Lund&Bendsen
developing developers

Event driven architecture

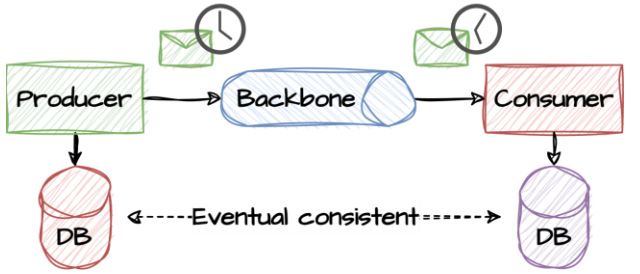
EDA overview

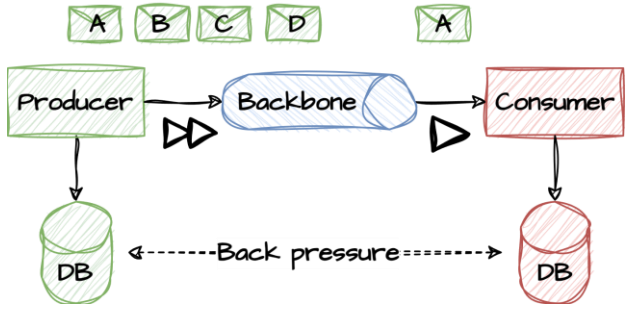
Lund&Bendsen
developing developers

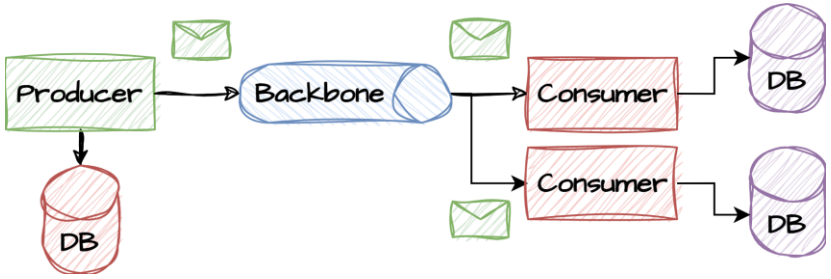
Event-driven architecture (EDA) is a software design pattern that enables an organization to detect “events” or important business moments (such as a transaction, site visit etc) and act on them in real time or near real time. This pattern replaces the traditional “request/response” architecture where services would have to wait for a reply before moving on to the next task. The synchronous communication approach does not scale well and introduces blocking code and increases risks of failures.

In EDA, events are emitted by an event emitter and broadcasted to any number of listeners through an event backbone. The event emitter has no knowledge of whom may consume the event. Event listeners are also completely decoupled from the event emitters and may emit new events onto the backbone as a response to the event. The decoupling of producers and consumers adds several benefits to the system design. The decoupling allows producers and consumers to produce/consume and different speed rates, allow new consumers to be added seamlessly and replay historic events.

The downside is added complexity when introducing an event pipeline in between components, observability issues and versioning of data.







Events, Commands and Queries

Lund&Bendsen
developing developers

Events

Something happened (provide identity + context)

Zero or many consumers (completely decoupled)

Single producer (owner of the event and its data)

Owned by producer

Commands

Invoke behavior (target a specific recipient)

Single consumer (consumer executes the command)

Multiple senders (Anyone can send a command)

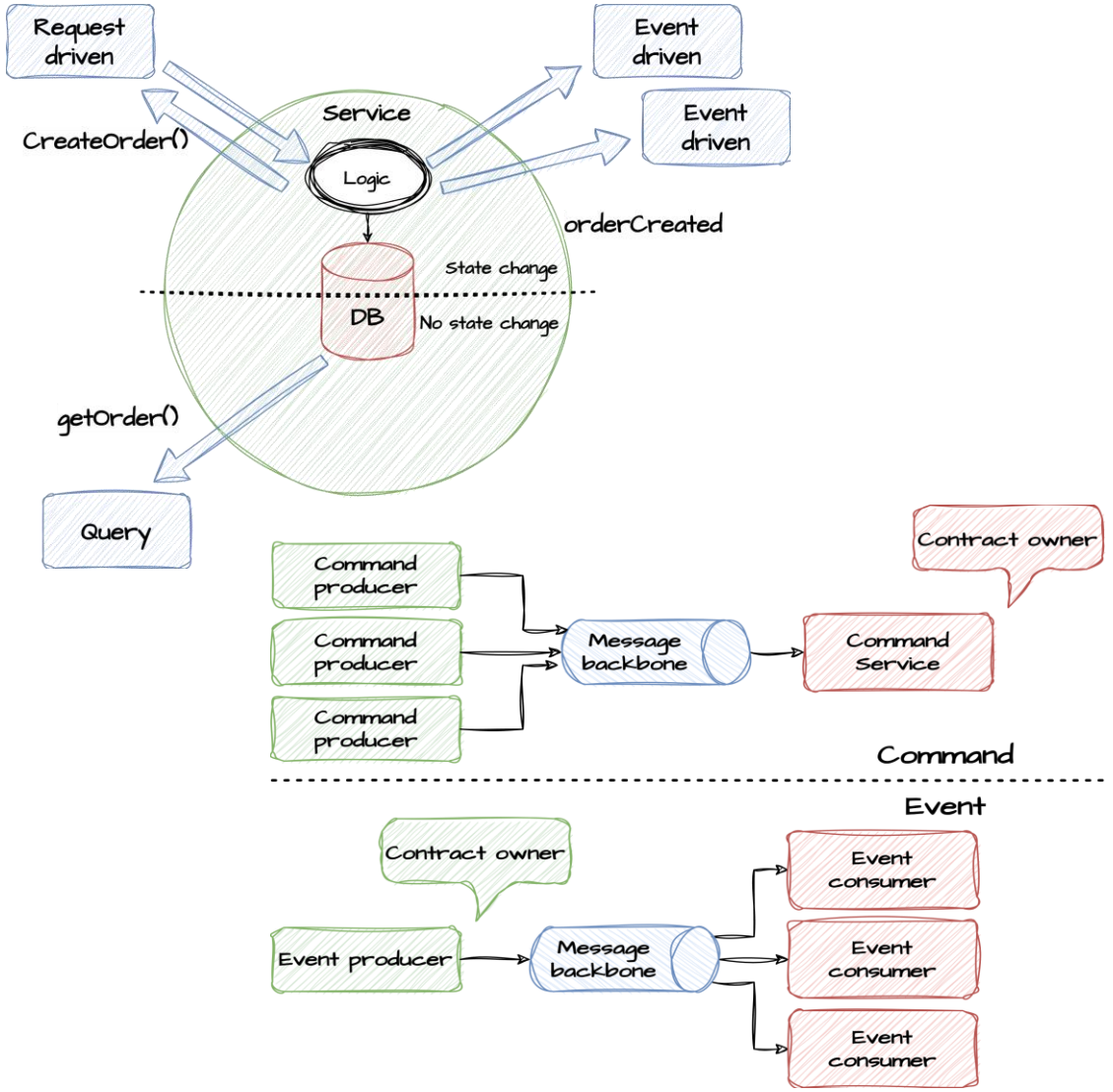
Owned by consumer

Queries

Peer-to-peer. Typically in the form of HTTP calls

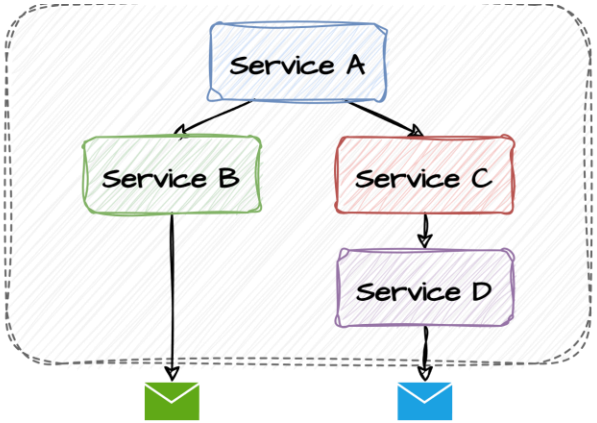
No side-effects (state changes) in the service.

Read-only and typically backed by a data-store



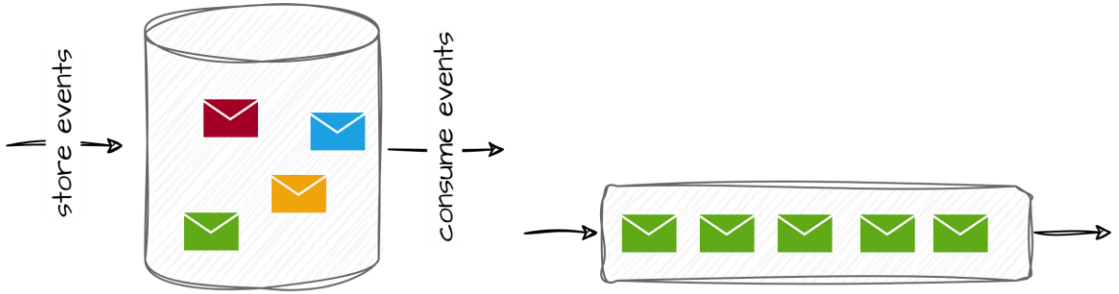
Event communication patterns

Lund&Bendsen
developing developers



Process Manager

Central unit that
orchestrates communication



Event Sourcing

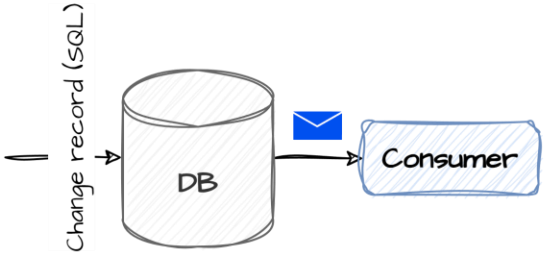
Store event and build
current state using events

Event streaming

Unbounded stream of events

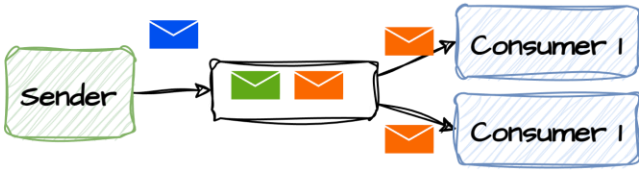
Point-to-point

Sender put message on queue
Receiver reads from queue and deletes message



Change data capture

Consume changes directly from DB



Pub/Sub

Publish same message to
many consumers. Messages preserved

Events

Lund&Bendsen
developing developers

Events is about something that has happened. They are **ordered** and **immutable** and once published cannot be changed or re-ordered. **Multiple consumers** can subscribe to these events. Event may carry more (**fact**) or less (**delta**) information which is a trade-off between high-coupling due to additional lookups (consistency) or risk of carrying stale data but with no additional lookup (availability).

Events should only carry data that it owns. To prevent model sprawl, its the responsibility of the consumer to consume data from other referenced domains.

Internal and External events (for DDD)

Internal (domain) events are used inside the domain whereas **external** (integration) events are used between domains

Delta event

Only communicate changes (deltas, provide context for the event past tense)

Medium risk of breaking data contract

Medium coupling

Notification event

Small risk of breaking data contract

Low coupling

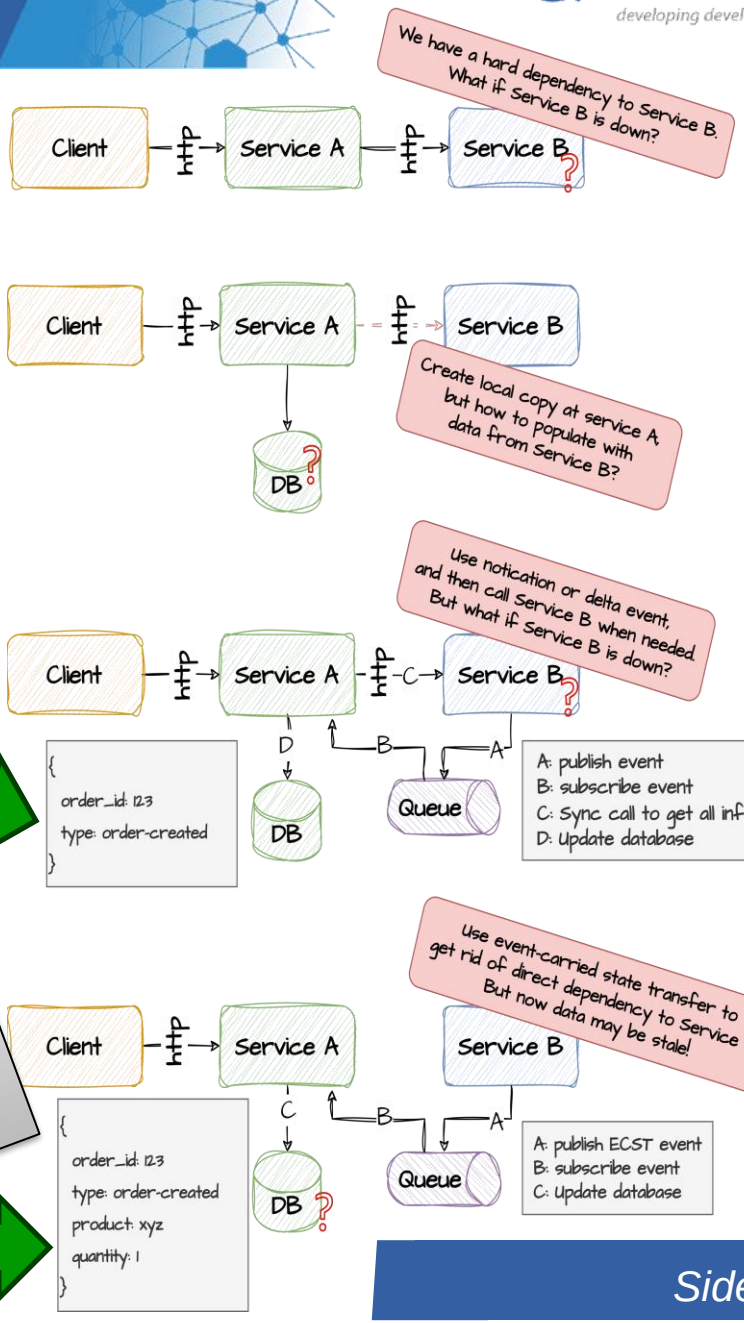
Minimal payload, expect consumers to fetch data at the owner

Event-carried state transfer (fact)

Rich events so no need to fetch additional data fro owner.

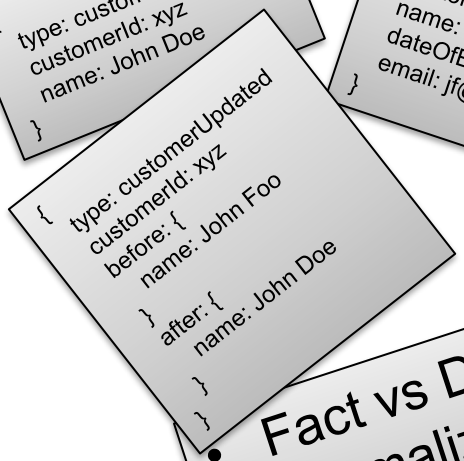
Risk of data becomming stale and breaking data contracts

Low coupling



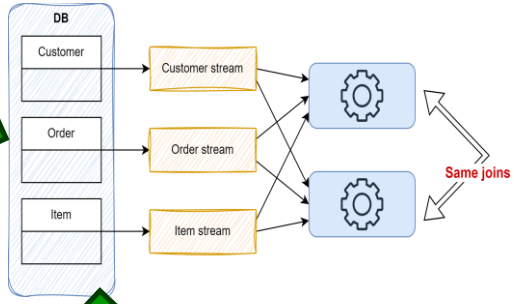
Designing events

Lund&Bendsen
developing developers

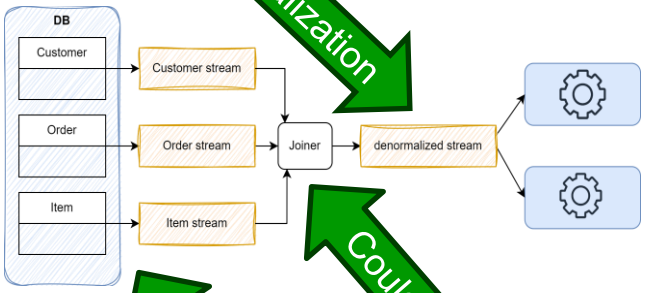


- Fact vs Delta Event types
- Normalization vs Denormalization
- Single vs Multiple Event types pr topic

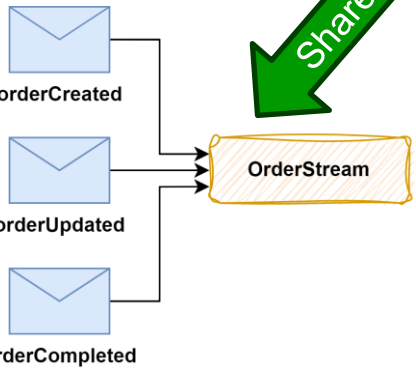
High coupling to datasource



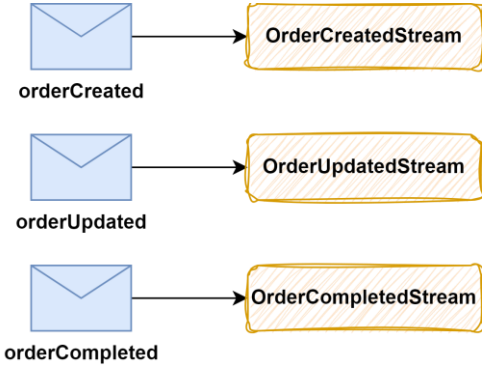
De-normalization



One topic pr event



Shared topic



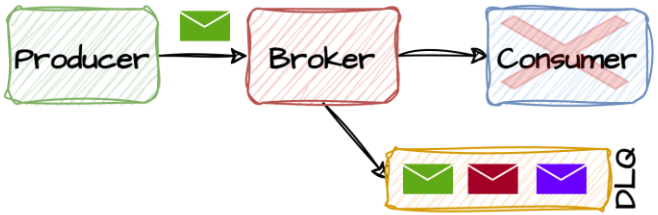
CDC

Could also be trans outbox

Order of event types is lost

Event delivery failures

Lund&Bendsen
developing developers

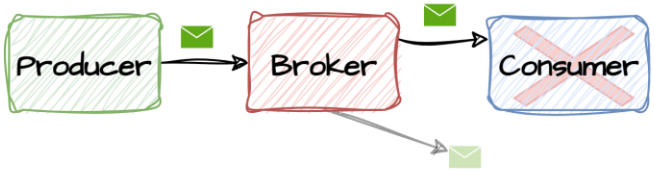


Dead-letter queues

Event fails to reach target, message can be sent to special queue for retry/inspection

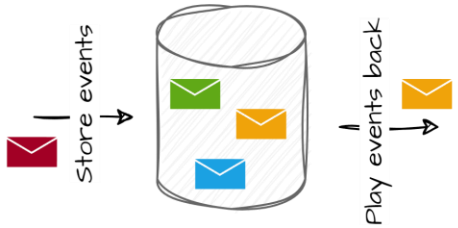
Redrive policies

Sometimes the broker may retry delivery for you based on certain retry criterias



Dropping events

Sometimes it makes sence to just drop events if they cannot reach target system



Archive / replay

Store events and retry later
Make sure consumers are idempotent

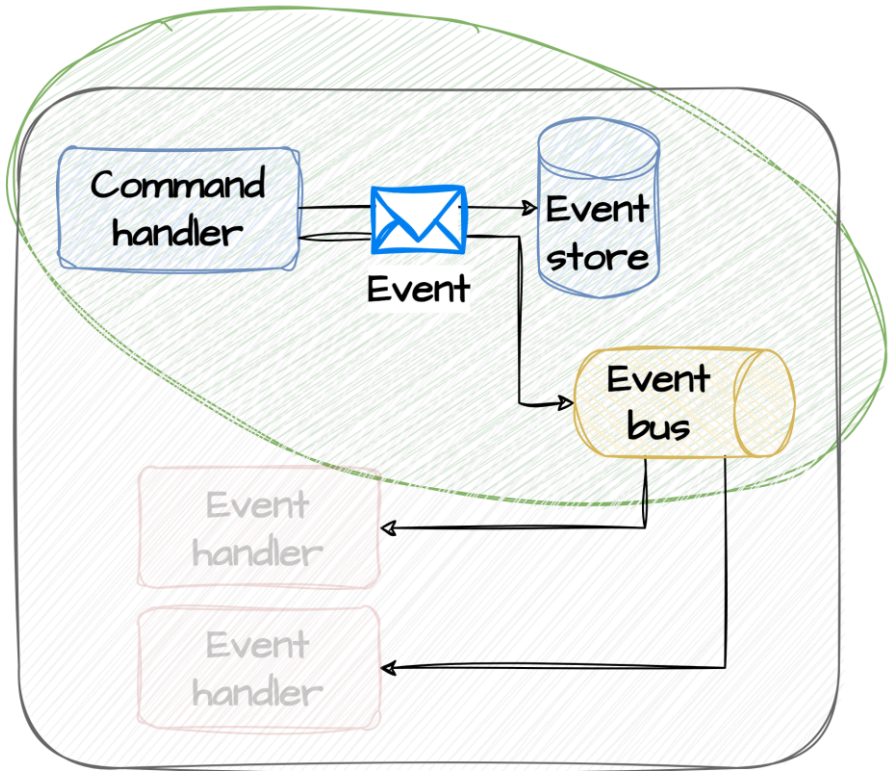
Event Sourcing

Lund&Bendsen
developing developers

Event sourcing is a way of modeling and persisting events (typically delta events) as a series of **immutable** and **ordered** events rather than focusing on the current state of the entity behind the event.

Events are stored in an event store which is local to the service (bounded context) that it lives in. Events in the event store can be replayed and thereby used for state reconstruction or used for auditing or debugging purposes.

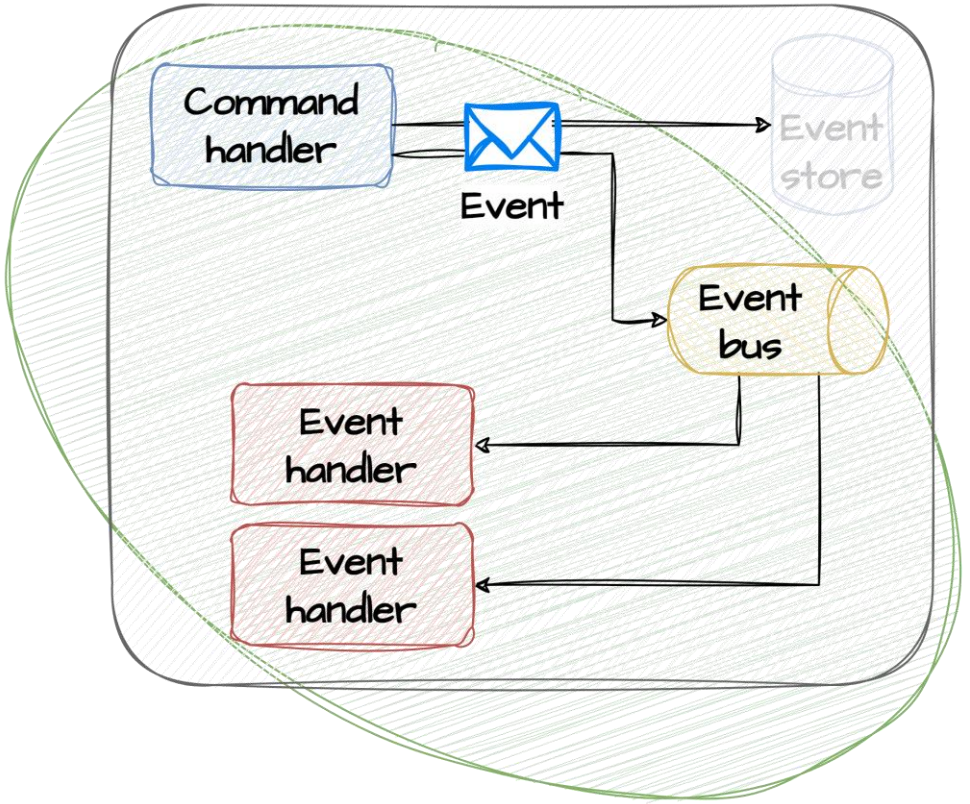
To optimize replay and state reconstruction sometimes snapshots are made. When recreating state, you find the latest snapshot and apply all changes made since then.



Event Streaming

Lund&Bendsen
developing developers

Event streaming is an **ordered**, **unbounded** sequence of **immutable** events flowing from producers to consumers through an event backbone with an **at-least-once** delivery guarantee. Whenever a service or bounded context makes changes to an entity that it owns, this can be published as an event on an event backbone. The event may carry context depending on the design. Event streaming uses a pub-sub approach to enable decoupled communication between systems. Consumers subscribe to a topic/channel on the event pipeline, and producers produce events to these topics/channels. Services may act as both consumers and producers of events. The pub-sub design decouples the producers and consumers, making it easier to scale each part of the system individually.



Event Projection - CQRS

Lund&Bendsen
developing developers

CQRS, which stands for **C**ommand **Q**uery **R**esponsibility **S**egregation, is an architectural pattern used to separate the concerns of handling commands (write operations) and queries (read operations) within an application.

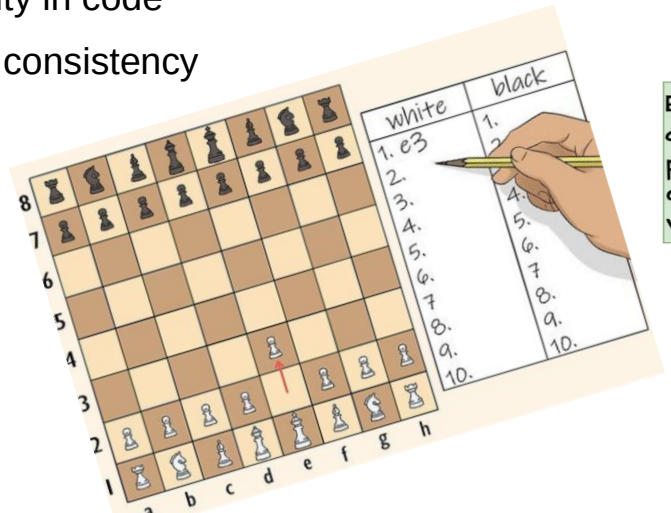
By separating read/write operations this way you can optimize each part independently. Read operations typically have different forms and transformers (event handlers) can project the same write operations into different projections that are read optimized.

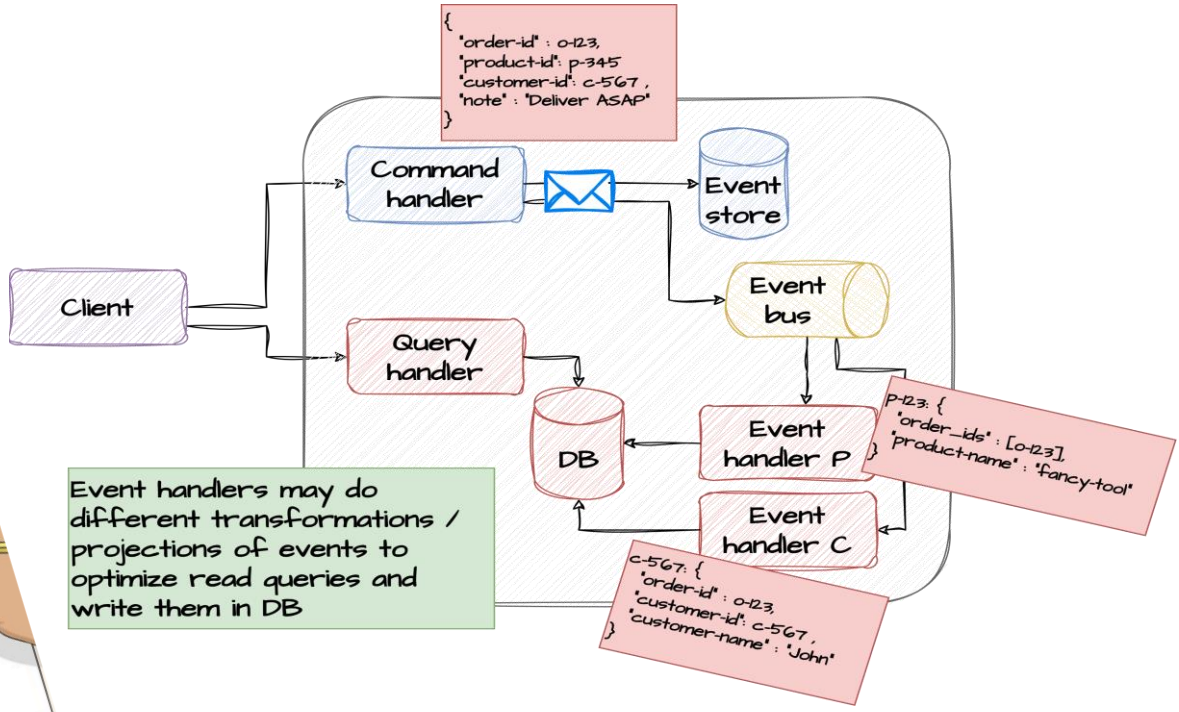
Advantages:

- Scalability – Separate read/write operations
- Flexibility – Create different projections

Disadvantages:

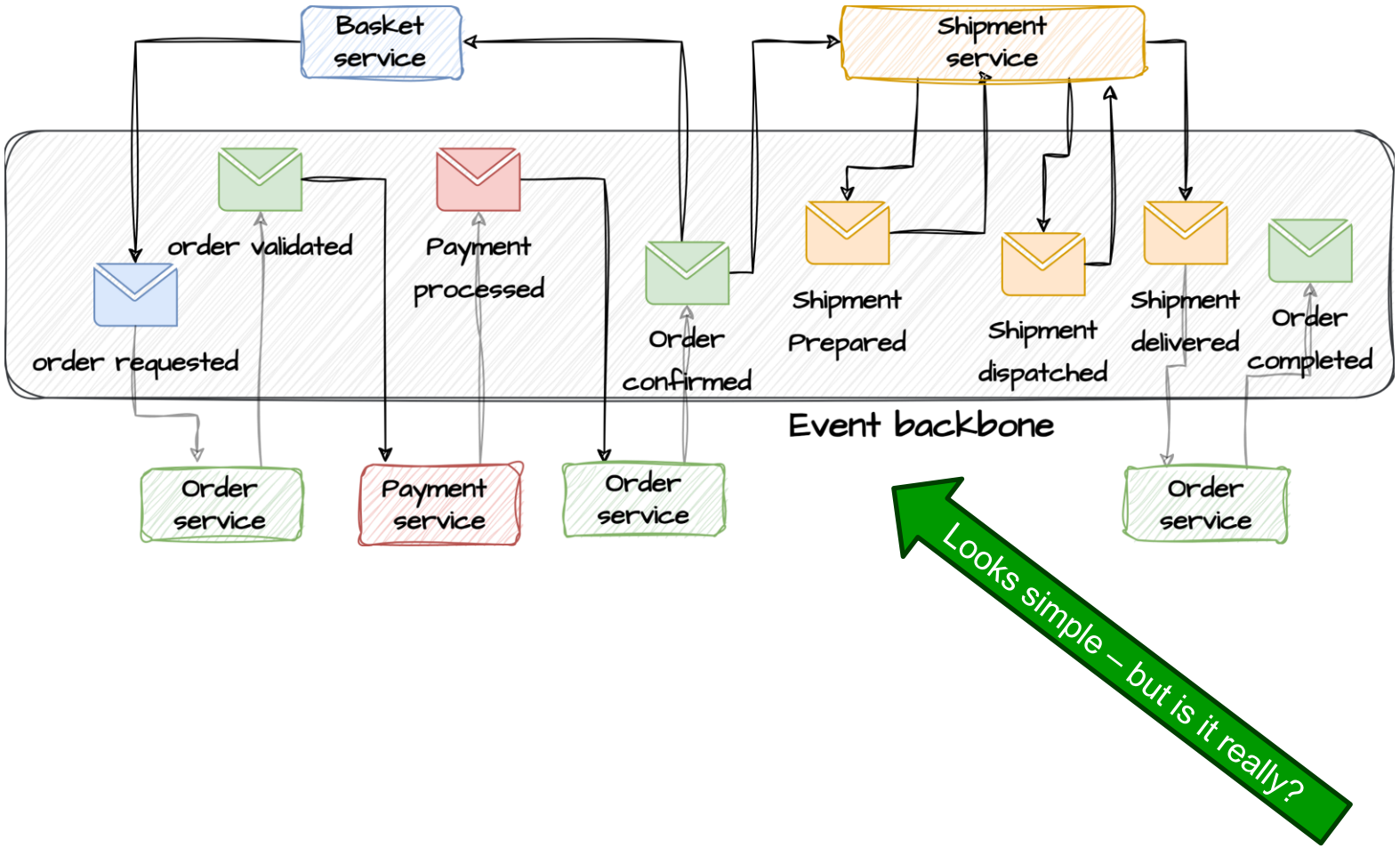
- Complexity in code
- Eventual consistency





EDA at a larger scale

Lund&Bendsen
developing developers

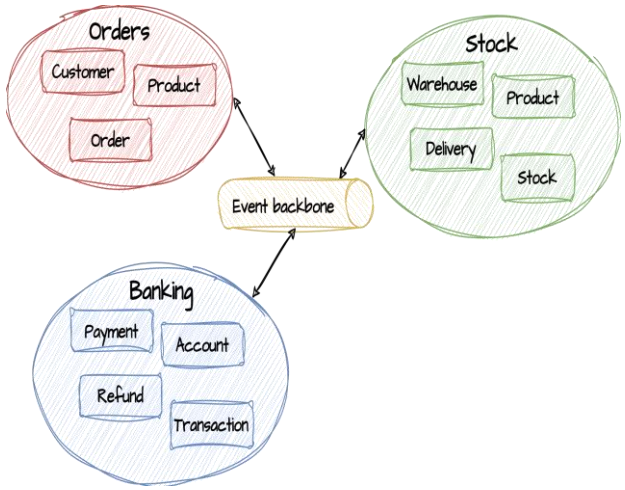


DDD and Bounded contexts

Lund&Bendsen
developing developers

DDD (domain driven development) is about technical- and domain model alignment so that domain experts and technical team members working together can communicate about the model in an unambiguous way.

In DDD, **bounded contexts** help to establish clear boundaries between different parts of the system by grouping related domain **entities**. This allows for better separation of work as smaller teams can focus on sub domains. Inside a boundary, the model must be **consistent** and **coherent**, meaning that the entities must be inter-related and dependent on each other. Messages sent between services inside a bounded context is referred to as **domain events**.



The same entity name may occur within different bounded contexts (like Product in Orders and Stock).

A public language (model) needs to be specified when communicating between bounded contexts.

Events sent between bounded context are referred to as **integration events**. Sometimes translations to/from integration events must happen before sending to the backbone or receiving from the backbone if the producing or consuming bounded context is not aware of integration event language.

Messages between bounded context

Lund&Bendsen
developing developers

Conformist

Don't translate message, consume as is

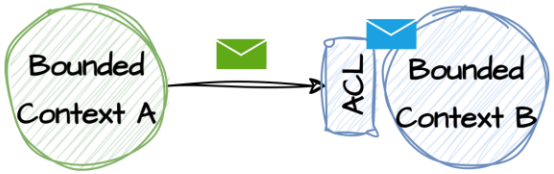
Open-Host Service

Producer does not conform to contract

Messages get translated prior to publishing

Enrich

Somtimes you want consumers getting more info vs producers providing it



Anti-corruption

Consumer does not conform to contract

Messages get translated before consumption