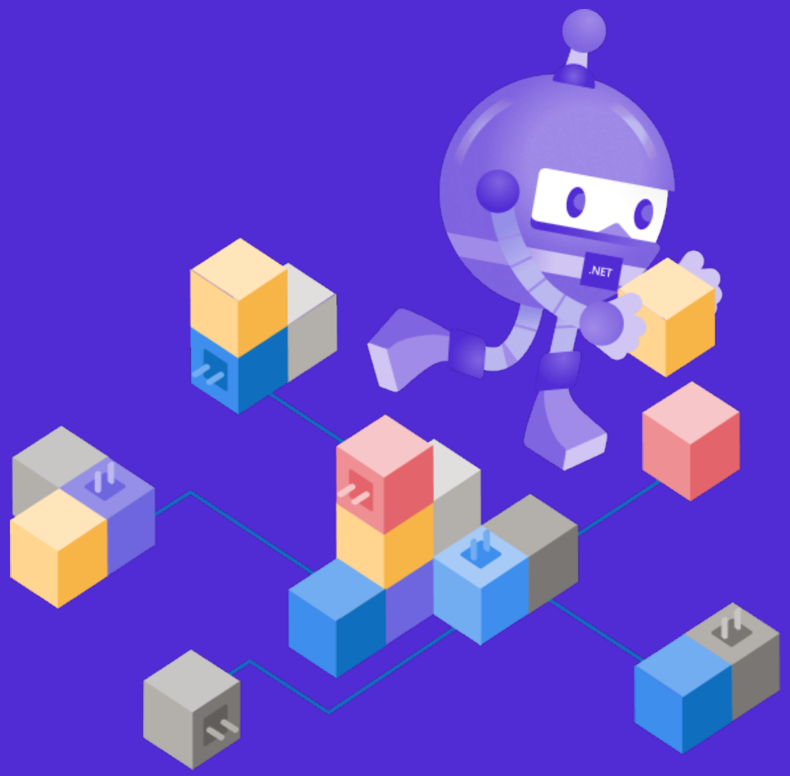
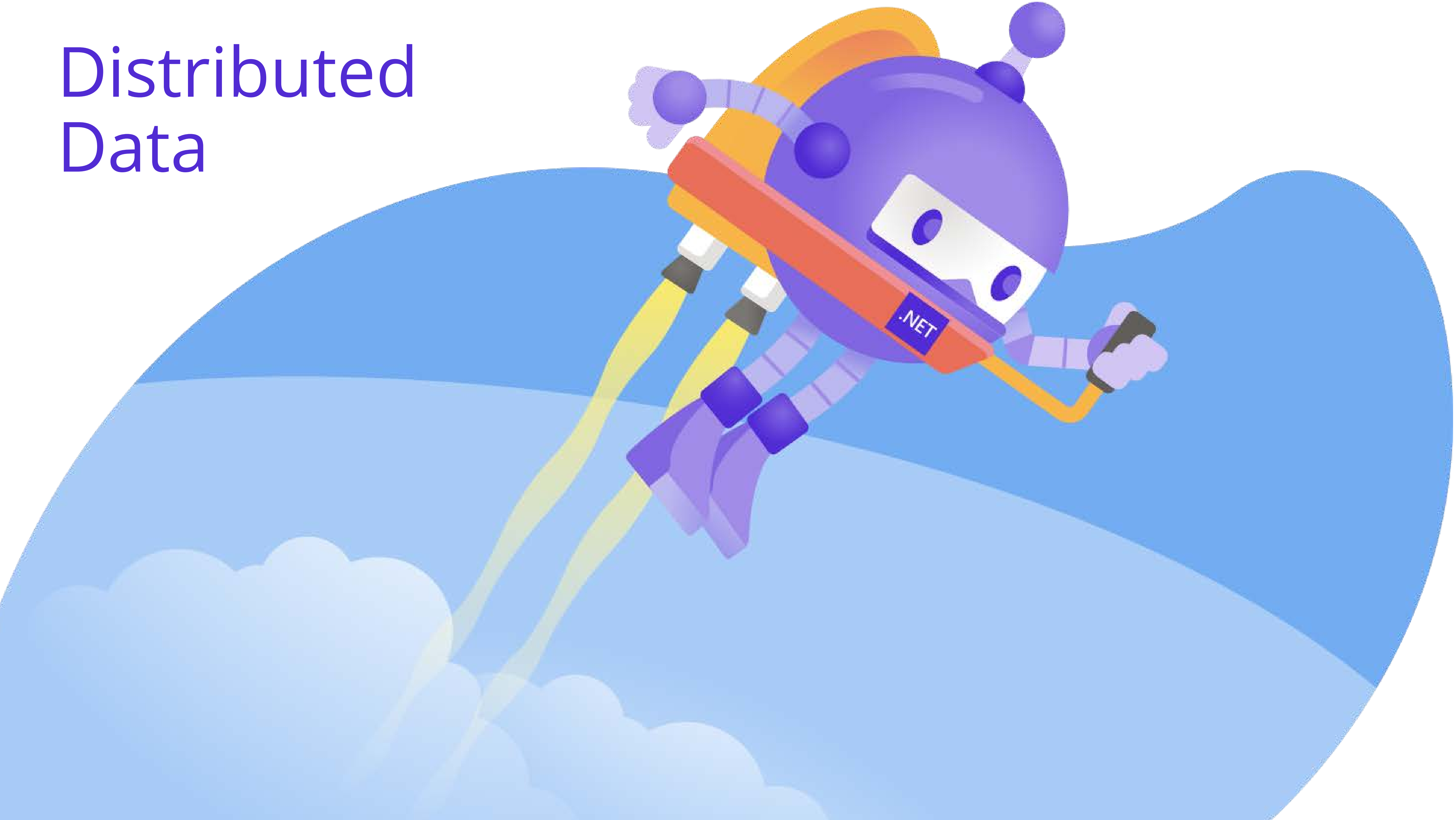
Distributed Data

Rob Vettor

Monu Bambroo



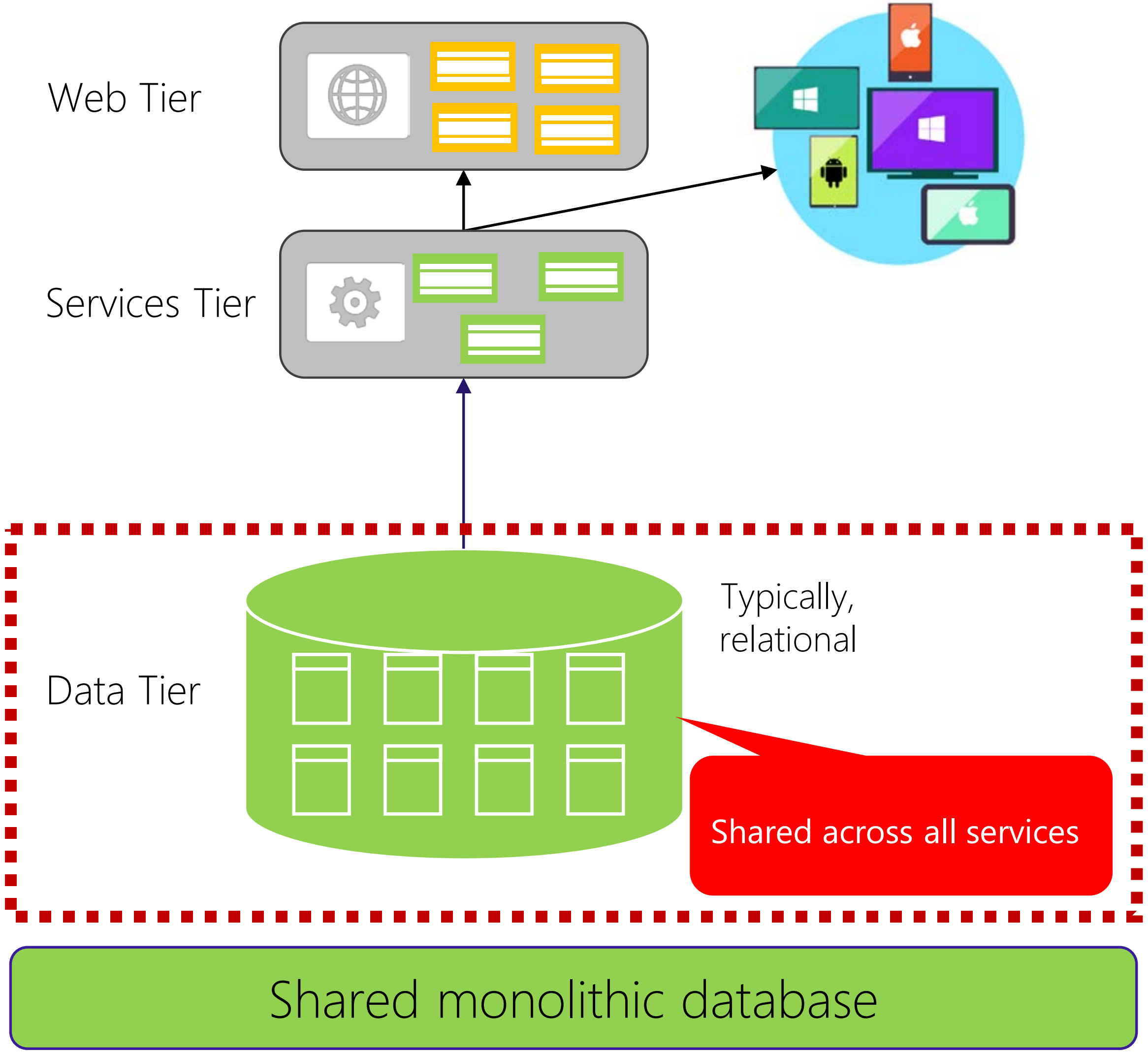
Distributed Data



Database per Microservice

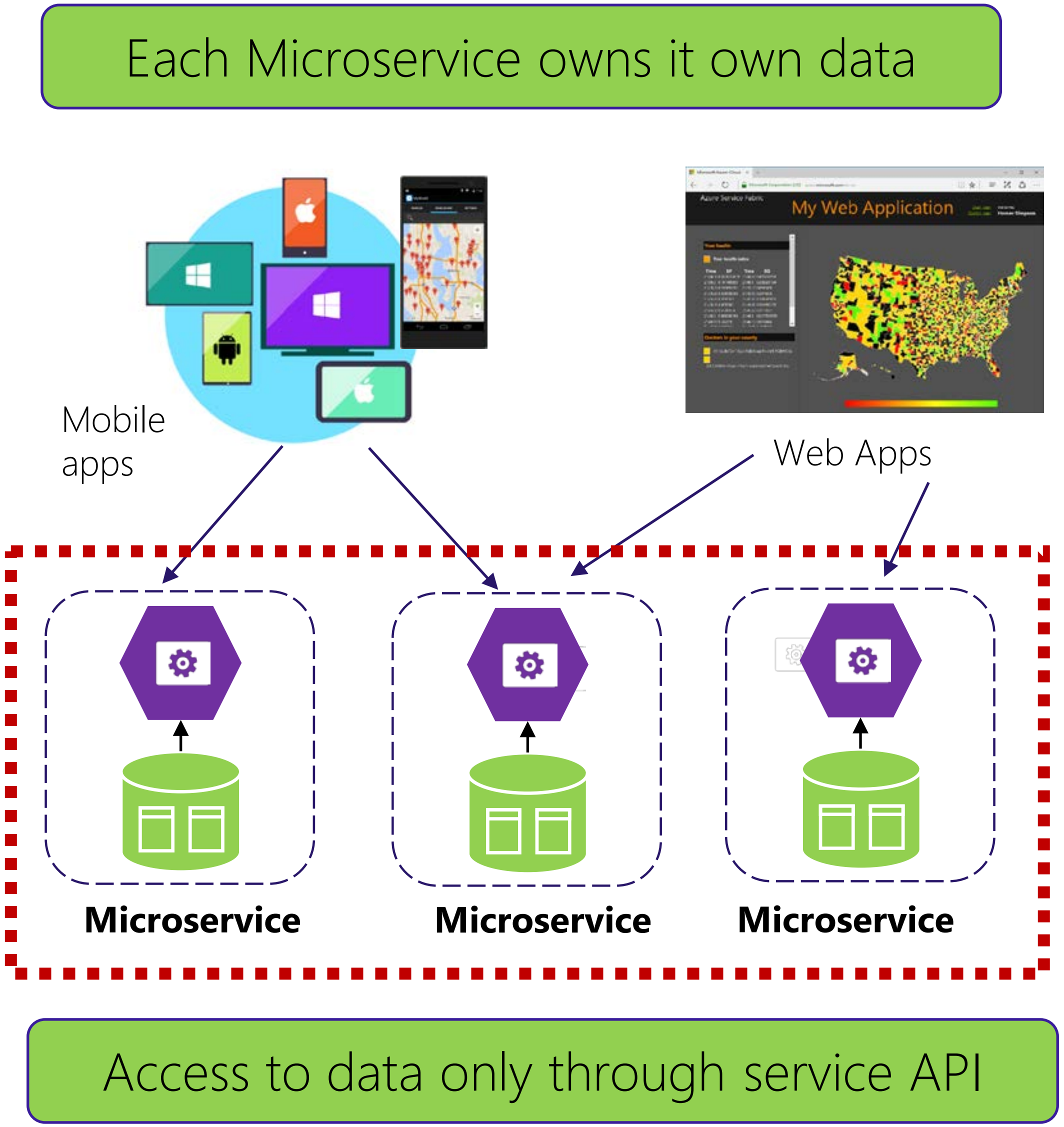
Distributed data:
Perhaps the hardest part

Data – Monolithic Approach



- Monolithic apps favor a shared data store
- Typically, a relational database
- Straightforward to…
  - Query multiple tables
  - Invoke ACID transactions
- *Immediate consistency (database always consistent)*
- Single resource to manage
- Scales up, not out

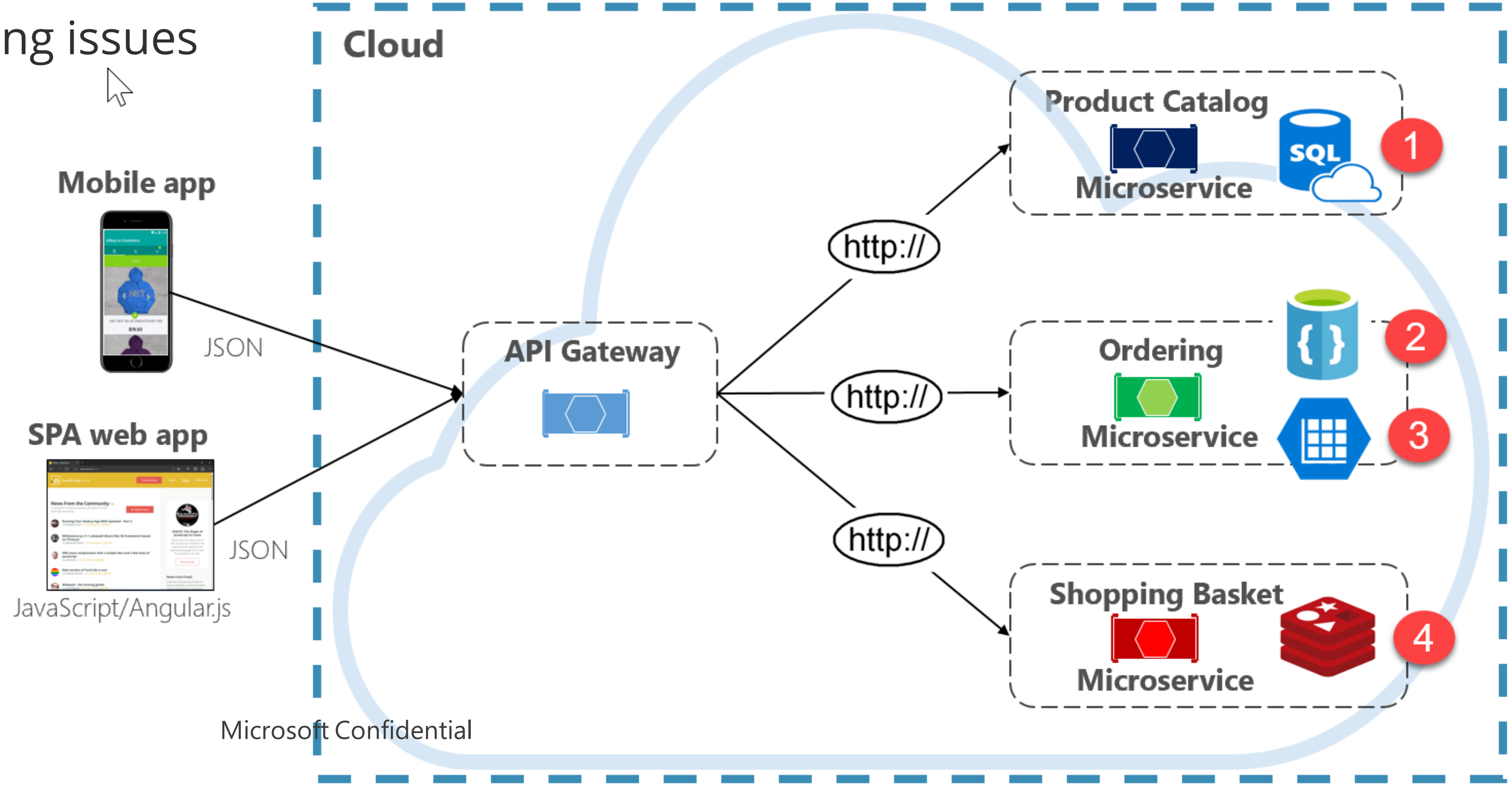
Data – Microservices Approach



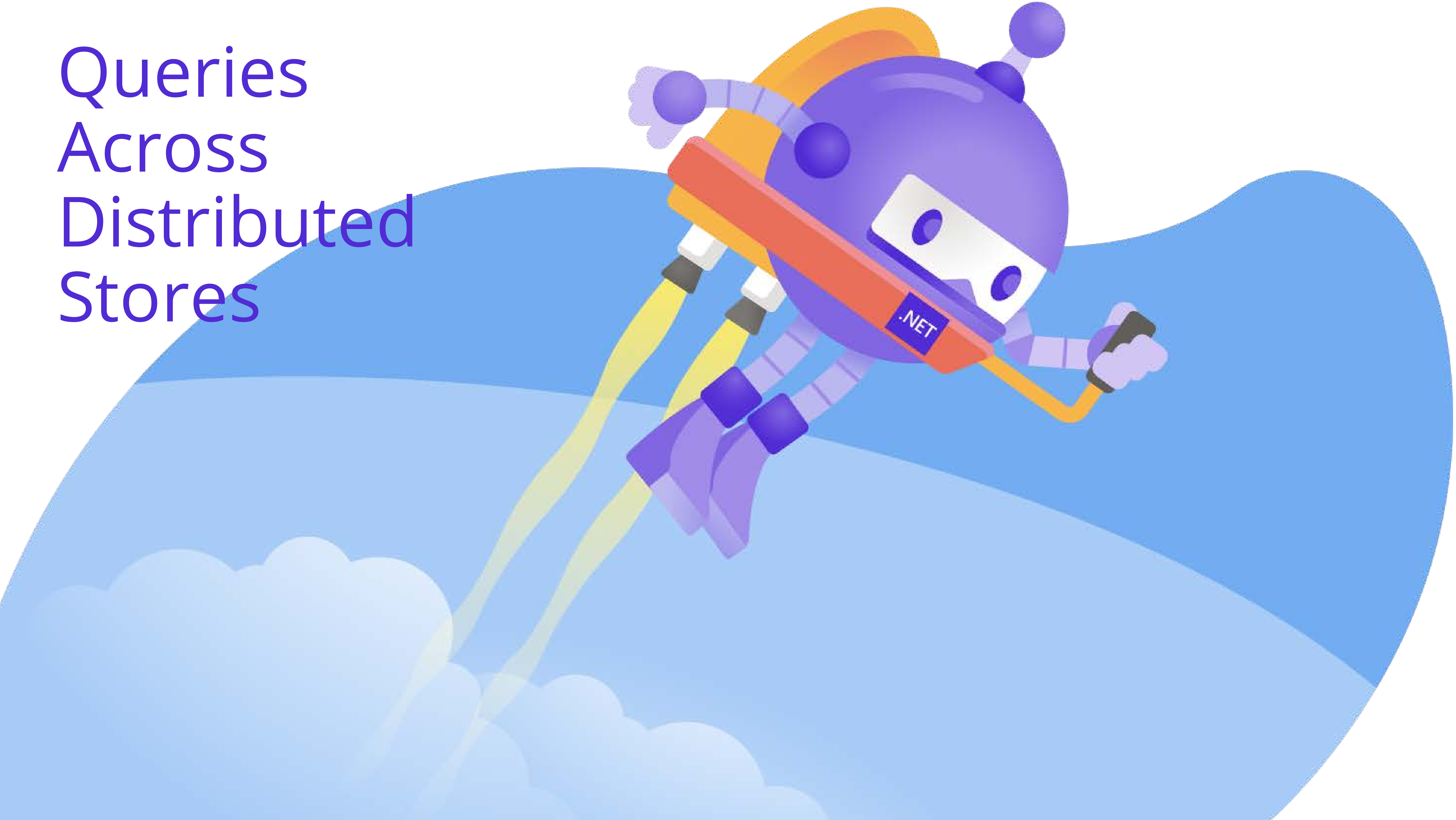
- Database per microservice
- Each microservice, by definition, encapsulates its domain data into its own datastore
  - Services are loosely coupled and can evolve independently
  - Avoids data model conflicts and data coordination challenges
  - Reduces contention and competing read/write patterns

Distributed Data

- Architecture supports *polyglot data* - multiple data technologies
  - Each service implements data store of choice for its workload
  - Separate stores…
    - Help guarantee proper bulkheads, or separation, between services
    - Resilient to upstream/downstream failures
    - Reduces contention and locking issues
    - Can scale independently

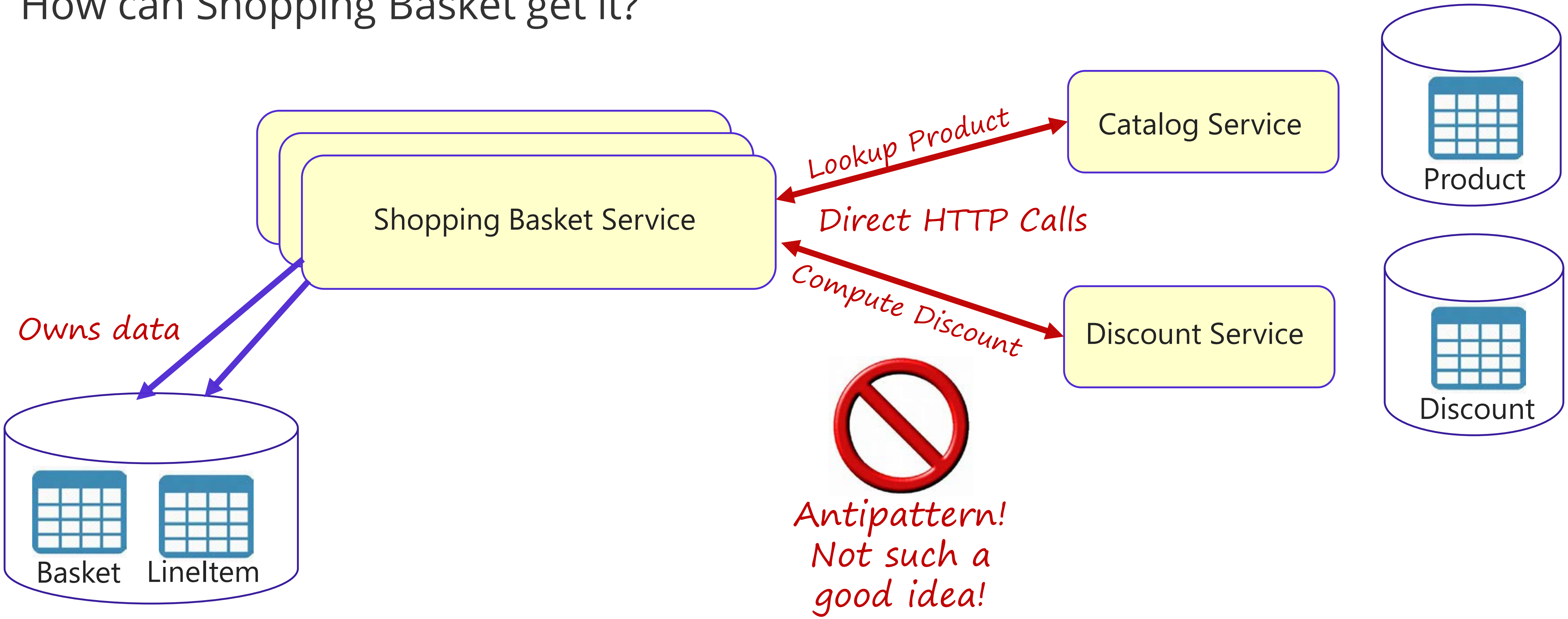


Queries Across Distributed Stores



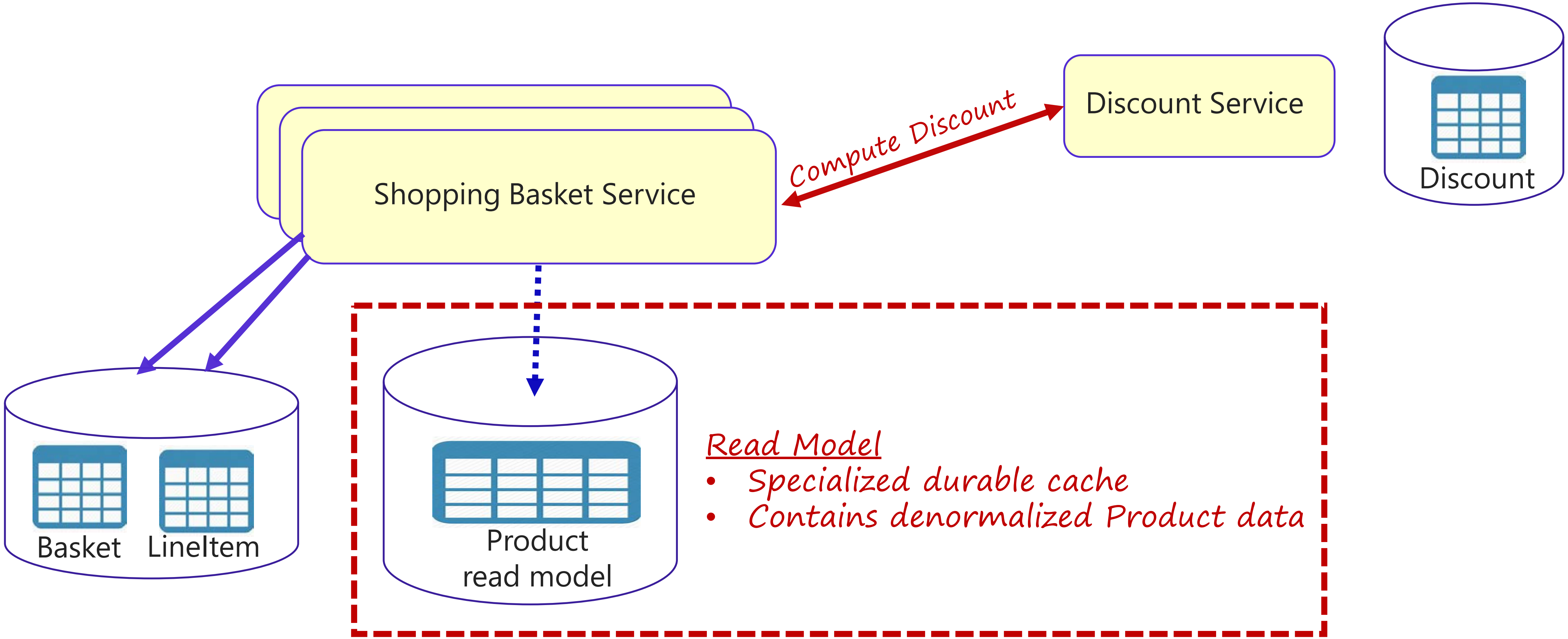
Querying Data Across Services

- Shopping Basket owns basket and lineItem data
- But, requires subset of product data and a computed discount value
- Other services own product and discount
- How can Shopping Basket get it?



Materialized View Pattern

- Best practice: ShoppingBasket maintains its own *read model*
- Contains copy of denormalized data owned by other services
- Decreases coupling and provides *locality* - improves response time and reliability - removes coupling



Materialized View Pattern – Close Up

- The target service (ShoppingBasket) maintains a local *read model* with denormalized data owned by Catalog service

Catalog Service

System of Record

Product

Id	Title	Price
1	Tribute	14.99
2	Miles Ahead	15.59

Artist

Id	Name	XYZ
14	Miles Davis	aaa
23	Queen	bbb

Genre

Id	Name	XYZ
1	Rock	aaa
2	Jazz	bbb

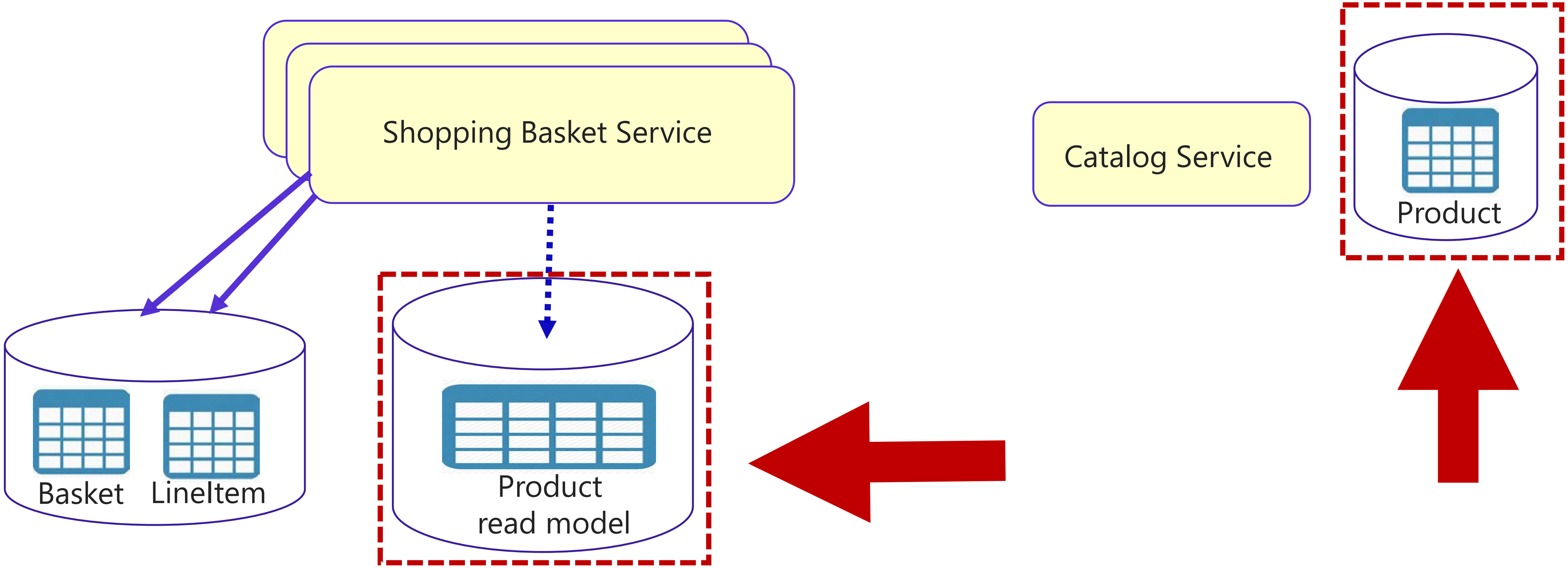
Shopping Basket Service

Read Model
Denormalilzed, Read-Only View

Materialized View - Product

Id	Title	Price	ArtistName	GenreName
1	Tribute	14.99	Queen	Rock
2	Miles Ahead	15.59	Miles Davis	Jazz

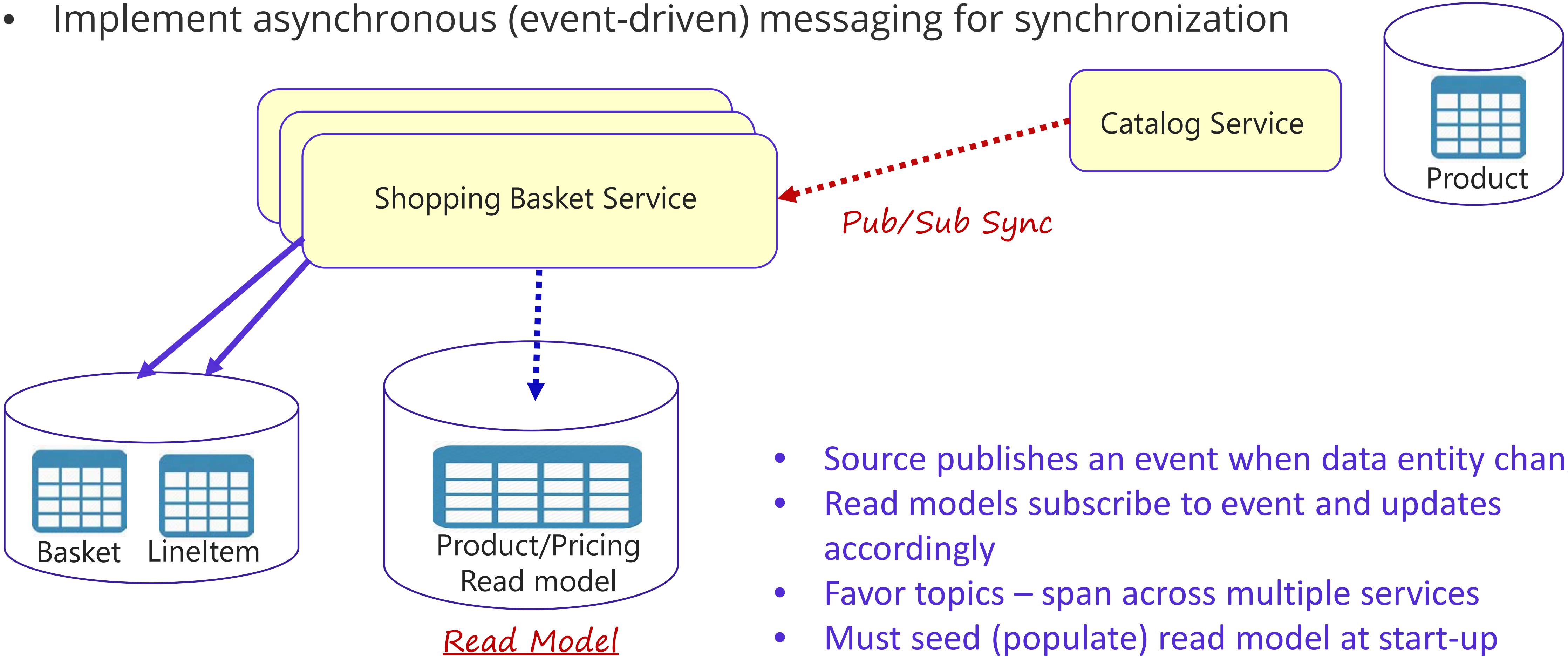
We now have Duplicate Data



- *Strategically* duplicating data is a common practice in distributed cloud-native services
- However, one and only one service owns the data and its state (system of record)
- All other copies are read-only

Read Model Consistency

- Must maintain consistency between read models and their source
  - Read model never updates itself
  - Implement asynchronous (event-driven) messaging for synchronization

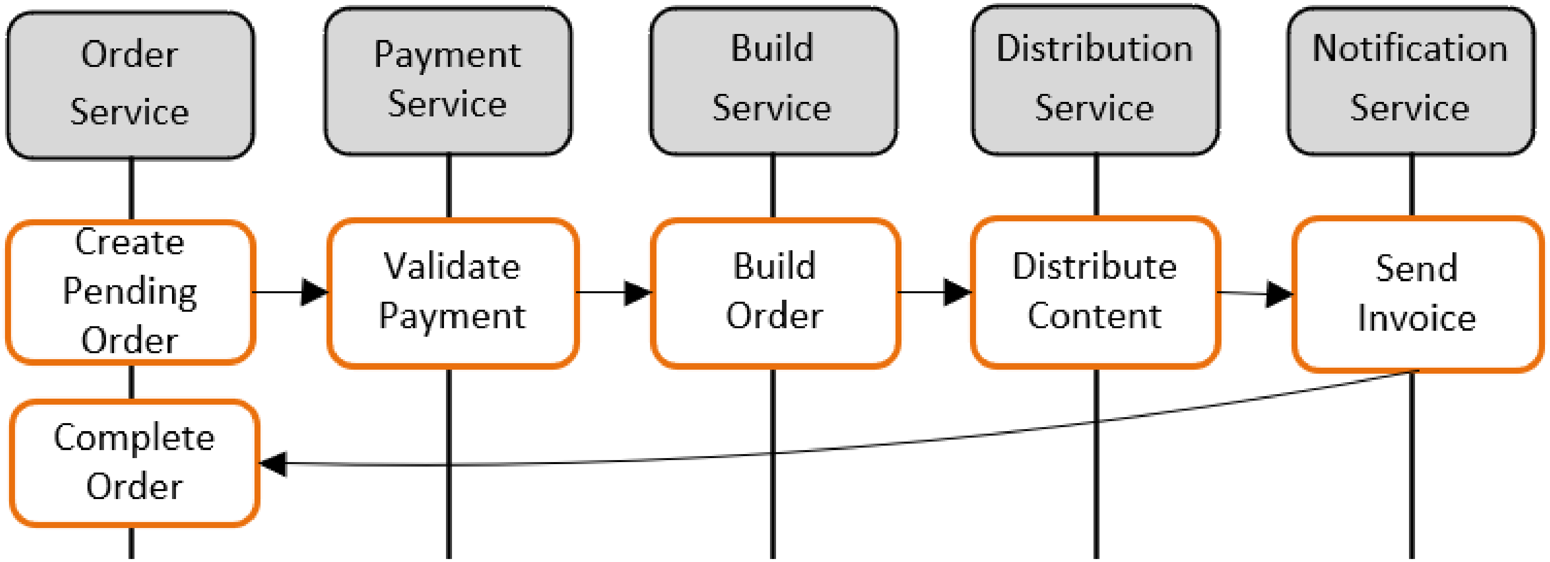


- Source publishes an event when data entity changes
- Read models subscribe to event and updates accordingly
- Favor topics – span across multiple services
- Must seed (populate) read model at start-up

Transactions Across Distributed Stores

Transactions across Microservices

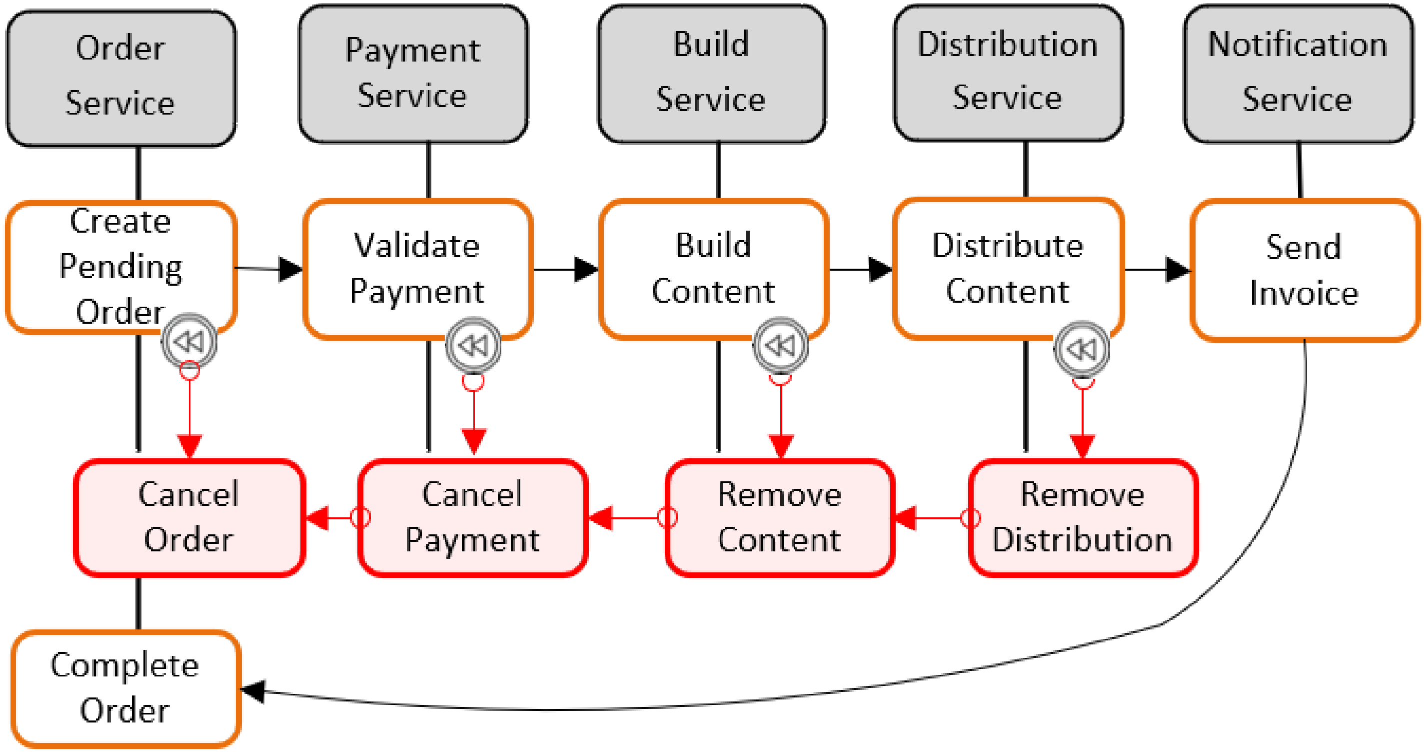
- How can we guarantee data consistency when modifying data across independent microservices?



- Can we wrap the order creation process in an ACID transaction?

Saga Pattern

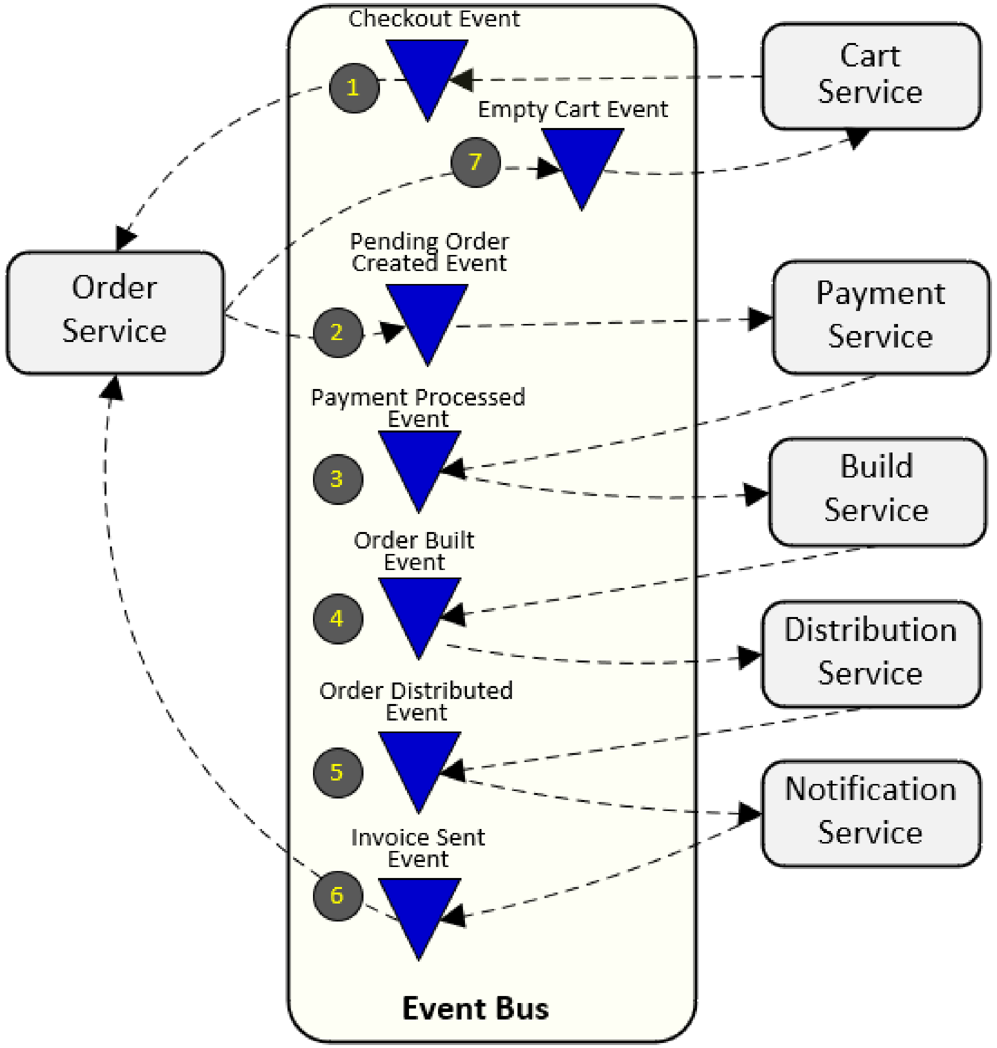
- Microservices do not support distributed transactions
- The Saga pattern can help enforce data consistency across microservices
  - Message-driven sequence of local transactions in which each service is sequentially updated
  - If a local transaction fails, the saga executes compensating transactions that undo updates made by preceding local transactions



Saga Pattern - Implementation

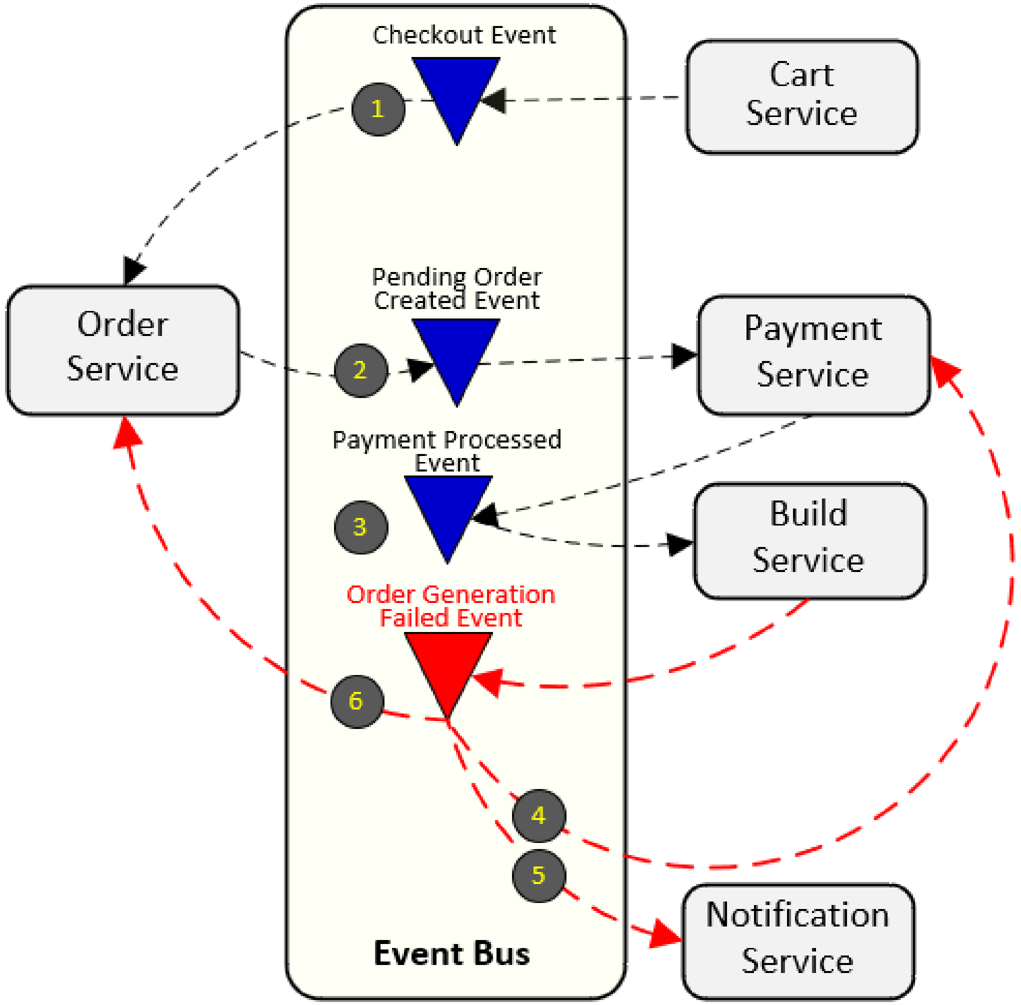
- Two common approaches…
  - Events/Choreography
    - Distributed decision making
    - Saga participants exchange events
    - Publish/subscribe pattern
  - Command/Orchestration
    - Centralize decision making
    - Saga orchestrator class coordinates
    - Command pattern

Saga Pattern – Choreography with Events



- Saga "participants" subscribe to events and respond accordingly
- Each step…
  - Performs an update operation
  - Commits local transaction
  - Publishes a corresponding event
- Each must happen atomically

Choreography Event Rollback



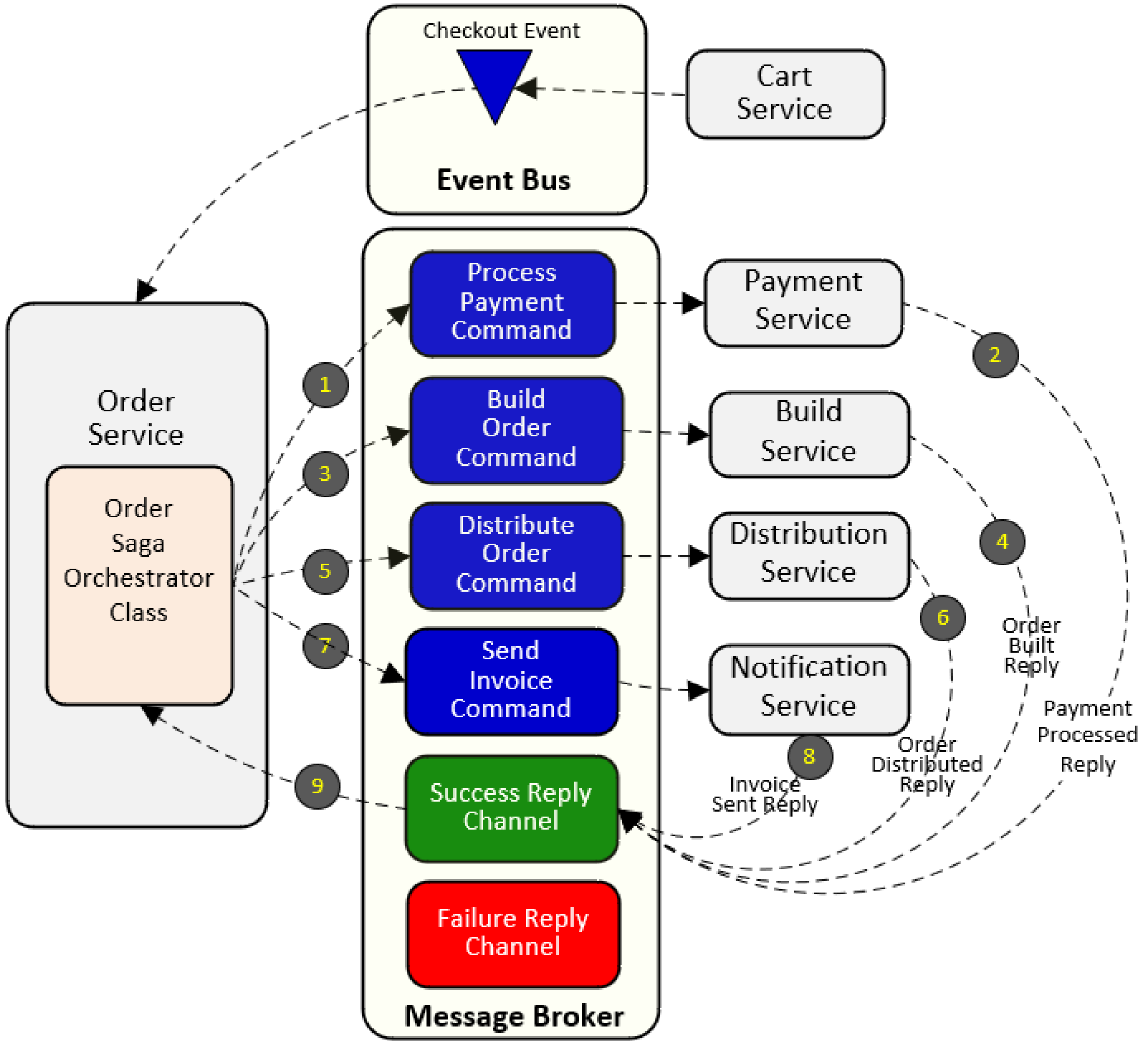
- What if a local transactions fails?
  - Invokes corresponding *compensating transaction* in reverse order to undo changes
  - Order is cancelled

Choreography – Benefits/Drawbacks

- No central coordination…
  - All participants are loosely-coupled
  - Must be programmed to respond to applicable events
  - Can be confusing to follow
- Simple plumbing
- Message broker guarantees resiliency -- buffers messages until operations execute

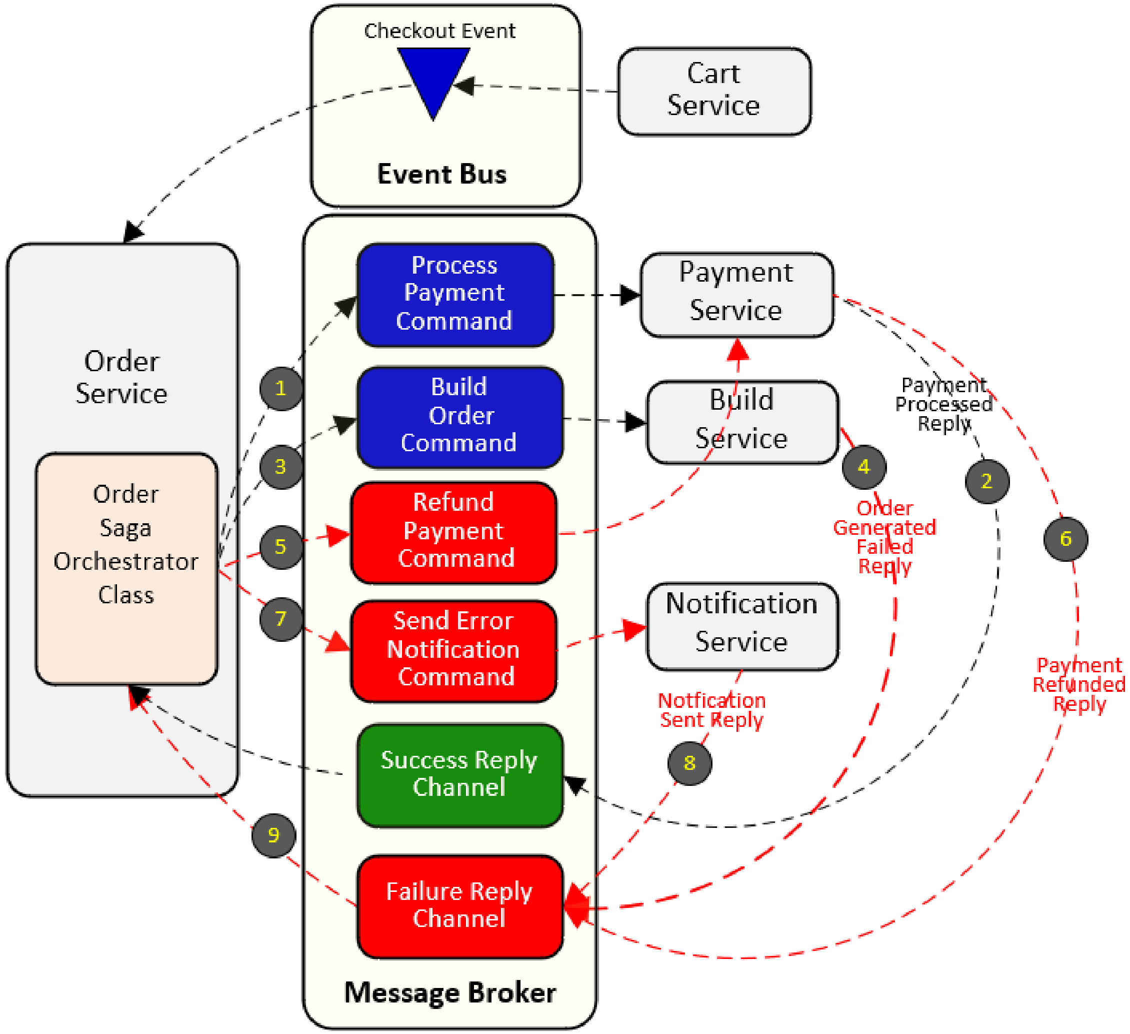
- Works well for simple transactions, but complex Sagas favor orchestration

Saga Pattern – Orchestration with Commands



- Orchestrator class directs operation
- Triggers each participant in an ordered sequence
- Invokes async/command messages using a queue and request/reply pattern
- Orchestrator processes reply message and proceeds to next step

Orchestration Event Rollback



- If any task fails, the orchestrator class receives failure reply and invokes compensating logic
- Sends compensating commands to each of the previous participants instructing it to rollback or cancel the previous operation
- Rollbacks are more straightforward with orchestration

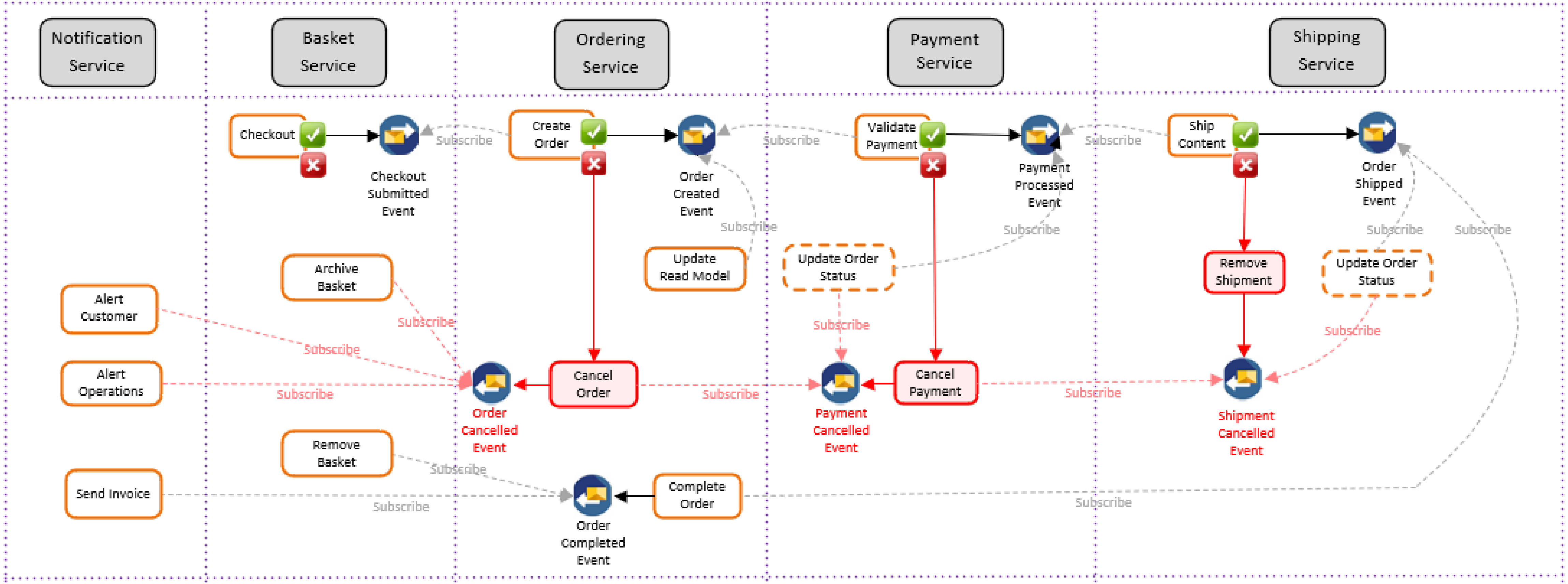
Orchestration – Benefits/Drawbacks

- Centralizes orchestration
- Reduces participant complexity
  - Each executes commands as directed by orchestrator
  - Each sends a corresponding reply to report status
- Straightforward to implement, follow and test
- Complexity remains linear as more steps are added

Saga Pattern – Best Practices

- Design orchestrators that contain sequencing but no business logic
- Create a unique identifier for each transaction for traceability
- Make operations idempotent – queues can deliver the same message twice
- Pass all data needed in the message or event – avoid incurring unnecessary overhead querying data stores

Proposed Saga



Subscribes to...
-- OrderCancelledEvent
-- OrderCompleted Event

Subscribes to...
-- OrderCancelledEvent
-- OrderCompleted Event

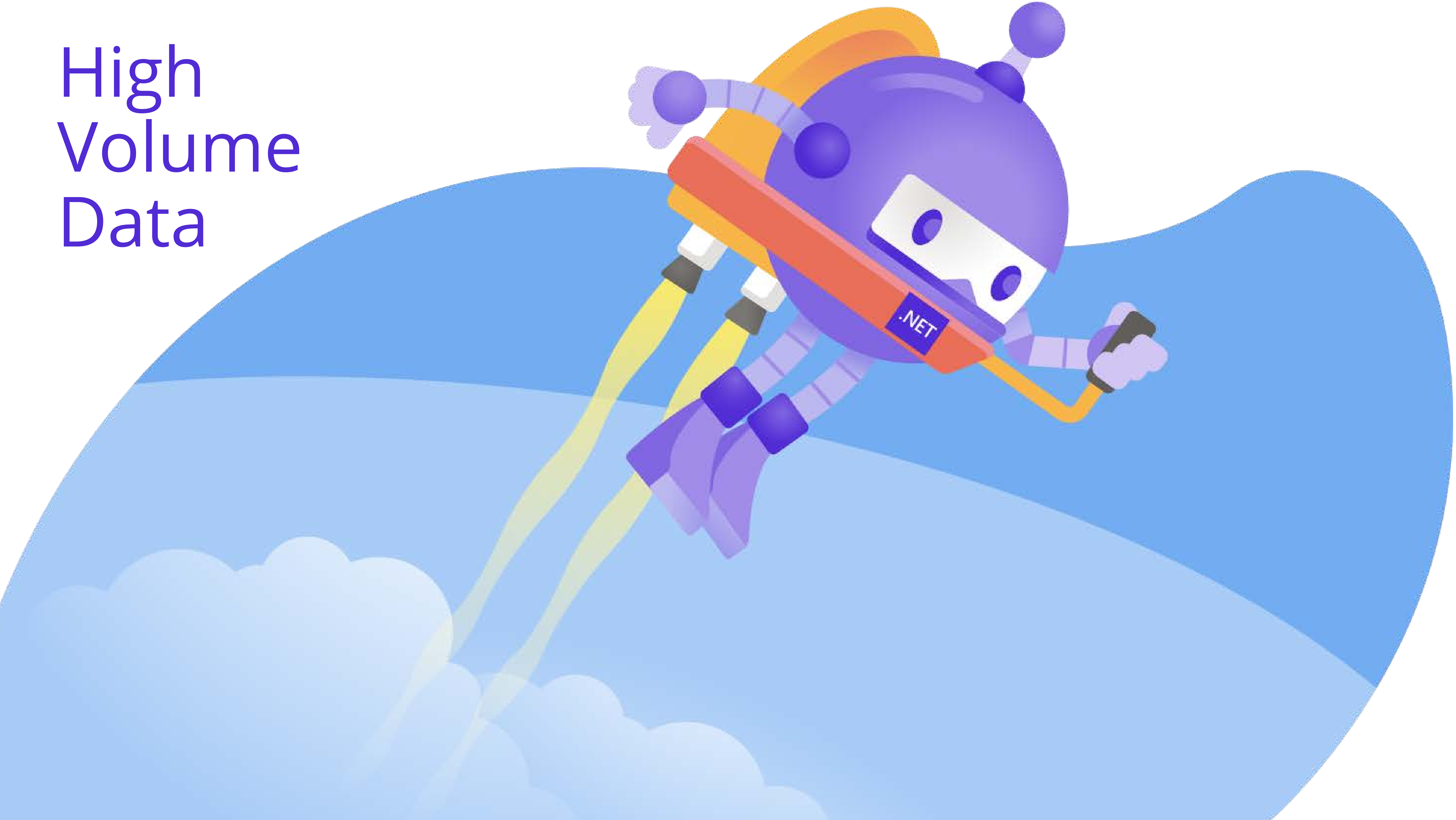
Subscribes to...
-- CheckoutEvent
-- OrderCreatedEvent
-- OrderShippedEvent
-- PaymentCanclledEvent
-- PaymentProcessedEvent

Subscribes to...
-- OrderCreatedEvent
-- ShipmentCancelledEvent

Subscribes to...
-- PaymentProcessedEvent

rob vettor
7/20

High Volume Data



High volume data in Cloud Native Applications

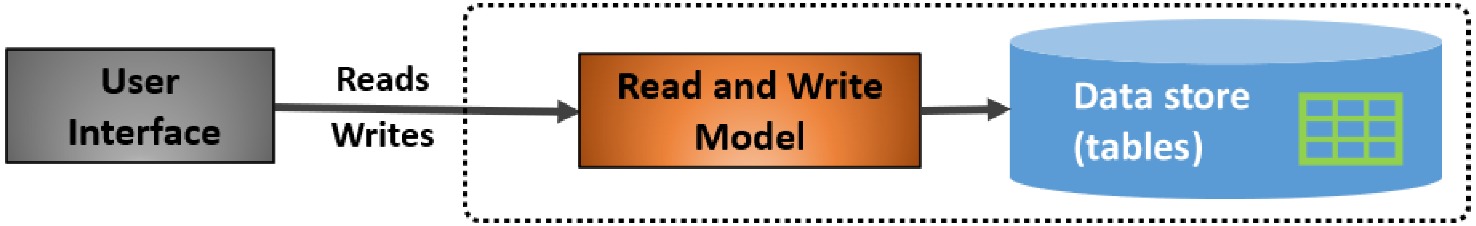
- Large cloud-native applications often support high-volume data requirements.
- In these scenarios, traditional data storage techniques can cause bottlenecks.
- For complex systems that deploy on a large scale, two patterns can help improve application performance:

CQRS

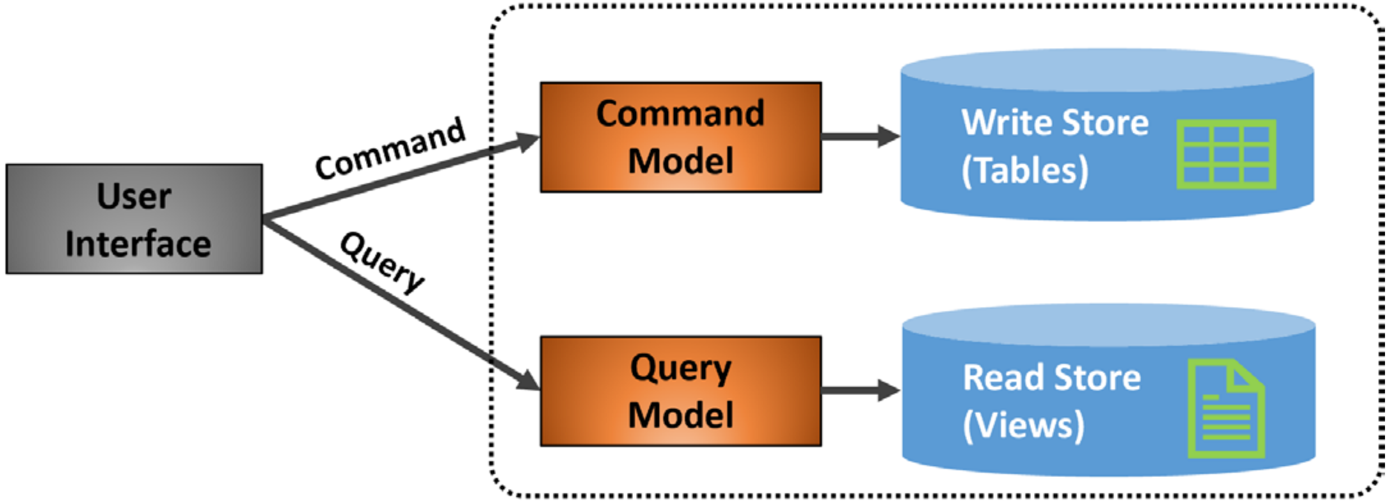
Event Sourcing

CQRS – Why?

- For most scenarios, read and write operations use the same data model and storage



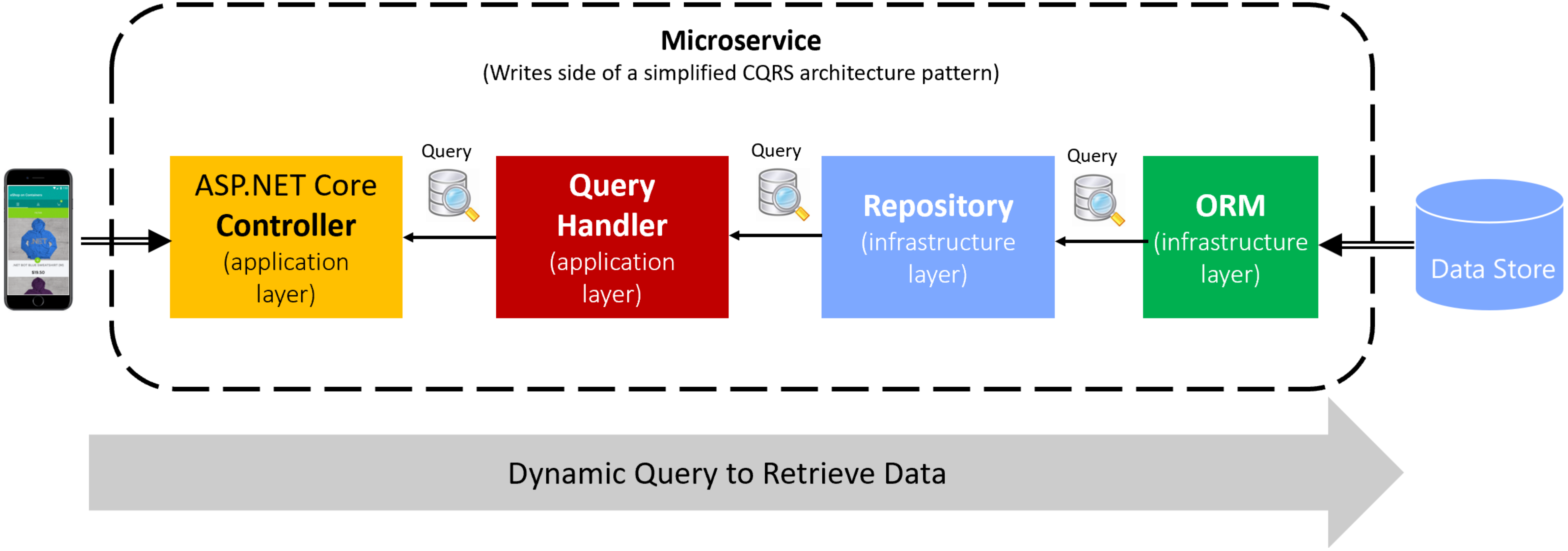
- However, high-volume scenarios might benefit from models that separate reads and writes



- Known as "Command and Query Responsibility Segregation Pattern"

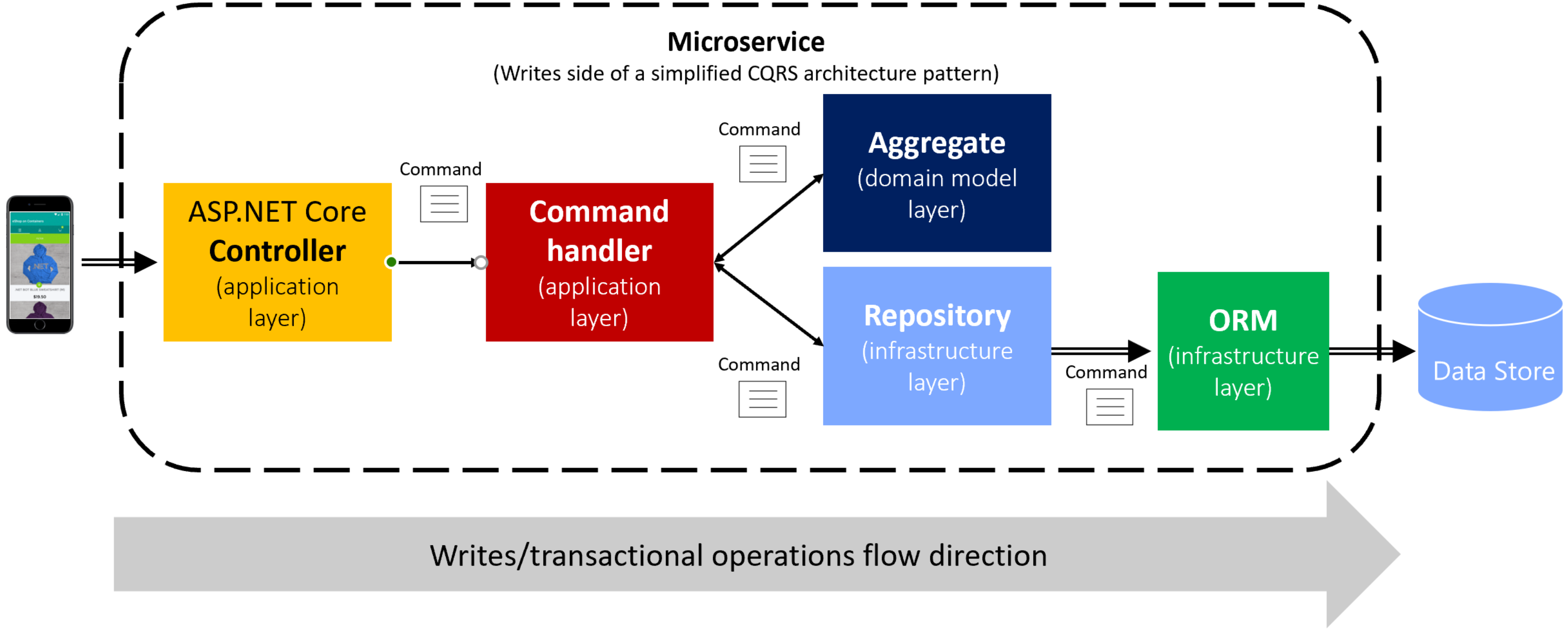
CQRS – Read operations

- To improve performance, read operations query against a highly denormalized data representation to *avoid* expensive table joins and locks



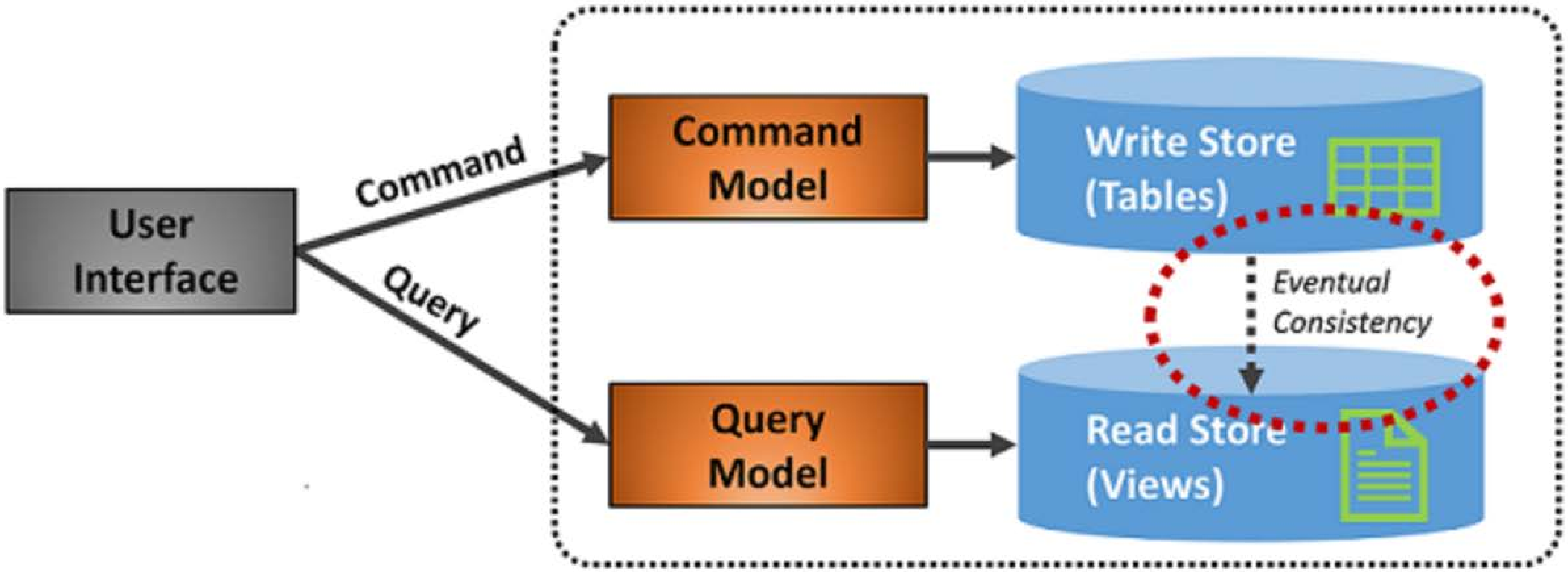
CQRS – Write operations

- The write operation, known as a command, updates against a fully normalized representation of data that guarantees consistency
- Complex business rules and domain logic are applied against write operations
- You might even impose tighter security on write operations than those exposing reads



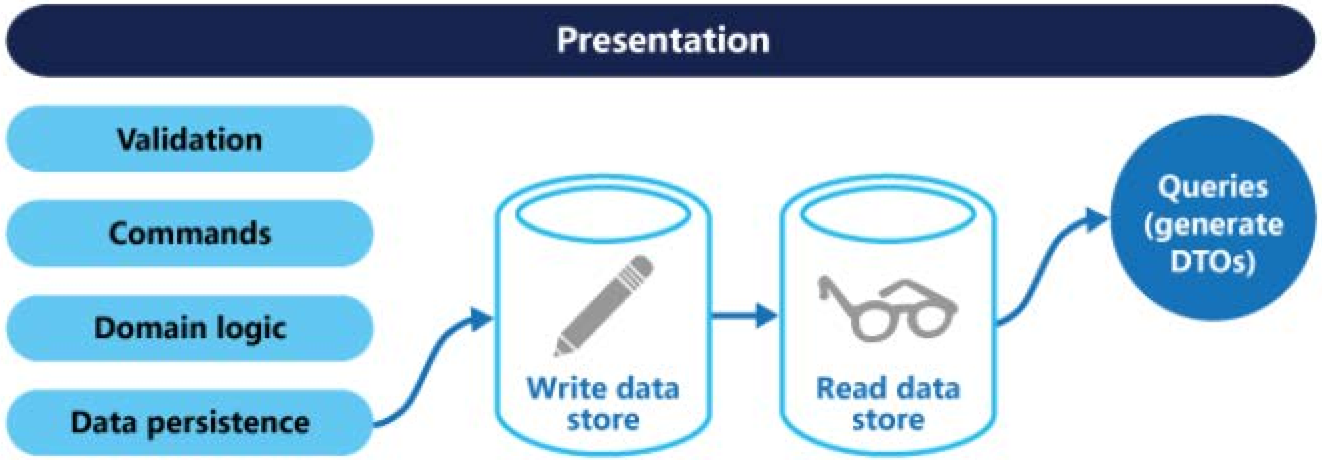
CQRS – Syncing Models

- You then need to implement a mechanism to keep both representations in sync.
- The write model store must update the read model store.
- You introduce *eventual consistency* to the system.



CQRS

- Segregate operations that read data from those that update data
- Help maximize, responsiveness, performance, scalability and security
- But, increases complexity
- Introduces eventual consistency
- The write model store must update the read model store
- Can scale read and writes separately
- Common where the volume of reads typically far exceed that of writes
- CQRS applied to limited sections of the system based upon needs



Demo: CQRS