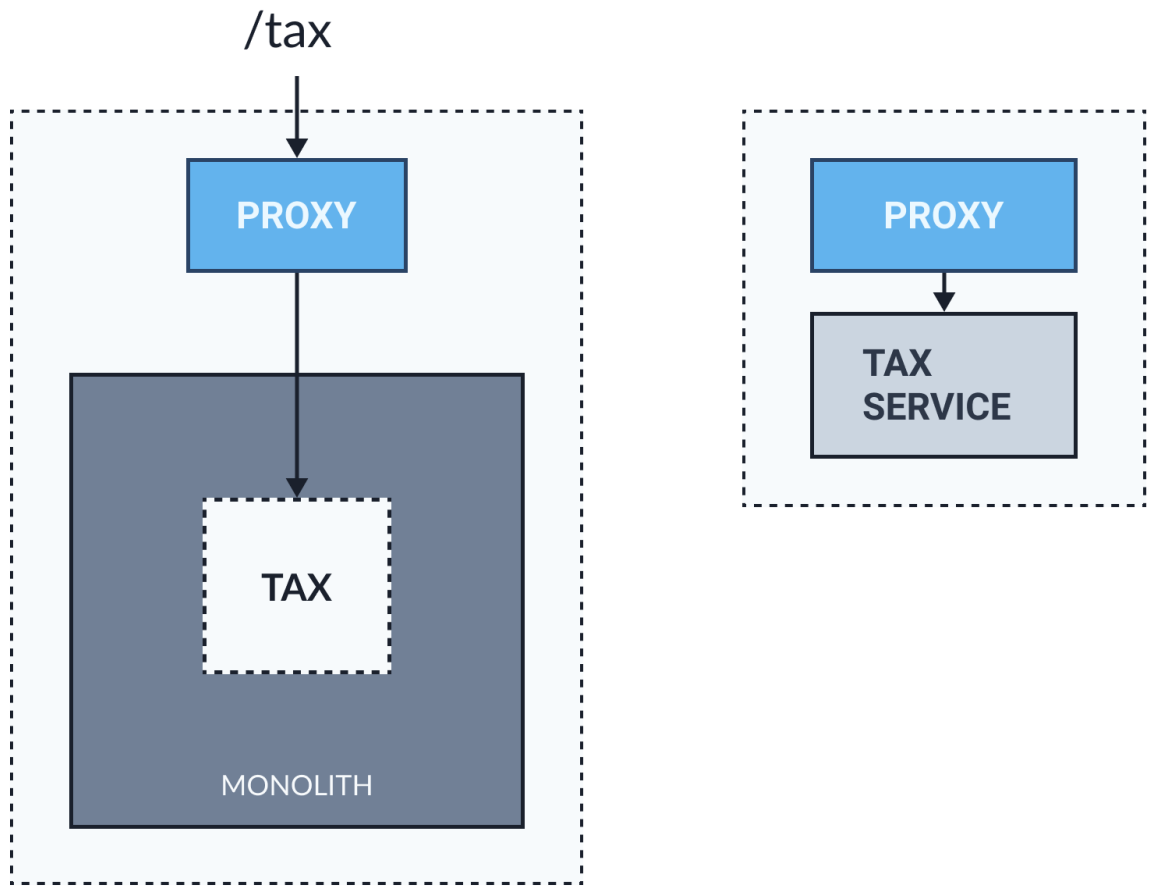

SERVICE
request
PROXY
response

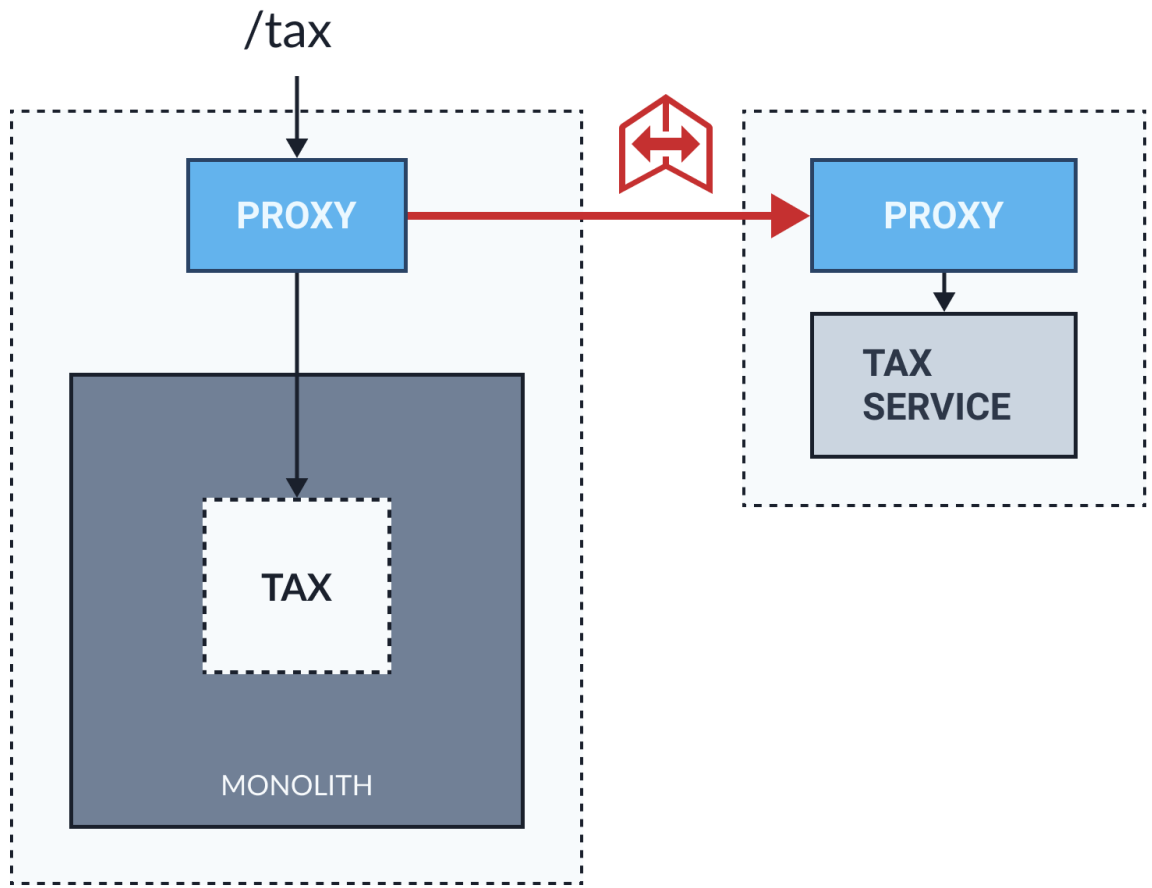

SERVICE
request
PROXY
404
CONFIG
- respond with 404
- delay for 5s
- re-route the traffic
- ....

DEMO

Parallel Run/Mirroring with Service Mesh


/tax
PROXY
TAX
MONOLITH


/tax
PROXY
TAX
MONOLITH
PROXY
TAX
SERVICE

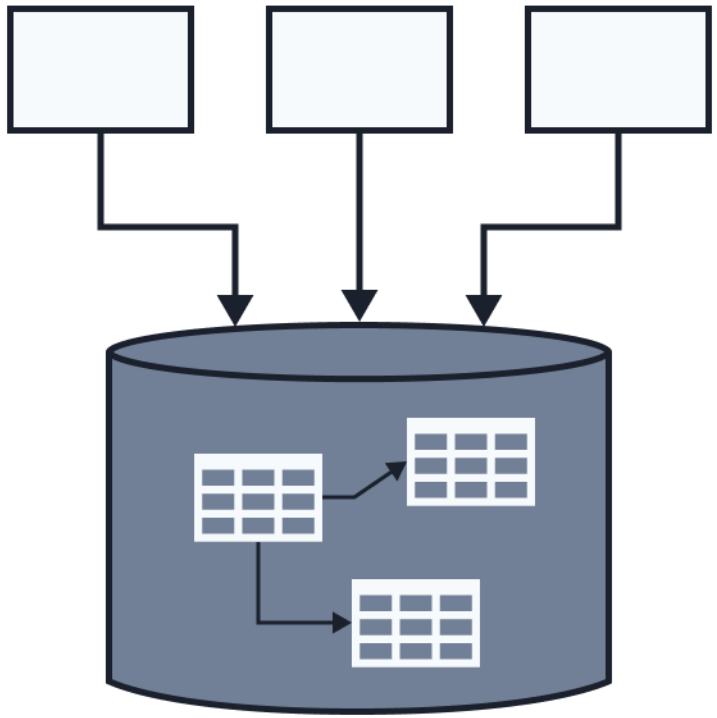

/tax
PROXY
PROXY
TAX
SERVICE
TAX
MONOLITH

Decomposing Databases

Sharing a Database

- You can't decide what's shared and what's hidden
  - Goes against one of the microservices characteristics
- Data control - where is the logic?

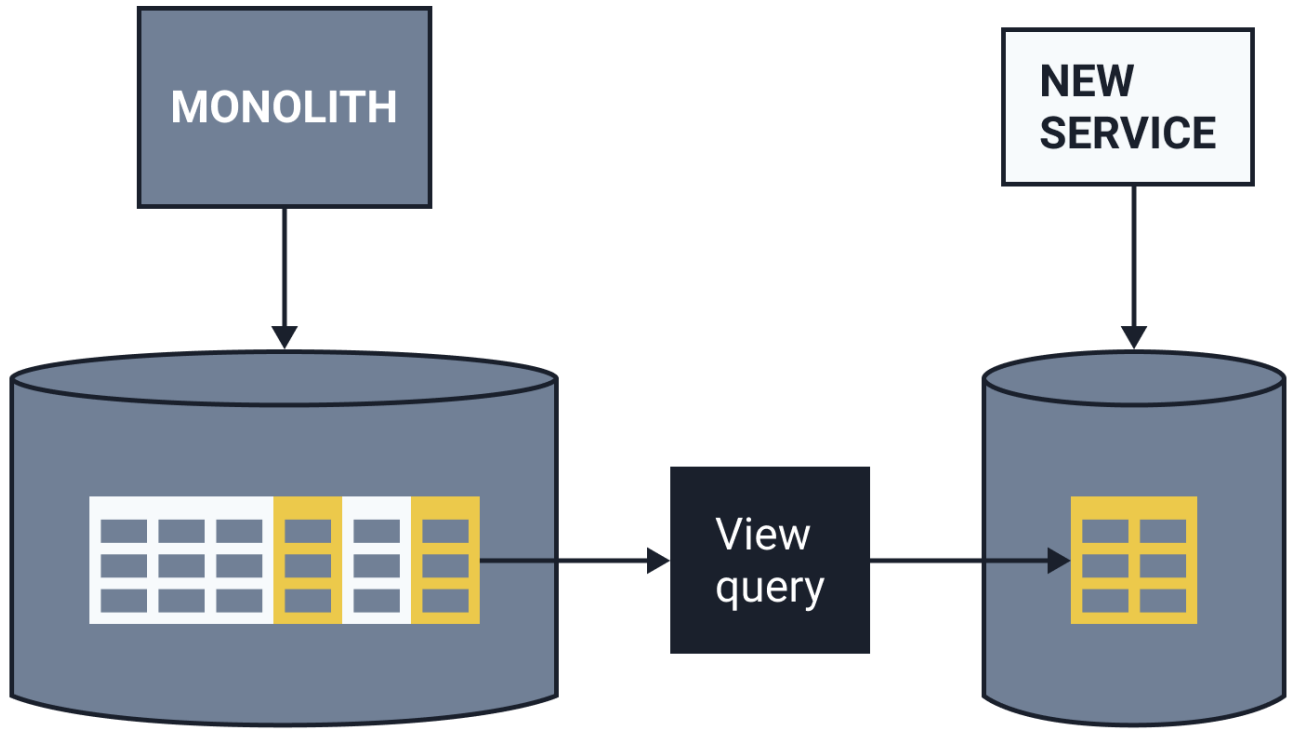
SHARED DATABASE



1. Database holds read-only/static data

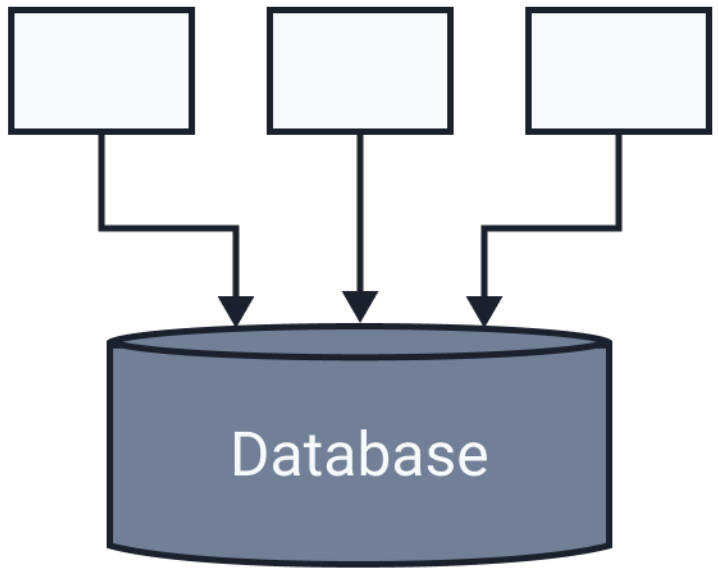
2. Database-as-a-Service Interface pattern

DATABASE VIEW PATTERN



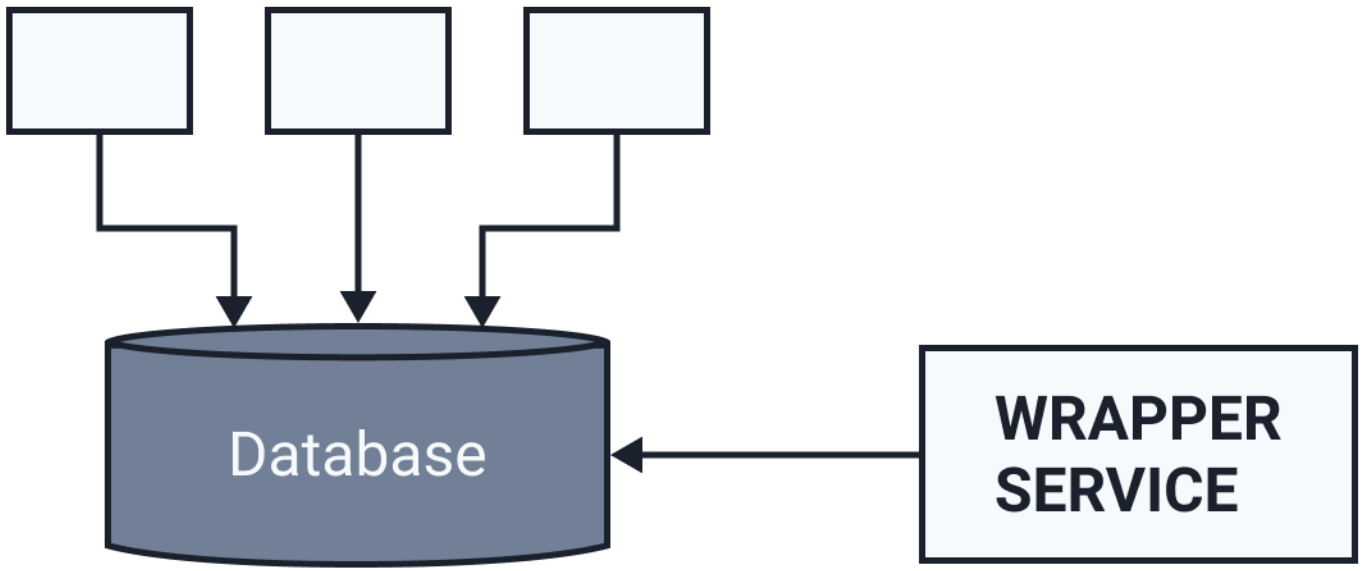

BIG DUO MAIL
SHANNON PJ
EK
DELIVERS
KEYWORDS / CUST REVS
STAR WARS ORDER
SCARLET
BBC
JUMP SHARE
ACORN
TV SERIES
SUN D&GS

DATABASE WRAPPING



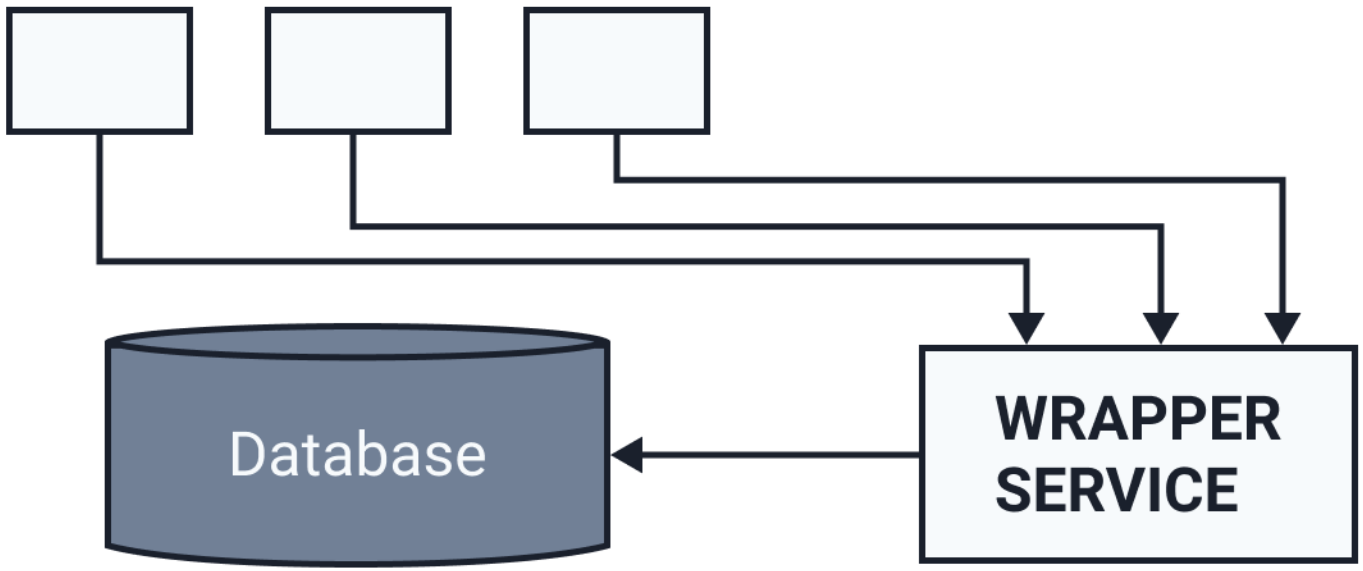
1

DATABASE WRAPPING



2

DATABASE WRAPPING

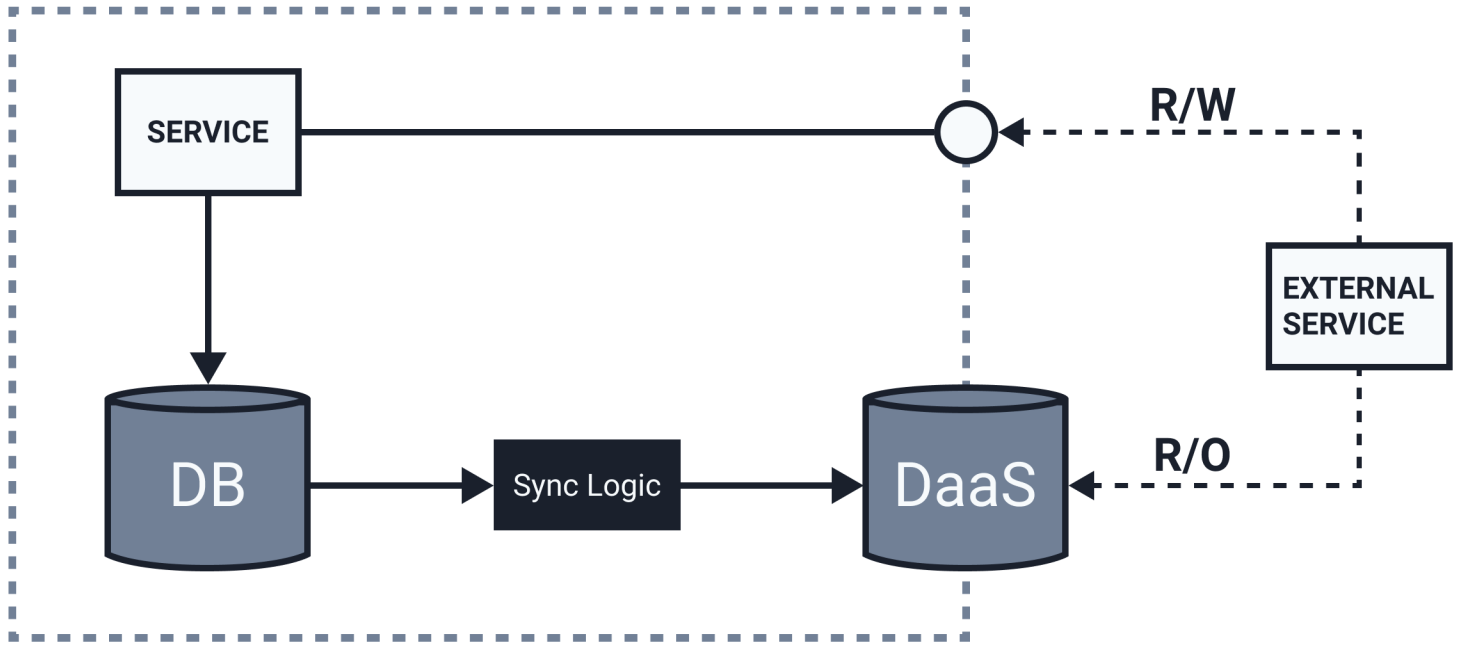


3

Wrapper vs. View

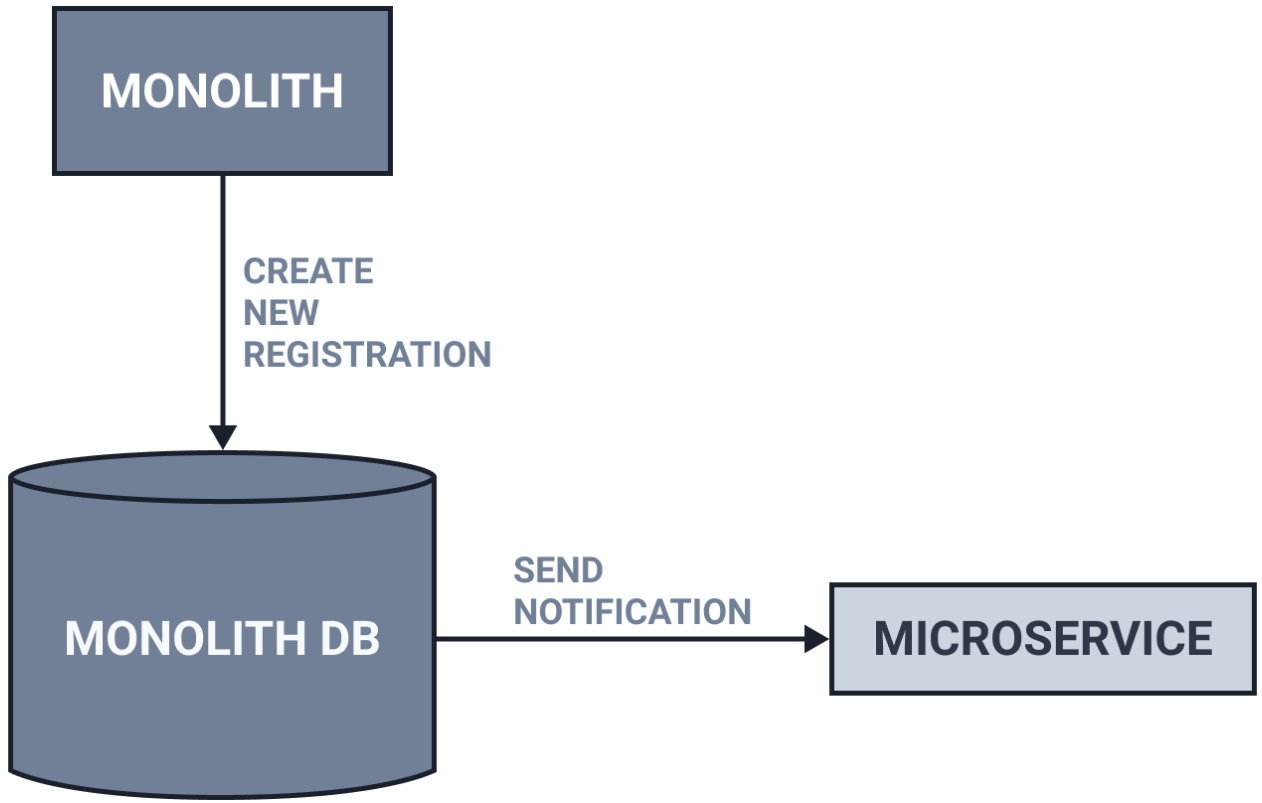
- Not constrained to a view
- Ability to create more complex views
- API for writing
  - requires changes to upstream services

DATABASE-AS-A-SERVICE



Updating the R/O database

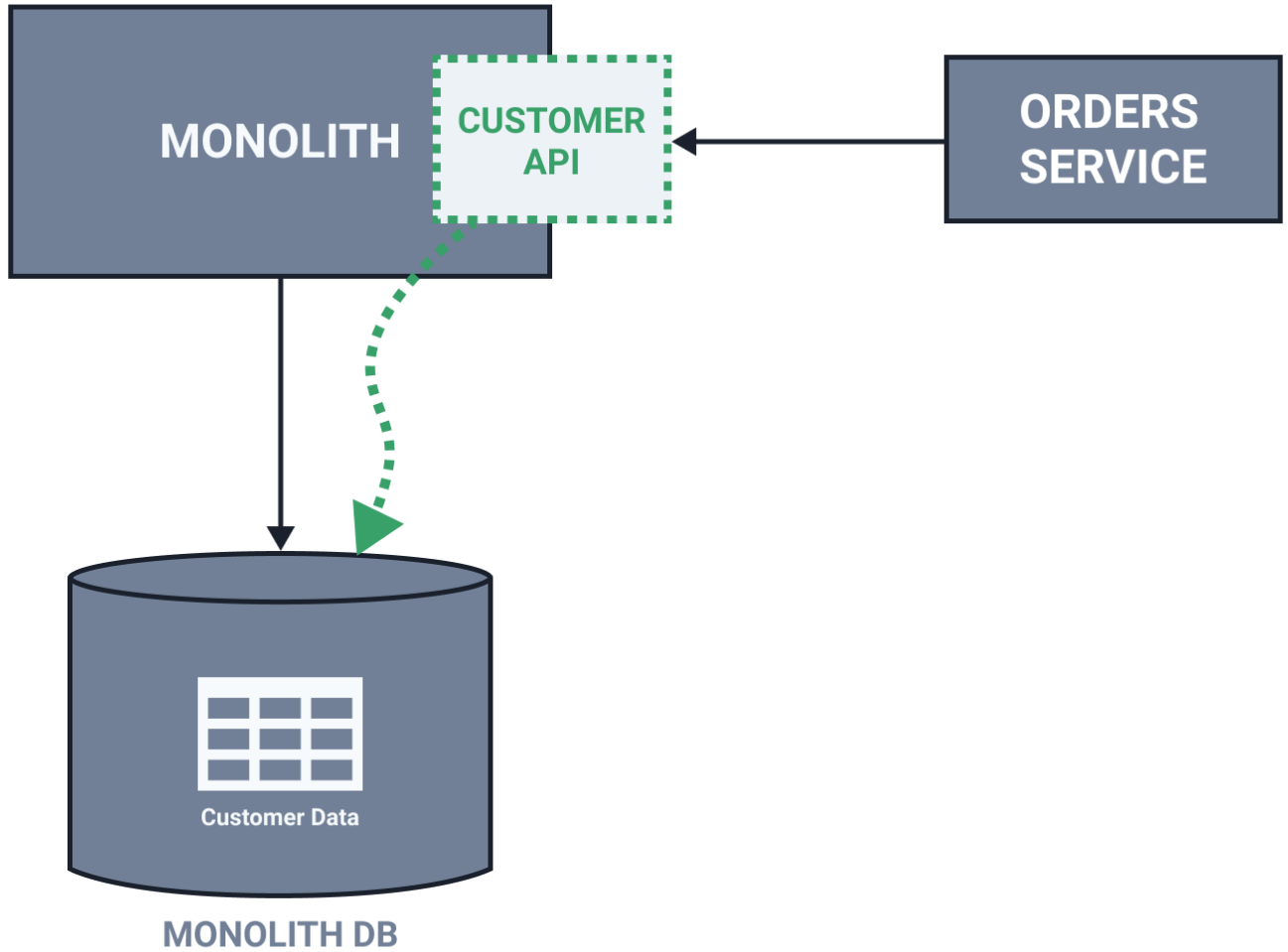
- Change data capture
- Batch process
- Service events

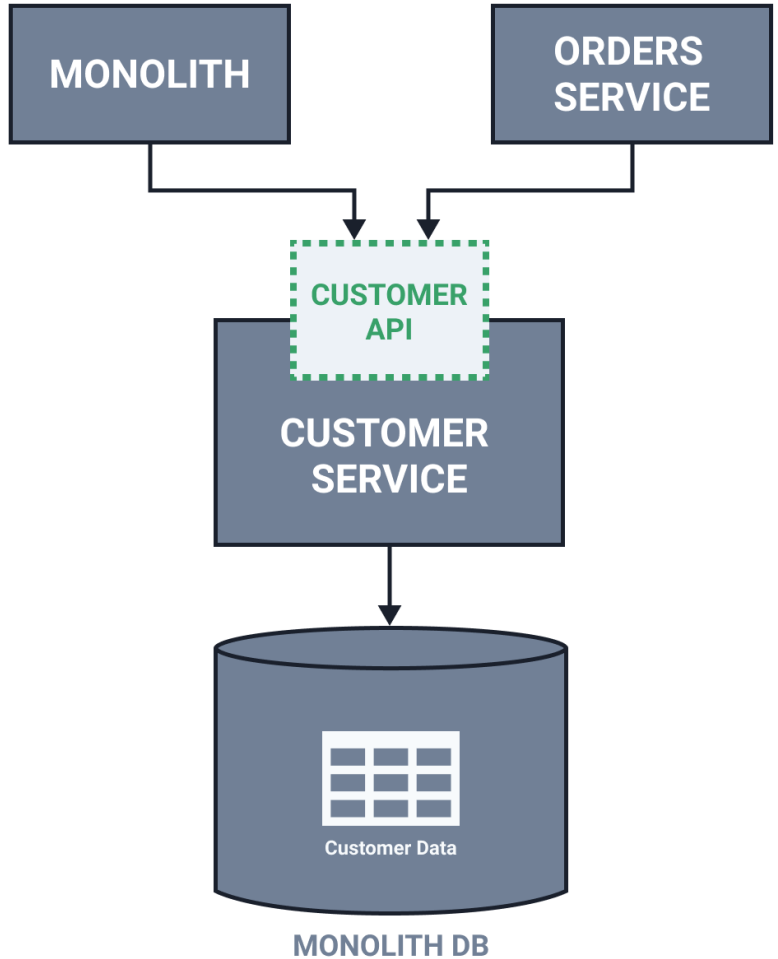

MONOLITH
CREATE
NEW
REGISTRATION
MONOLITH DB
SEND
NOTIFICATION
MICROSERVICE

Change Data Capture

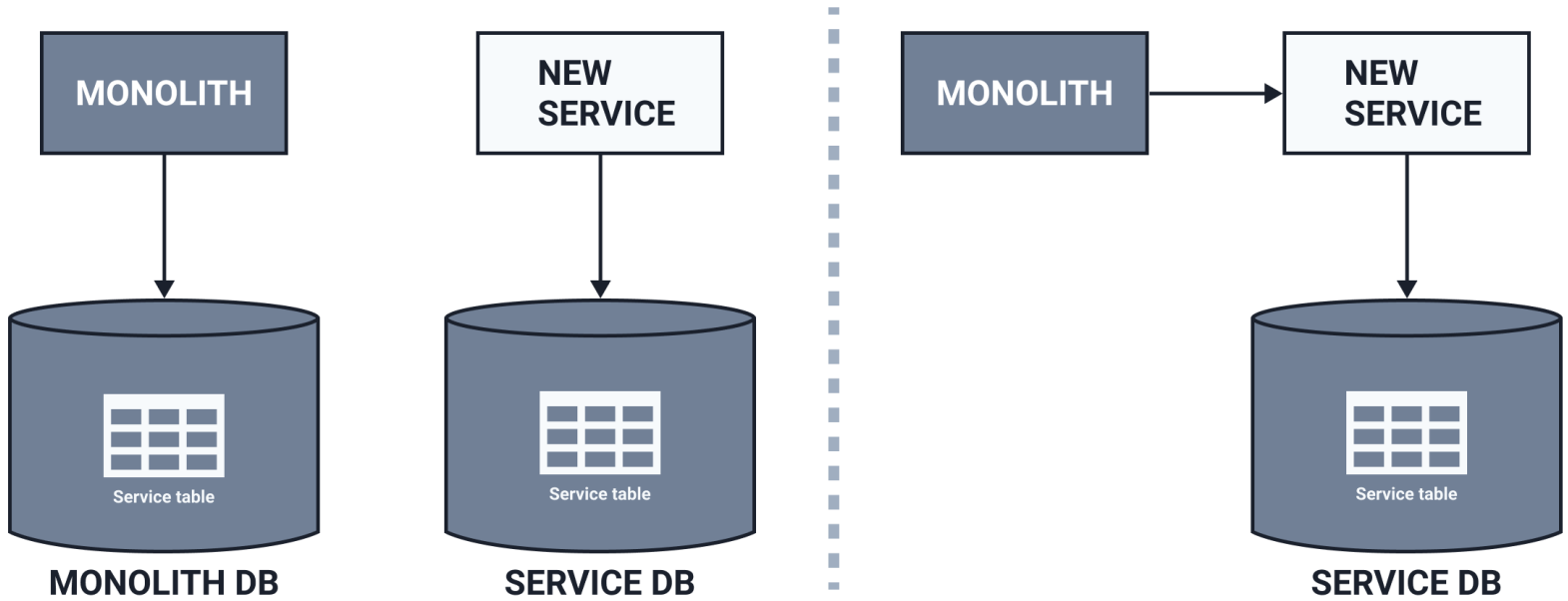
- Database triggers
  - trigger behavior on data changes
- Transaction log pollers
- Batch delta copier

Aggregate Exposing Monolith

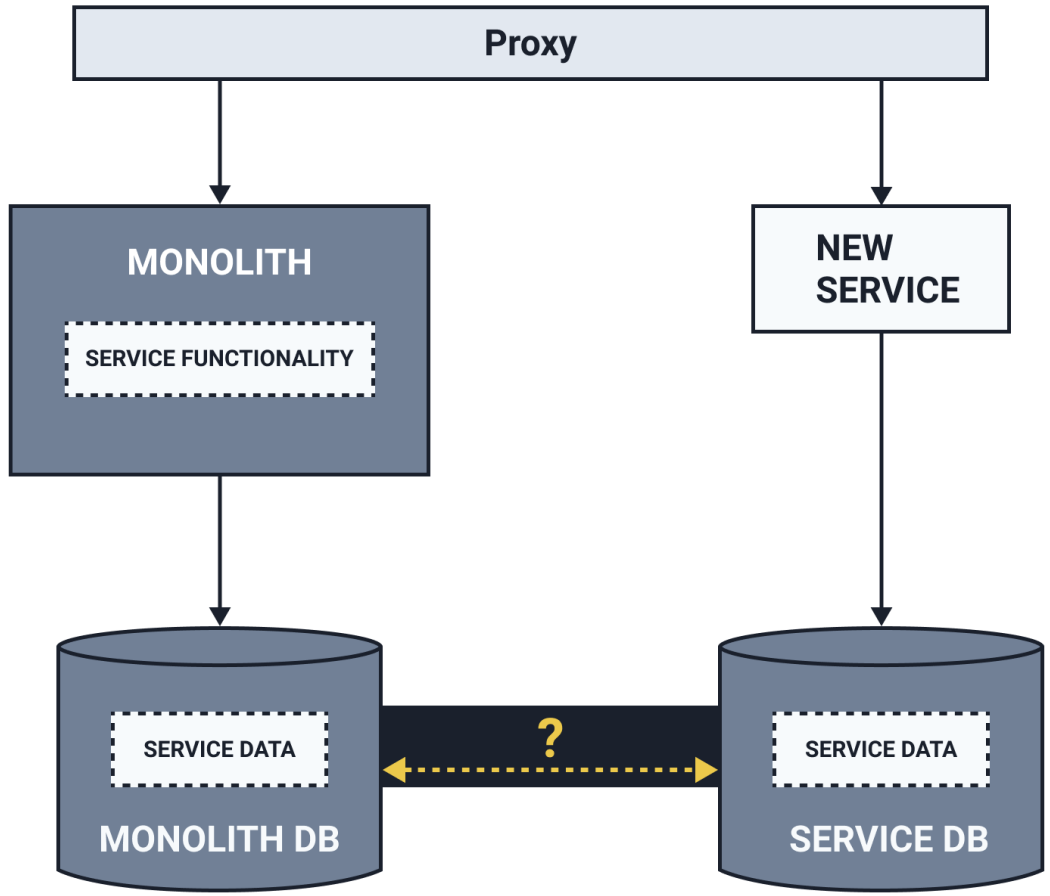

MONOLITH
CUSTOMER API
ORDERS SERVICE
Customer Data
MONOLITH DB


MONOLITH
ORDERS SERVICE
CUSTOMER API
CUSTOMER SERVICE
Customer Data
MONOLITH DB

Change Data Ownership Pattern


MONOLITH
NEW SERVICE
Service table
Service table
MONOLITH DB
SERVICE DB
MONOLITH
NEW SERVICE
Service table
SERVICE DB

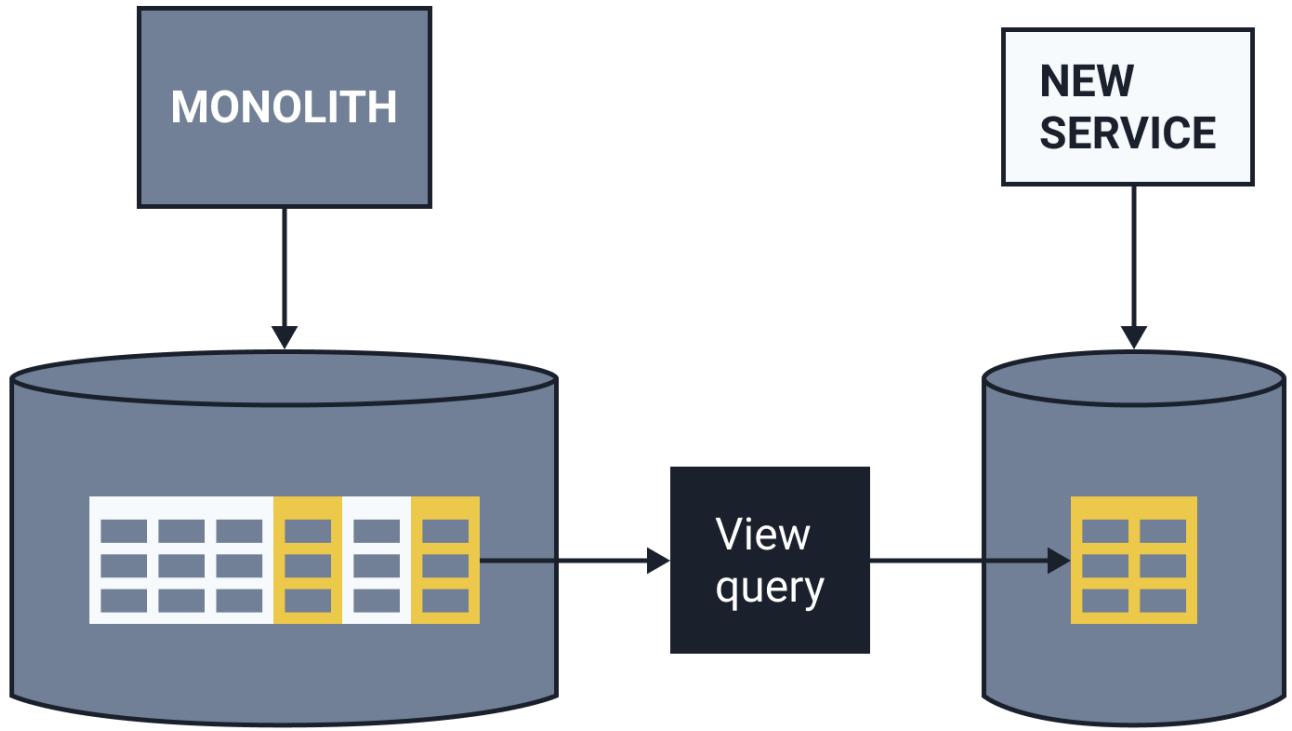
Data Sychronization


Proxy
MONOLITH
SERVICE FUNCTIONALITY
NEW SERVICE
SERVICE DATA
MONOLITH DB
?
SERVICE DATA
SERVICE DB

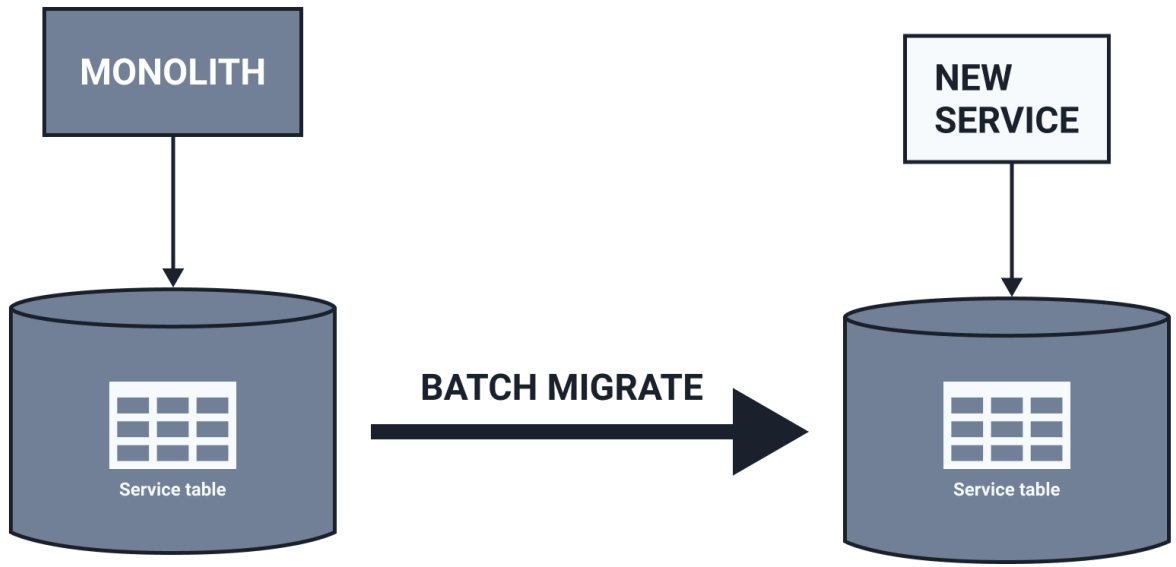
What degree of consistency we need?

1. Keep data in one place
2. Batch copy all the data
3. Sync via code

DATA IN ONE PLACE



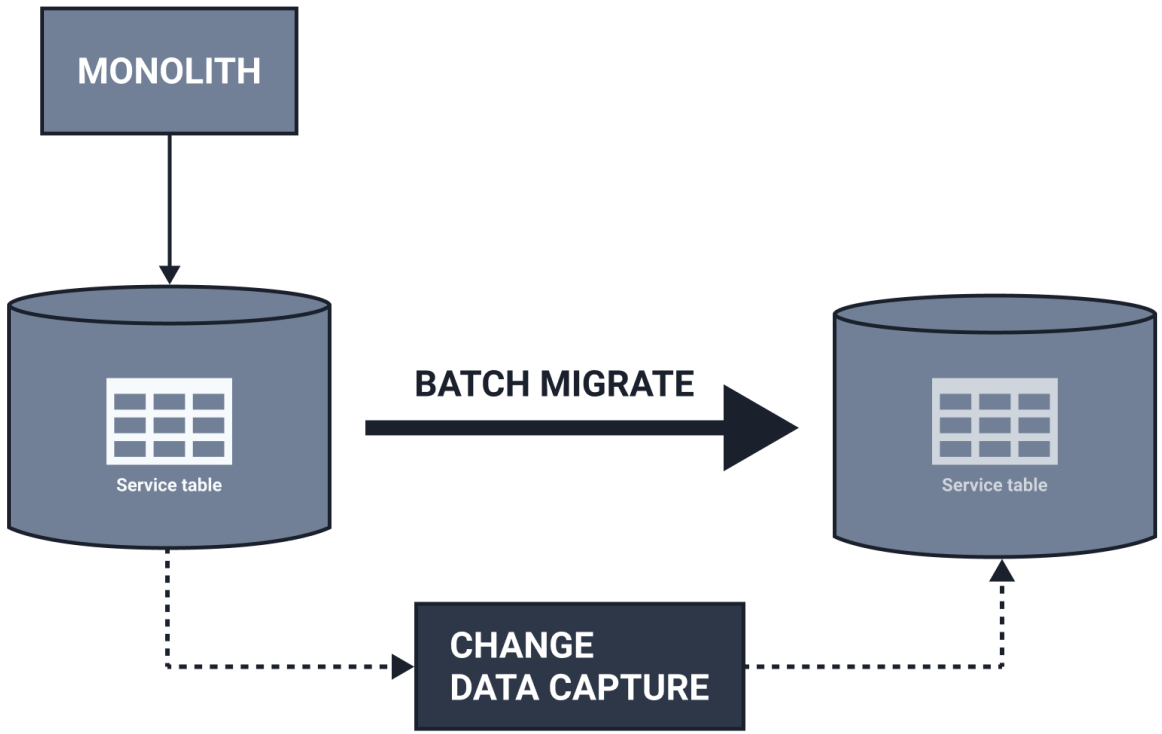
BATCH MIGRATE



Sync via Code

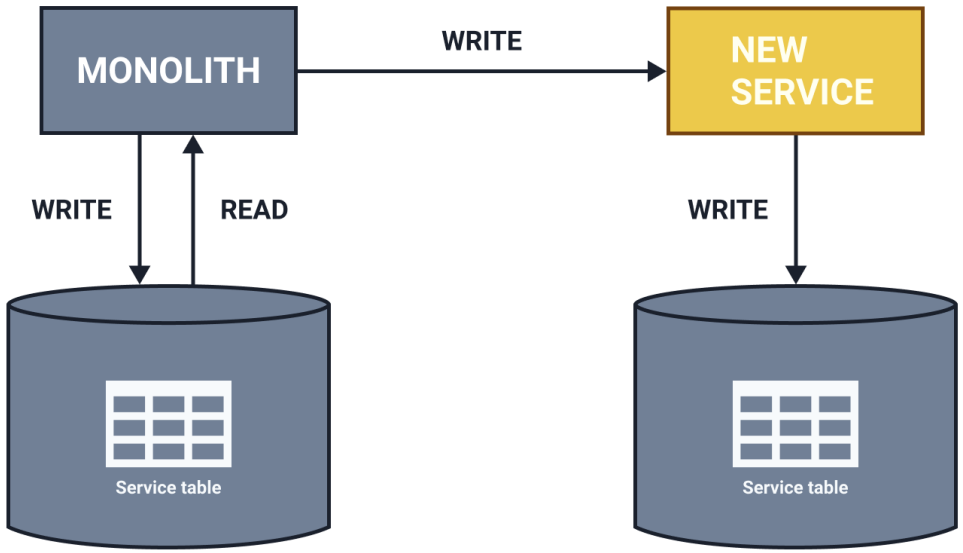
1. Batch migrate the data to new database
2. Deploy the new service (sync on write, read from old)
3. Make the new database the source of truth
4. Verify and remove old DB/schema and switching logic

BATCH MIGRATE



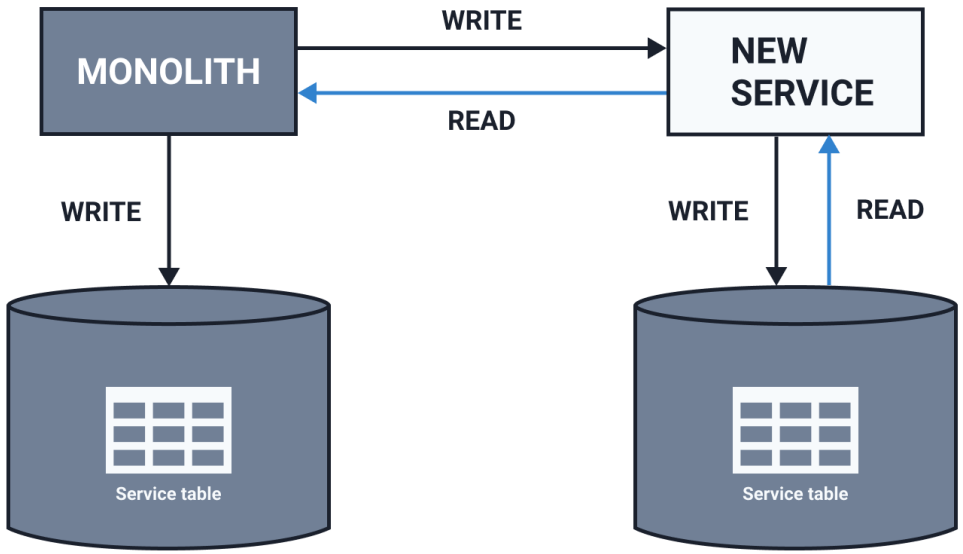
1

WRITE TO BOTH/READ FROM OLD



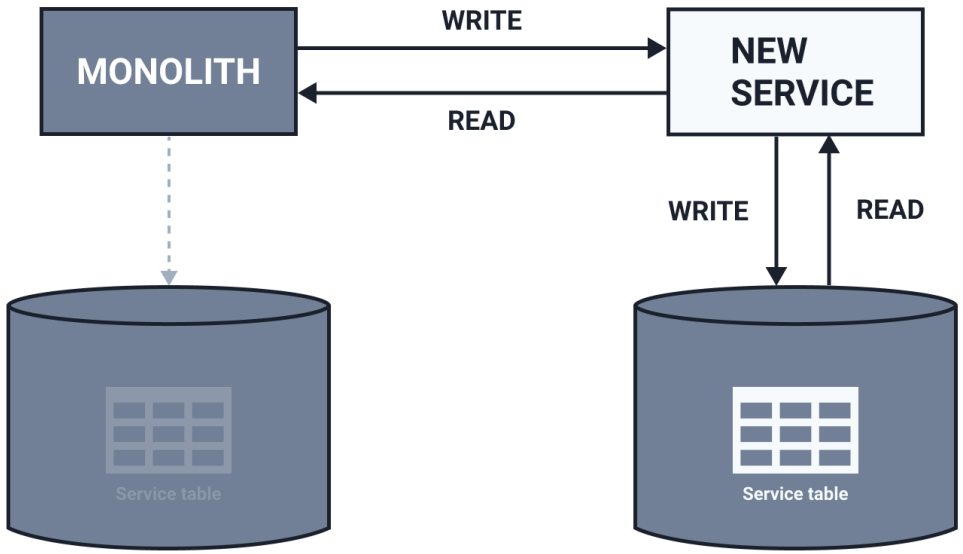
2

WRITE TO BOTH/READ FROM NEW



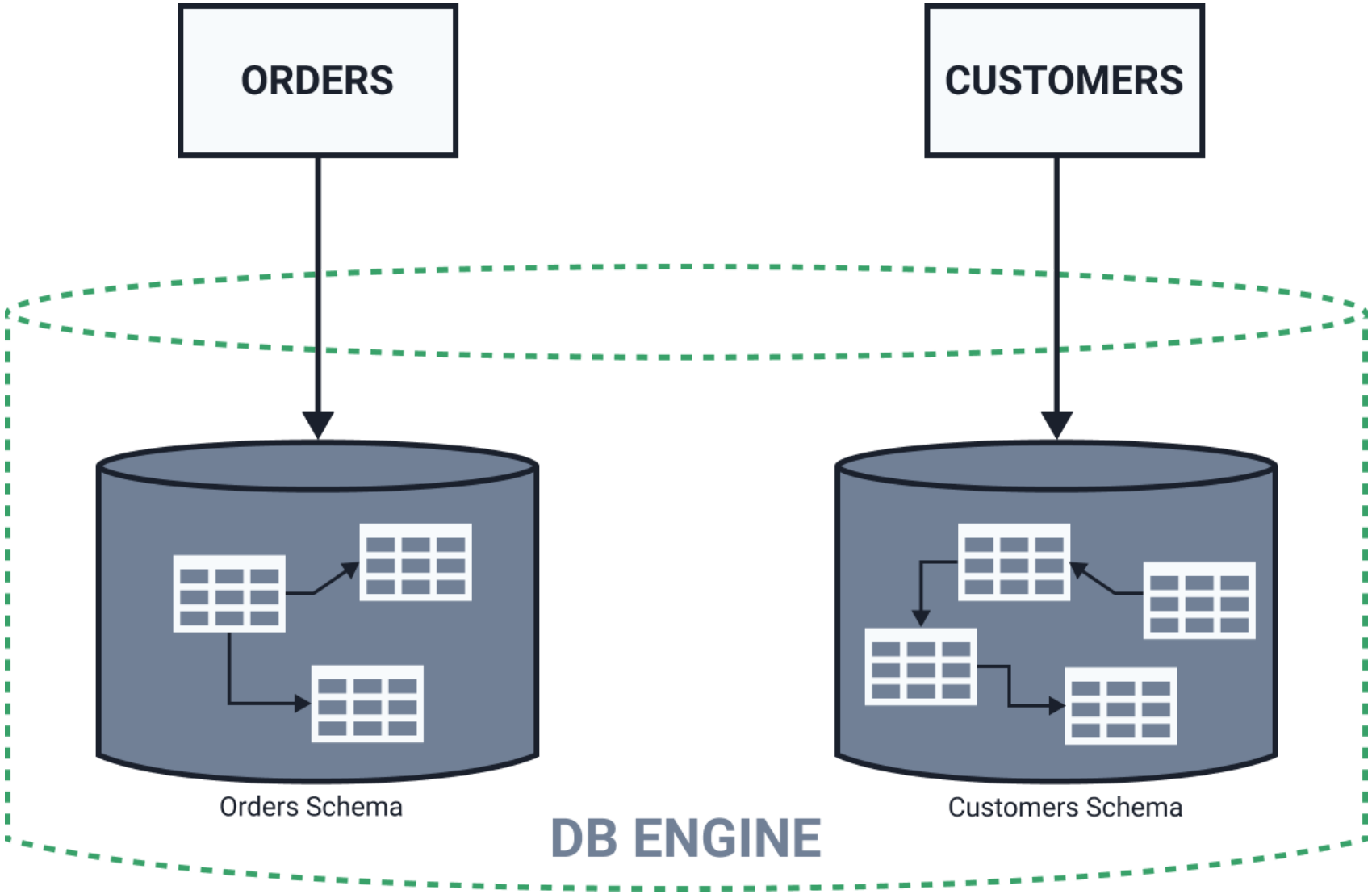
3

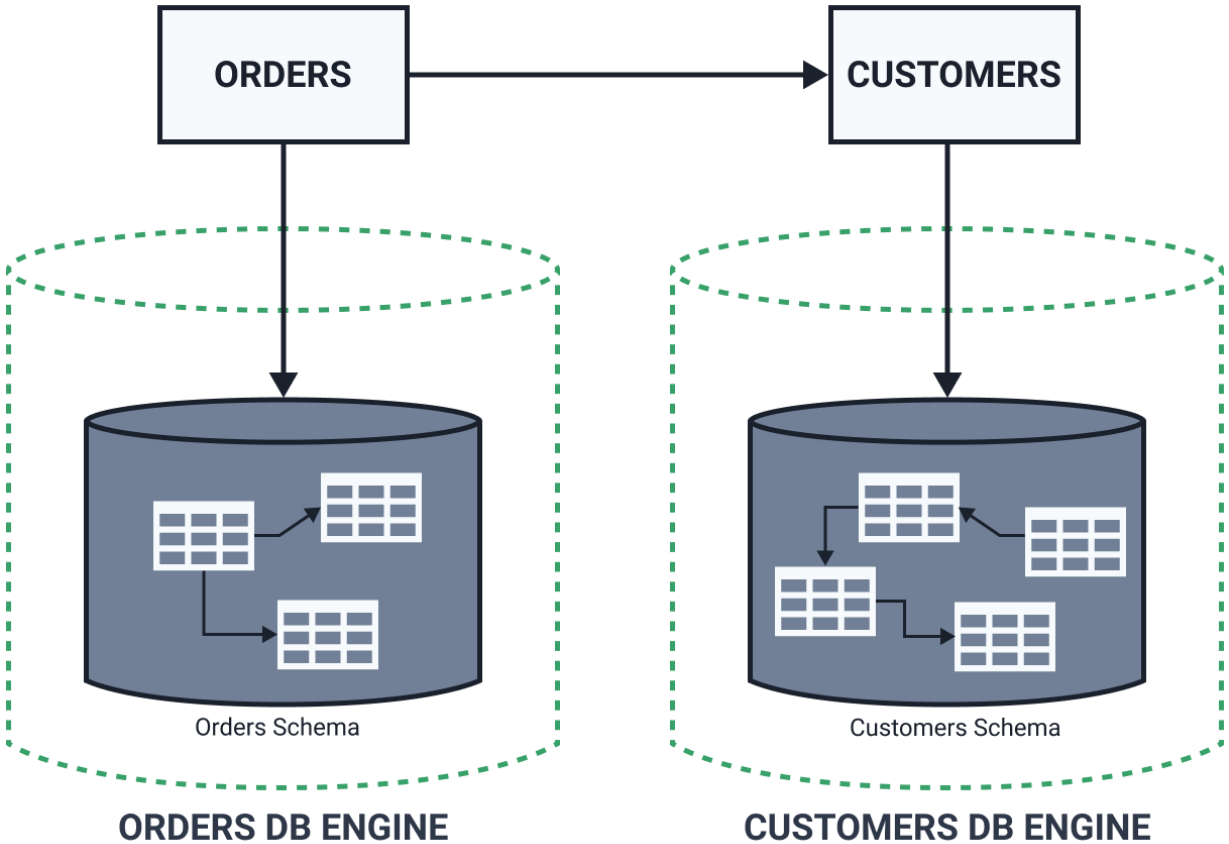
SWITCH TO NEW



4

Splitting Databases

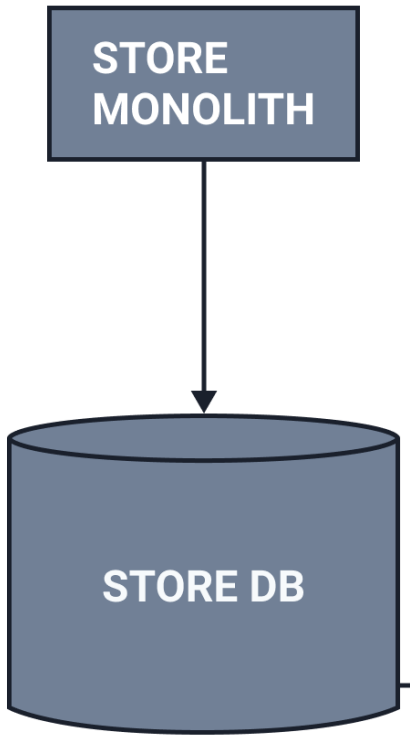

ORDERS
CUSTOMERS
Orders Schema
Customers Schema
DB ENGINE


ORDERS
CUSTOMERS
Orders Schema
Customers Schema
ORDERS DB ENGINE
CUSTOMERS DB ENGINE

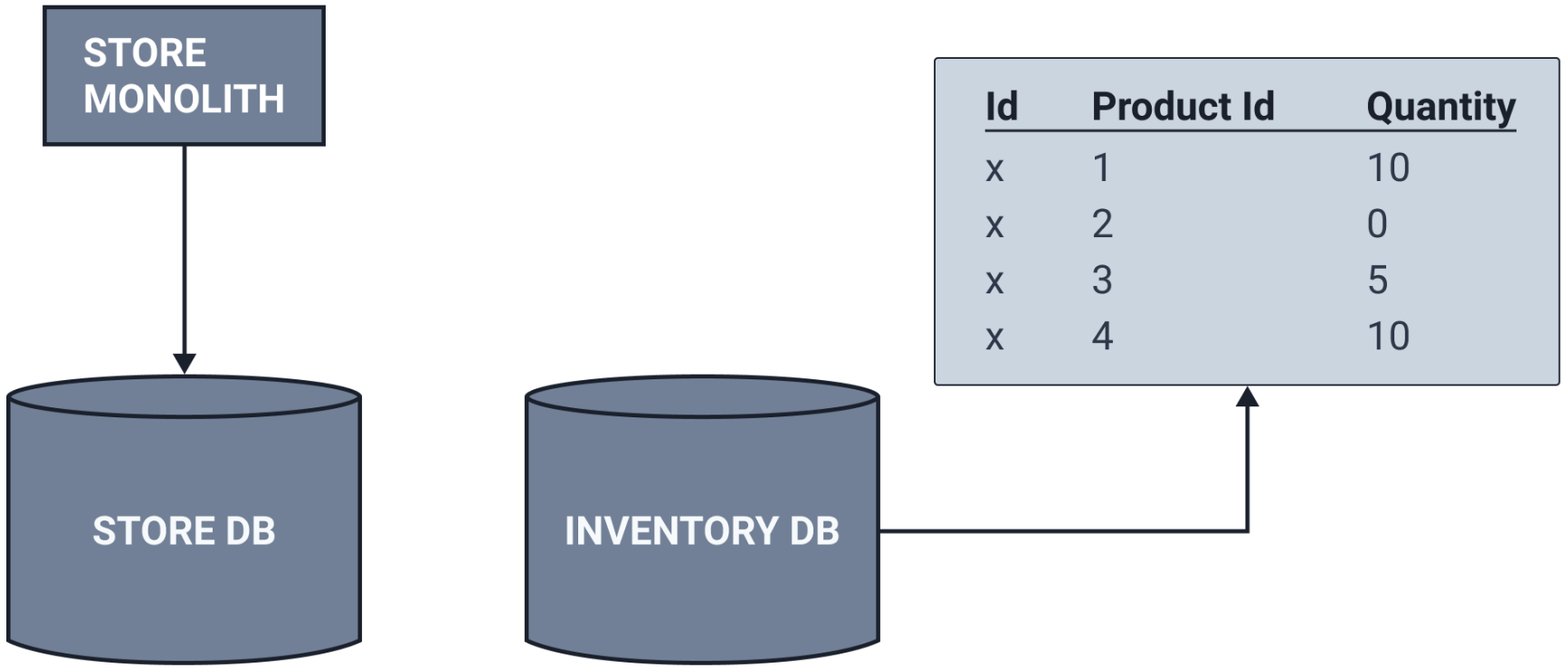
What to split first?

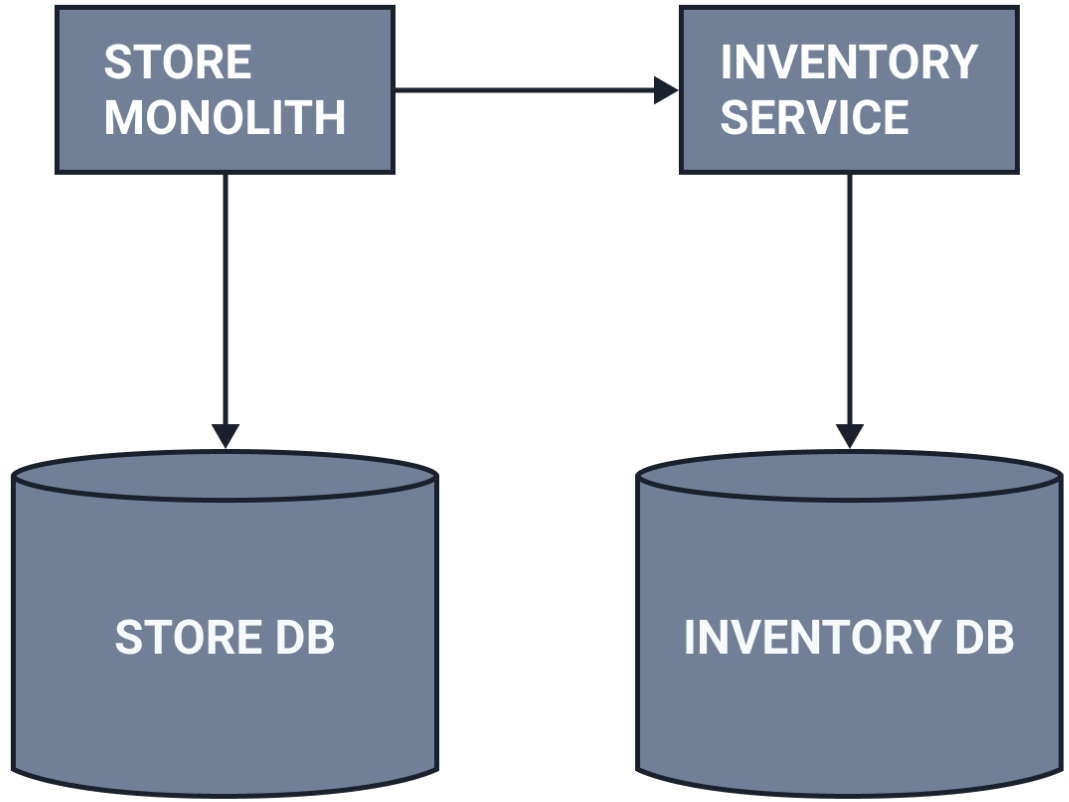
- Database first, code second
- Code first, database second
- Both at once

DEMO



Id	Product Name	Product Price	Quantity
1	Product 1	19.99	10
2	Product 2	10.99	0
3	Product 3	5.99	5
4	Product 4	19.99	10


STORE MONOLITH
STORE DB
INVENTORY DB
Id
Product Id
Quantity
x
1
10
x
2
0
x
3
5
x
4
10

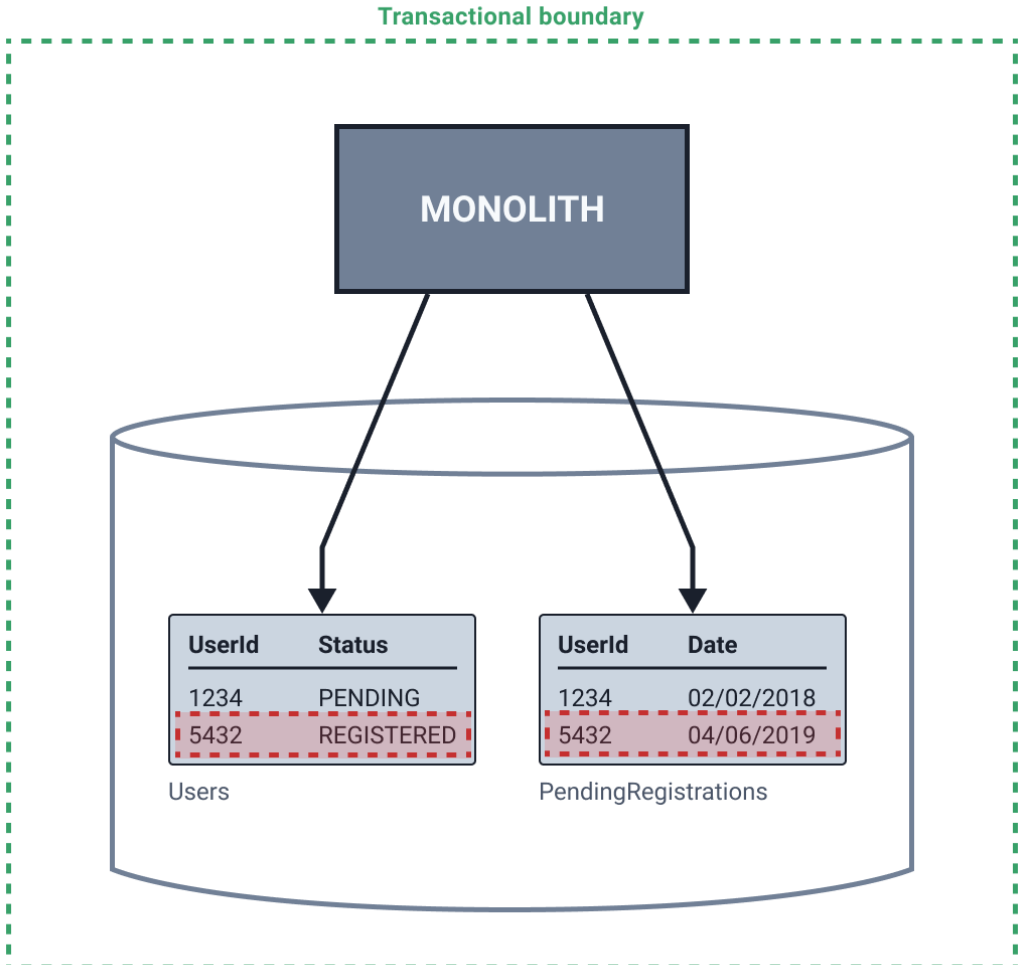

STORE MONOLITH
INVENTORY SERVICE
STORE DB
INVENTORY DB

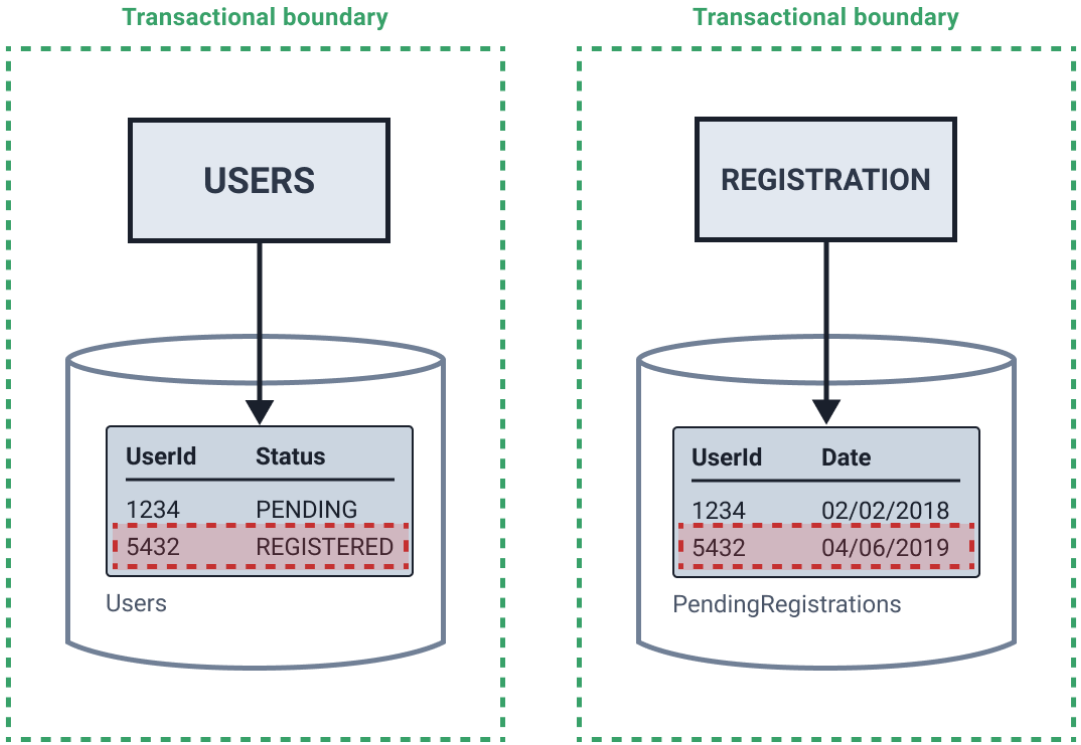
BREAK

Transactions

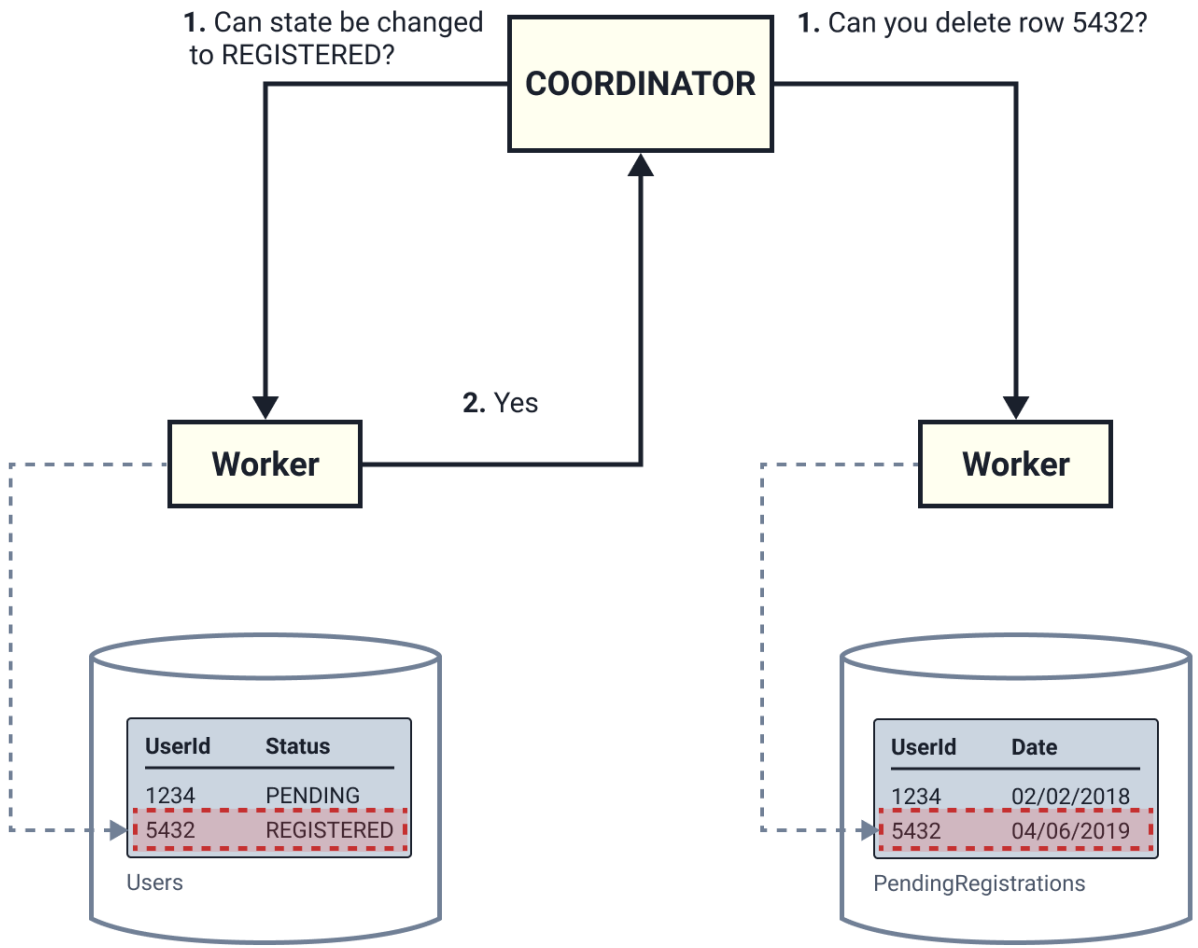
ACID Transaction

- Atomicity
- Consistency
- Isolation
- Durability


Transactional boundary
MONOLITH
UserId
Status
1234
PENDING
5432
REGISTERED
Users
UserId
Date
1234
02/02/2018
5432
04/06/2019
PendingRegistrations


Transactional boundary
USERS
UserId
Status
1234
PENDING
5432
REGISTERED
Users
Transactional boundary
REGISTRATION
UserId
Date
1234
02/02/2018
5432
04/06/2019
PendingRegistrations

Two-Phase Commit (2PC)



Two-Phase Commit

Pros:

- Guarantees an atomic transaction

Cons:

- Slow, depends on the transaction coordinator
- Database row locking can lead to deadlocks
- Doesn't scale

Alternatives

1. Don't split the data

2. Sagas

Sagas

- Coordinate multiple state changes
- How to handle long-lived transactions?
  - Break-up the LLT into sub-transactions
- Short-lived sub-transactions

Sagas - Example

Online Purchase

1. Check if in stock and reserve -> Warehouse service
2. Charge the user for the product -> Payment service
3. Send the notification -> Notification service
4. Package and send the order -> Warehouse service

Handling Failures

- Backward recovery
  - Rollback
  - Compensating actions
- Forward recovery
  - Continue and retry

https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf

Compensating transaction

Online Purchase

1. Check if in stock and reserve -> Warehouse service (OK)
2. Charge the user for the product -> Payment service (OK)
3. Send the notification -> Notification service (OK)
4. Package and send the order -> Warehouse service (ERROR)

Compensating transaction - Rollback

Online Purchase Rollback

1. Check if in stock and reserve -> Warehouse service (OK)

   (ROLLBACK) Remove the reservation

2. Charge the user for the product -> Payment service (OK)

   (ROLLBACK) Return the money back to the user

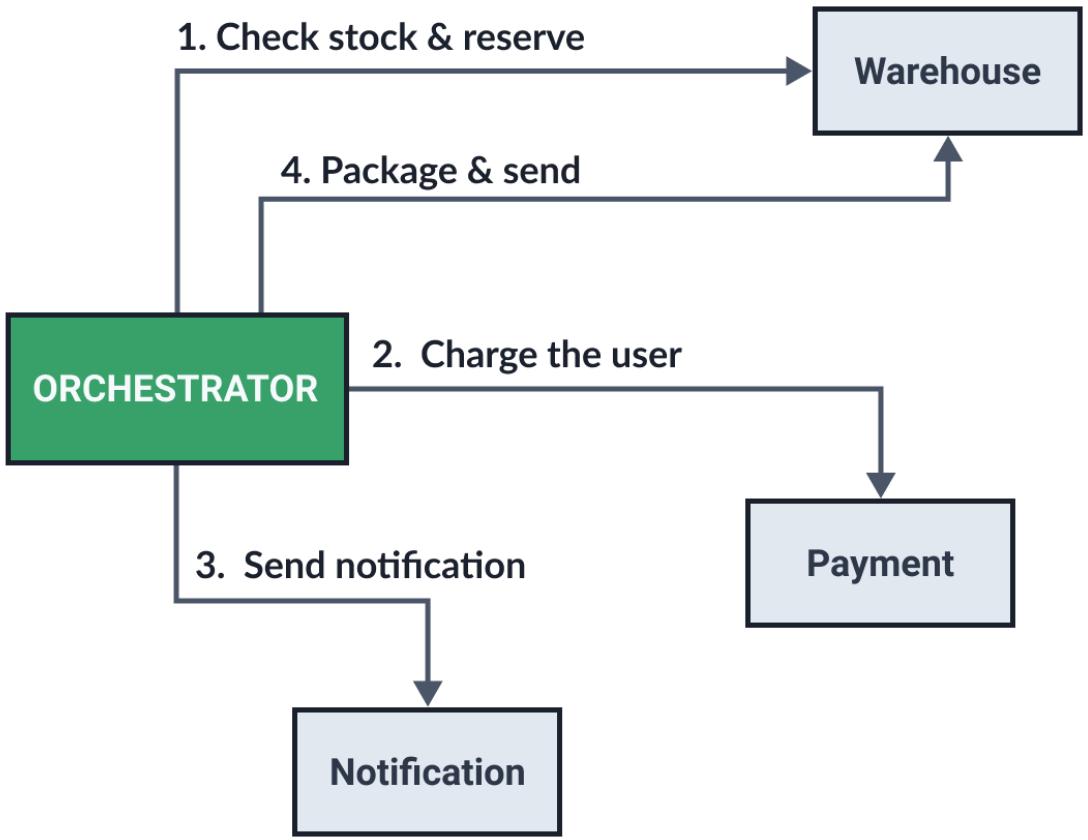
3. Send the notification -> Notification service (OK)

   (ROLLBACK) Notify use that the item is not available

4. Package and send the order -> Warehouse service (ERROR)

Implementing Sagas

- Orchestrated Sagas
  - rely on centralized coordination
- Choreographed Sagas
  - no centralized coordination
  - more complicated tracking


1. Check stock & reserve
Warehouse
4. Package & send
ORCHESTRATOR
2. Charge the user
Payment
3. Send notification
Notification

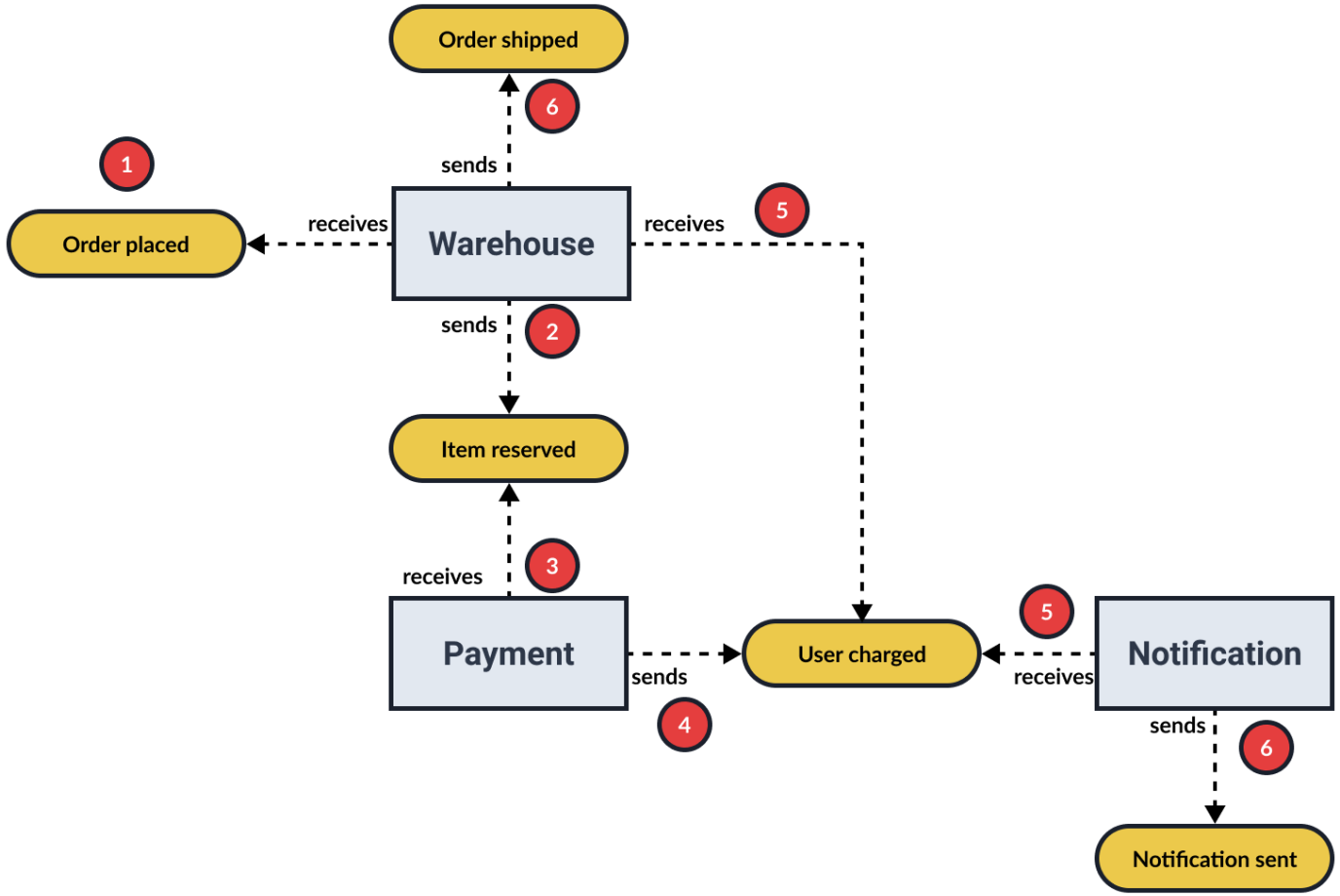
Orchestrated Sagas

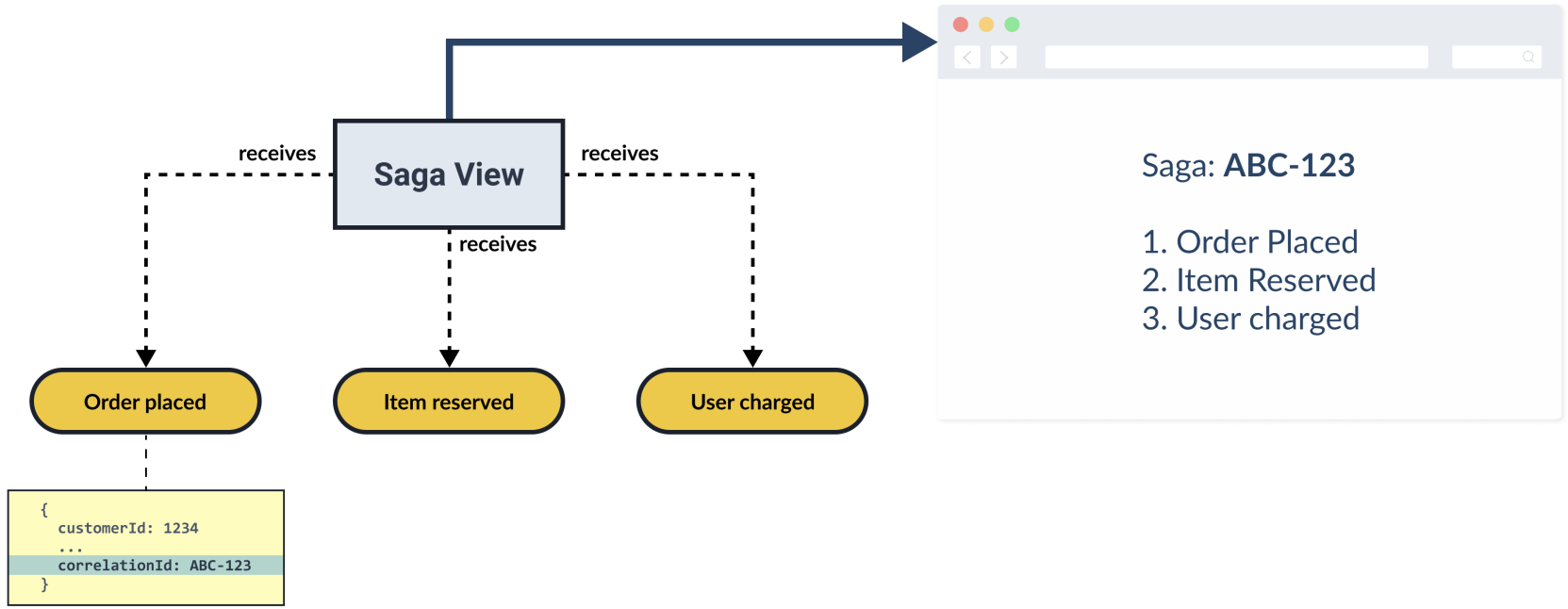
Pros:

- Whole business process centralized orchestrator
- Easier to understand

Cons

- Increased coupling (orchestrator knows about everything)
- Anemic services (logic in the orchestrator)


Order shipped
6
sends
1
Order placed
receives
Warehouse
receives
5
sends
2
Item reserved
receives
3
Payment
sends
4
User charged
5
receives
Notification
sends
6
Notification sent


receives
Saga View
receives
receives
Order placed
Item reserved
User charged
{
customerId: 1234
...
correlationId: ABC-123
}
Saga: ABC-123
1. Order Placed
2. Item Reserved
3. User charged

Choreographed Sagas

Pros:

- Loose coupling (services react to events)
- No centralization

Cons:

- Hard to know what's happening
- Hard to get the state of saga

Resources

Books:

- Monolith to Microservices by Sam Newman
- Cloud Native: Using Containers, Functions, and Data by Scholl, Swanson, and Jausovec

Blogs/Articles:

- SAGAS by Hector Garcia-Molina and Kenneth Salem
- Chris Richardson's Blog
- Martin Fowler's Blog

Thank you

Contact

- @pjausovec
- peterj.dev
- Slides: https://slides.peterj.dev
- Demos: https://github.com/peterj/gids-deconstructing-monoliths