

Microservices

Event Driven Architecture

Hönnun og smíði hugbúnaðar

Haust 2022



Hvað er Event Driven Architecture?

- Service hafa samskipti með *events*
- Eitt service (***producer***) publish-ar event-i og önnur (***consumers***) bregðast við þeim
- Sá sem publish-ar veit ekki hverjir eða hvort einhver er að hlusta
- Asynchronous
- Eins decoupled og það getur verið

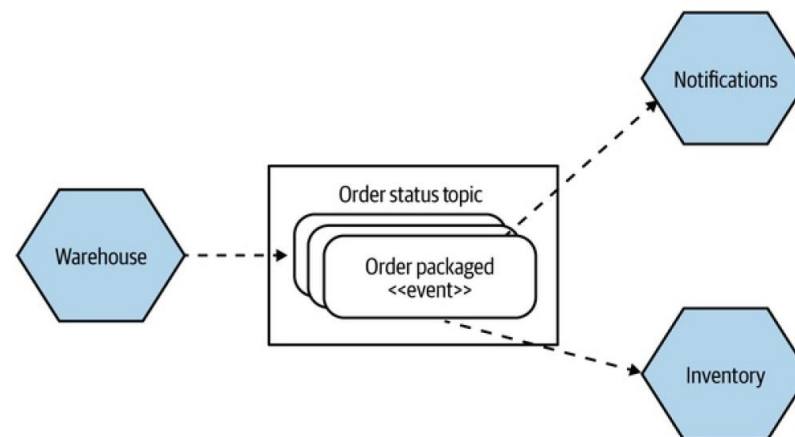


Figure 4-11. The Warehouse emits events that some downstream microservices subscribe to

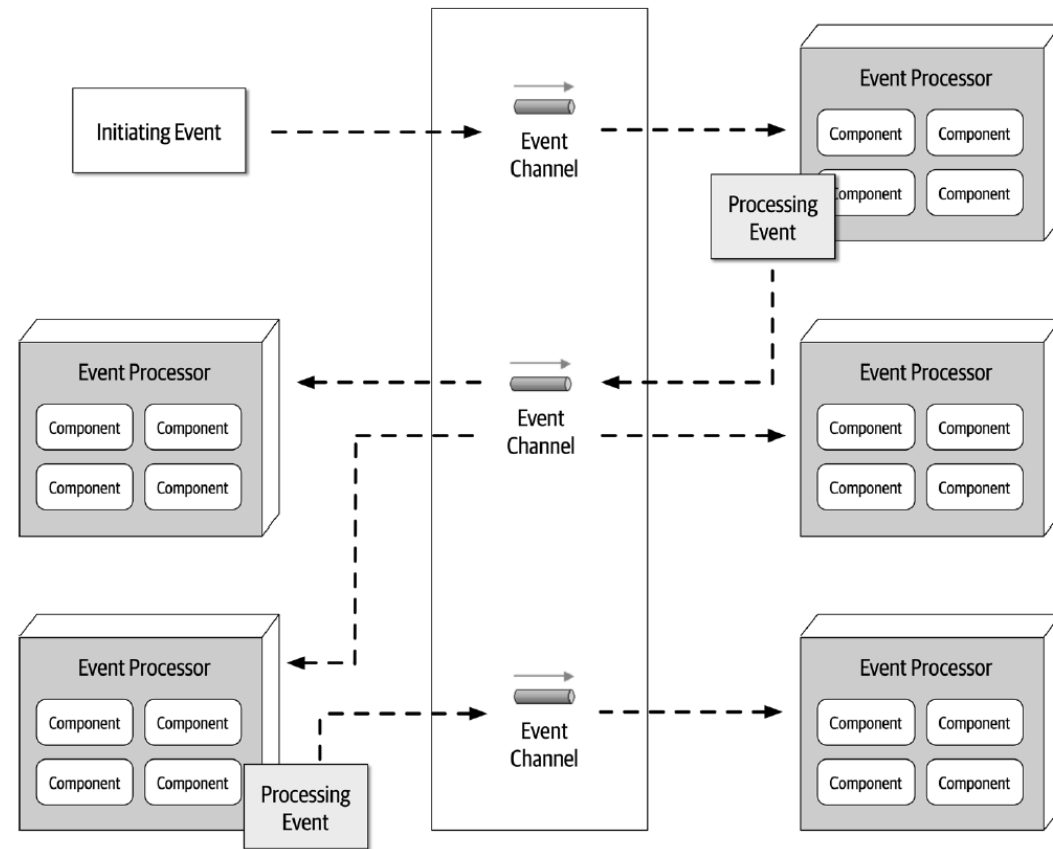


Figure 14-2. Broker topology

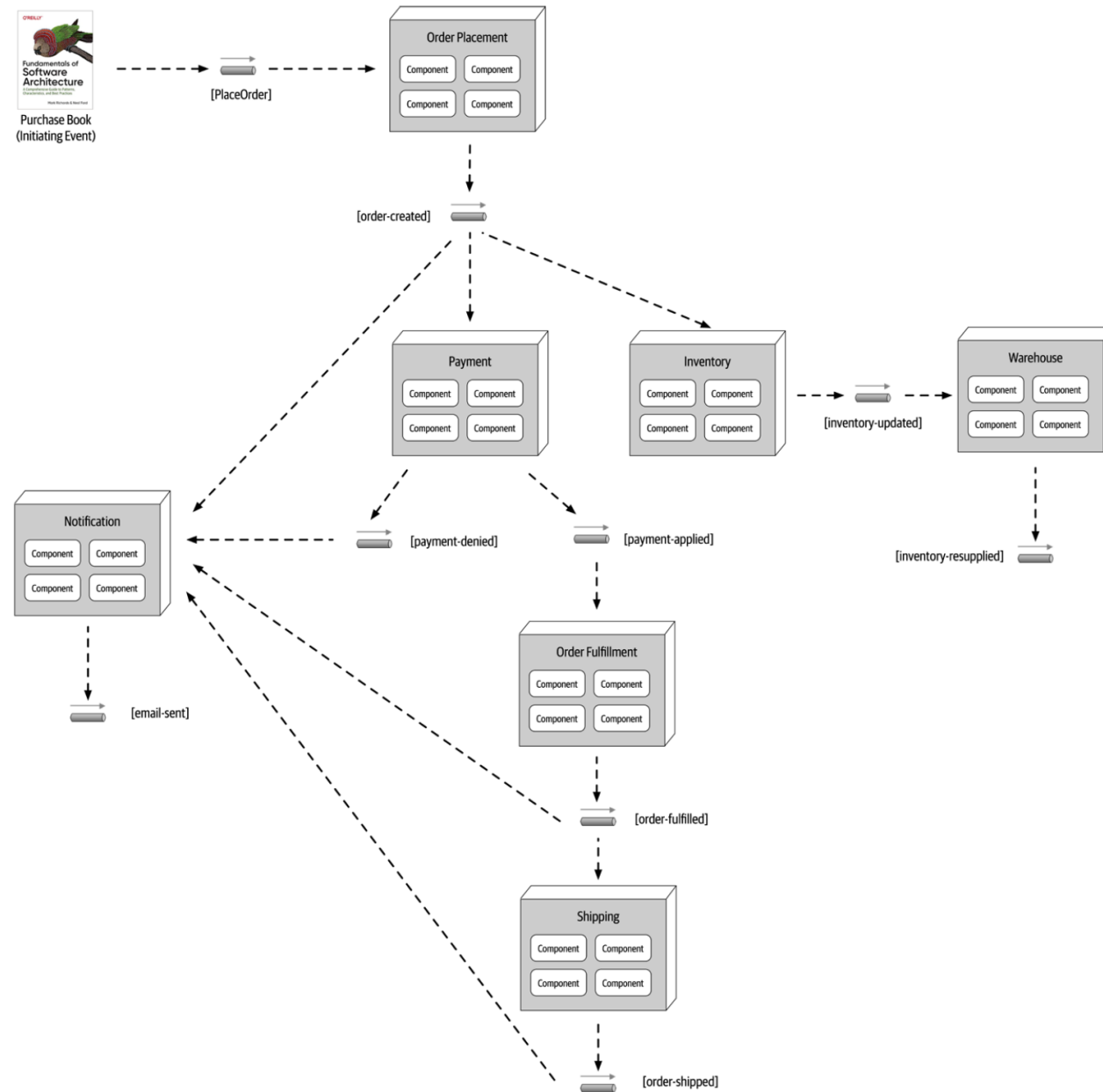
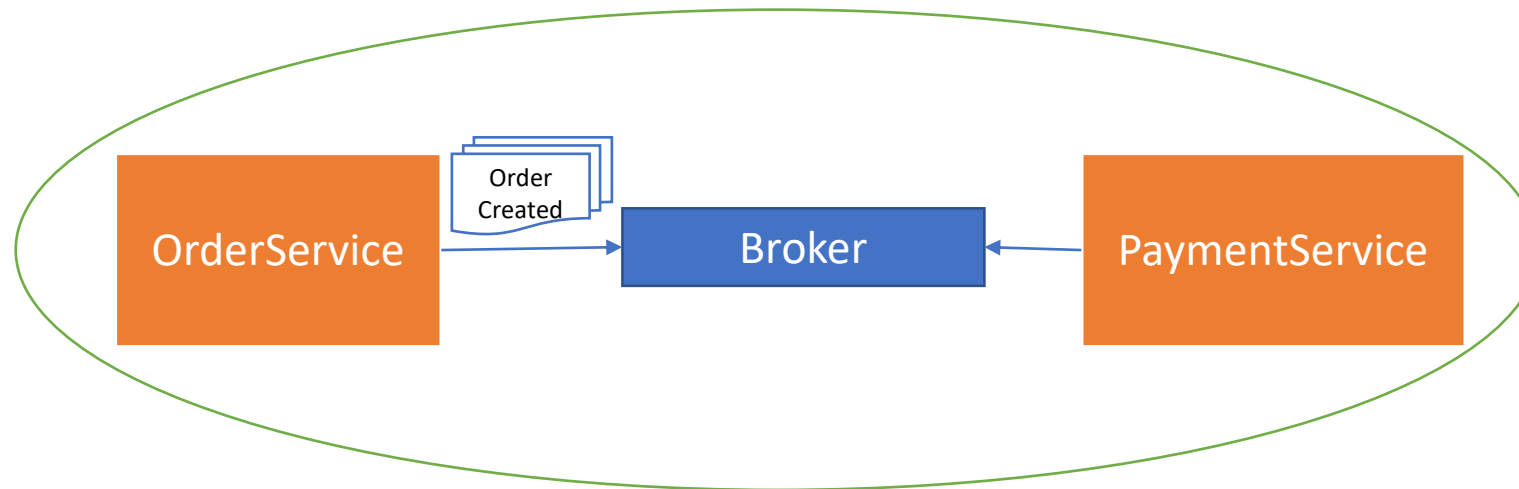


Figure 14-4. Example of the broker topology

Inversion of Responsibility

- Service a kallar ekki lengur í service b
- Service b hlustar núna eftir event-um frá service a
- Coupling og ábyrgð snúin við
- Leiðir til betri *logic* dreifingu
- Í raun sama hugmynd og Dependency Inversion Principle

Inversion of Responsibility Frh.



Brokers

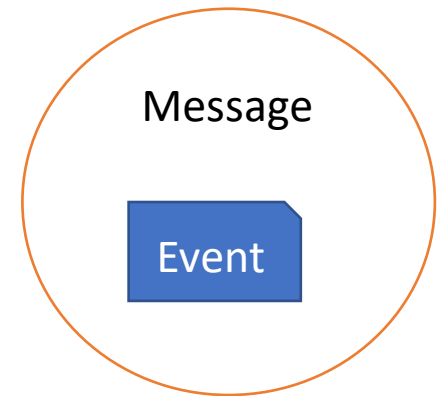
- Tæknin á bakvið event driven kerfi
- Kerfi sem fær, geymir og útvegar event-in
- Producer sendir event á broker-inn
- Consumer fær event frá broker-num
- Broker-inn geymir event-in í queue eða *partitioned stream*

Brokers frh.

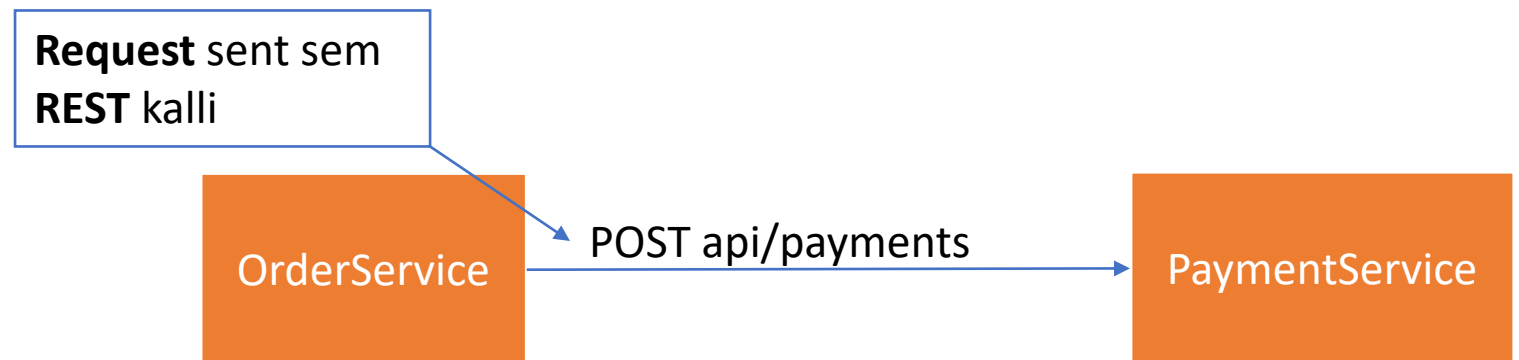
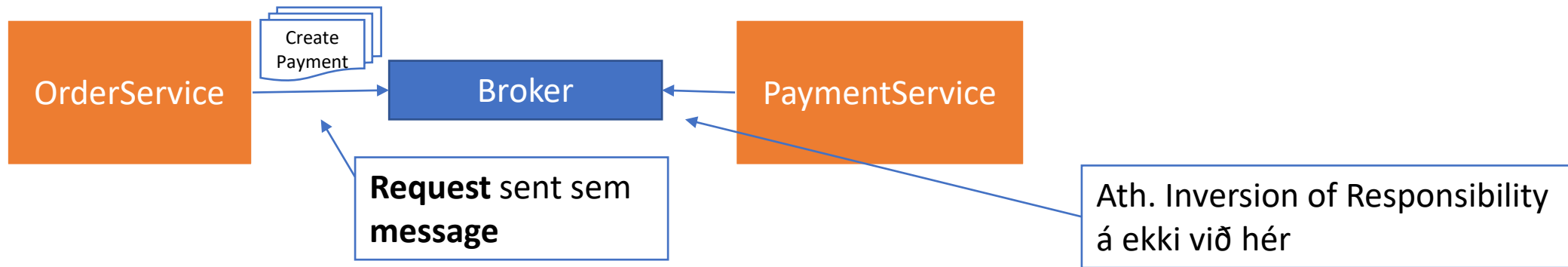
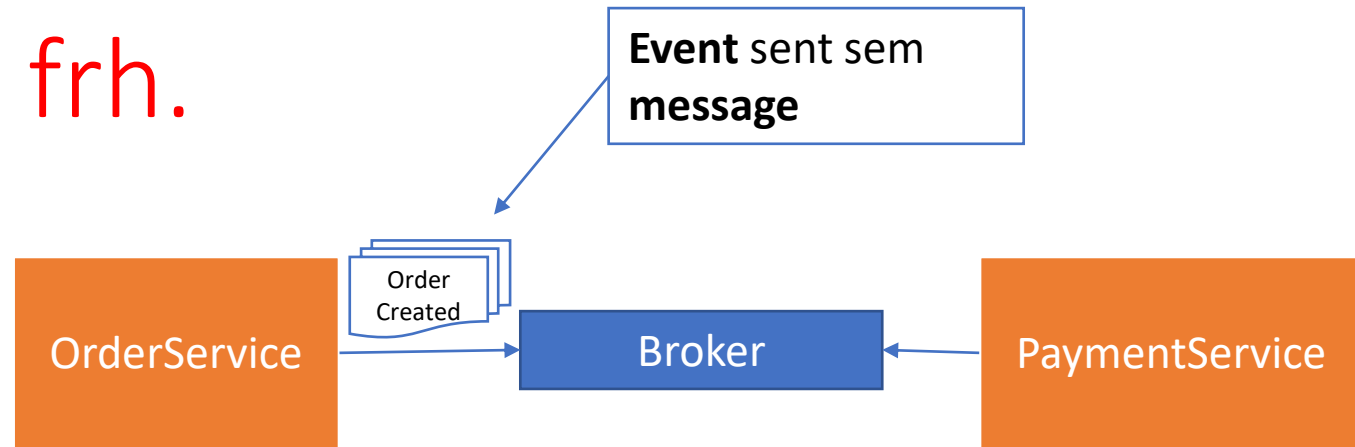
- Broker **samanstendur af mörgum dreifðum broker instances**
- Hefur hátt **scalability**
 - Hægt að bæta við fleiri broker instances eftir álagi
- Útvegar hátt **durability**
 - Gögn dreifð á milli broker instances / nodes
 - Ef broker instance deyr þá eru gögnin ekki töpuð
- Hefur átt **performance**
 - Þarf að geta höndlað hundruð þúsundir af writes & reads á sekúndu
- Hefur hátt **resilience**
 - Eitt broker instance getur farið niður og kerfið virkar ennþá eðlilega

Events vs Messages

- Oft ruglað saman / sett undir sama hatt
- Message er eitthvað sem við sendum með broker
- Messages geta verið notuð til að senda async requests
- Event eru líka send sem messages
- Event er staðreynd / gögn um eitthvað sem gerðist



Events vs Messages frh.

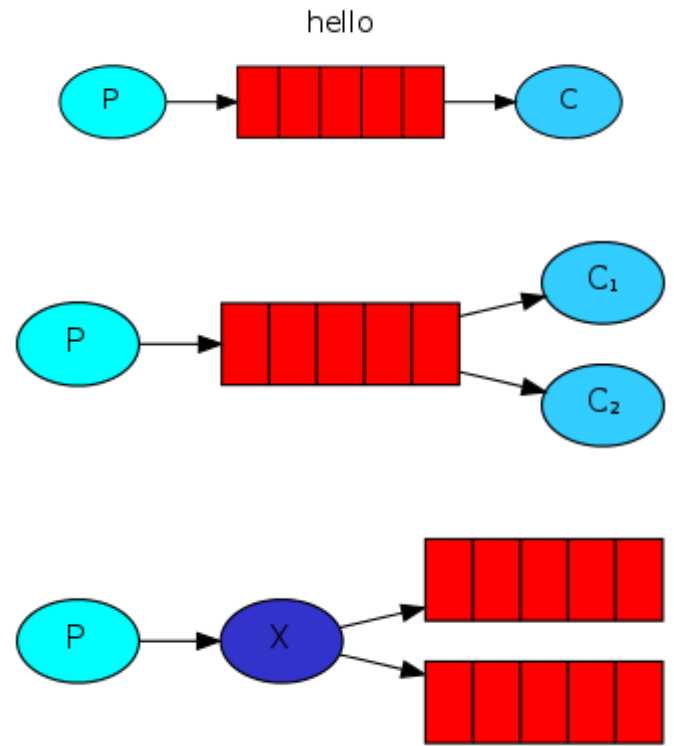


Event Broker vs Message Broker

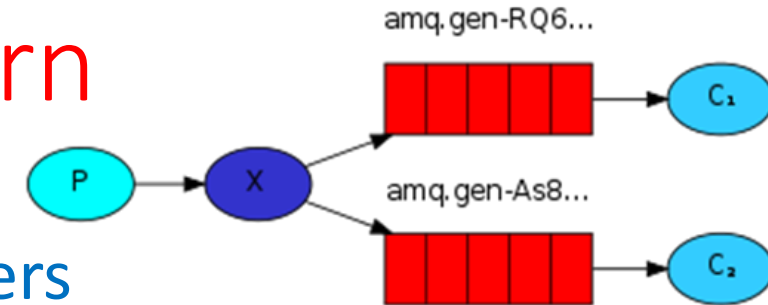
- Message Brokers sérstaklega hannaðir með *messages* í huga
- Event Brokers sérstaklega hannaðir með *events* í huga
- Requests náttúrulegri með Message Brokers
- Events náttúrulegri með Event Brokers
- Bæði Event og Message Brokers geta þó ráðið við bæði messages og events

Message Broker

- Útfært sem **Queue**
- Producer publish-ar message í queue-ið
- Consumers keppast um að pop-a af queue-inu
- Messages eru eydd eftir að þau eru consumed
- Observer patternið og events ónáttúrlegri með Message Brokers
- Requests sem messages eru náttúrulegri en með Event Brokers

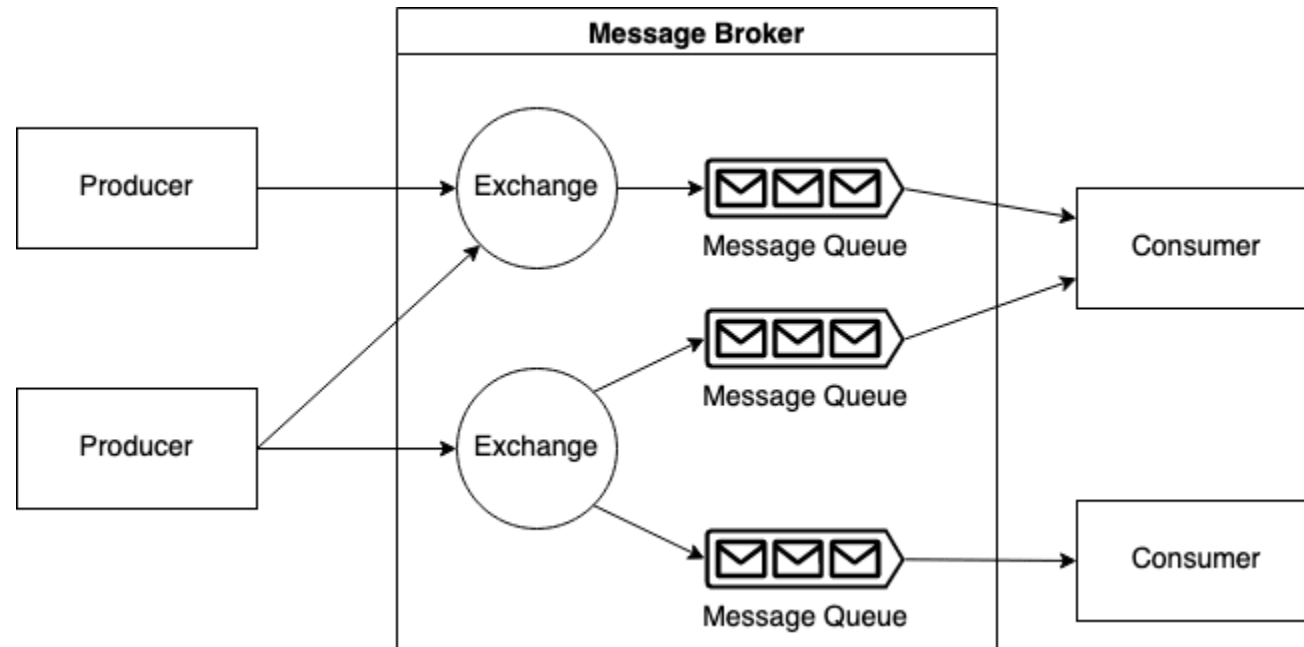


Message Broker – Observer Pattern



- Observer Pattern ónáttúrlegra með Message Brokers
- Observer pattern-ið útfært með því að publish-a í mörg Queue
- Mismunandi consumer groups pop-a þá af eigin queues
- Hvert consumer group fær þannig message-ið amk einu sinni
- Hvert group getur síðan haft einn eða fleiri consumers / instances

Message Broker – Observer Pattern

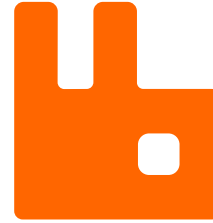


RabbitMQ



- Vinsælasti **message broker-inn** <https://rabbitmq.com/>
- **Queue**
 - First-In-First-Out by default en hægt að stilla með priority
 - Message eytt af queue þegar búið að consume-a
 - Queue-in geta verið durable / persistent / replicated
- **Exchanges**
 - Sjá um að route-a message-um á queues
 - Nokkrar tegundir → direct, topic, headers, fanout

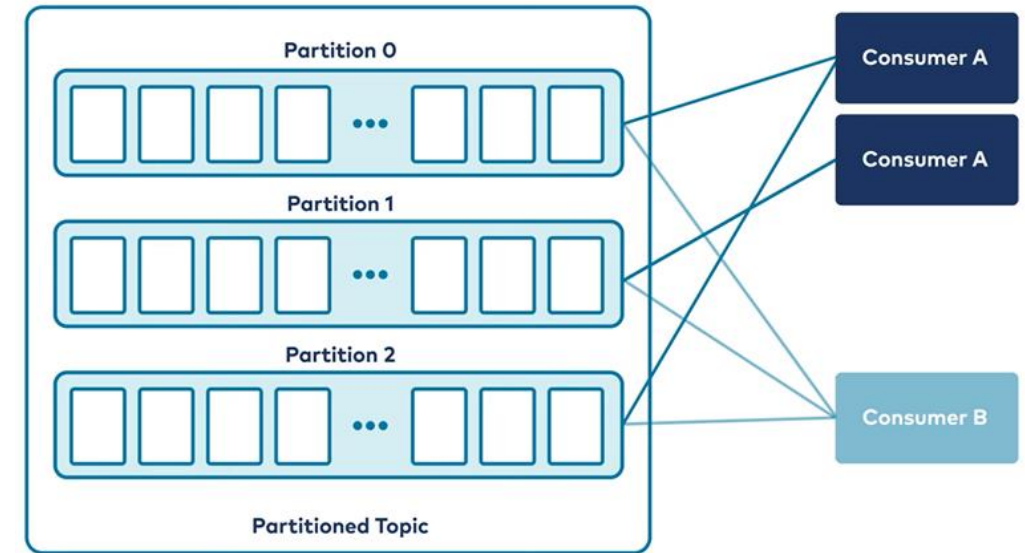
RabbitMQ pros and cons



- Kostir
 - Einfalt
 - Engin transaction stuðningur
 - Complex routing
 - Queue virkni
- Gallar
 - Ekkert schema support
 - Styður events ekki jafn vel
 - Styður observer patternið ekki jafn vel
 - Ekkert replayability
 - Engin transaction
 - Ekki hægt að hafa sem *single source of truth*

Event Broker

- Útfært sem óbreytanlegur log af event-um
- Producer publisher event-i á log-inn
- Consumers subscribe-a á breytingar á log-inum
- Consumers consume-a af sama log
- Opna möguleika fyrir streaming vinnslu
- Observer patternið og events náttúrlegt með Event Brokers
- Requests sem messages eru ónáttúrulegri en með Message Brokers



Event Broker - Eiginleikar

- Partitioning

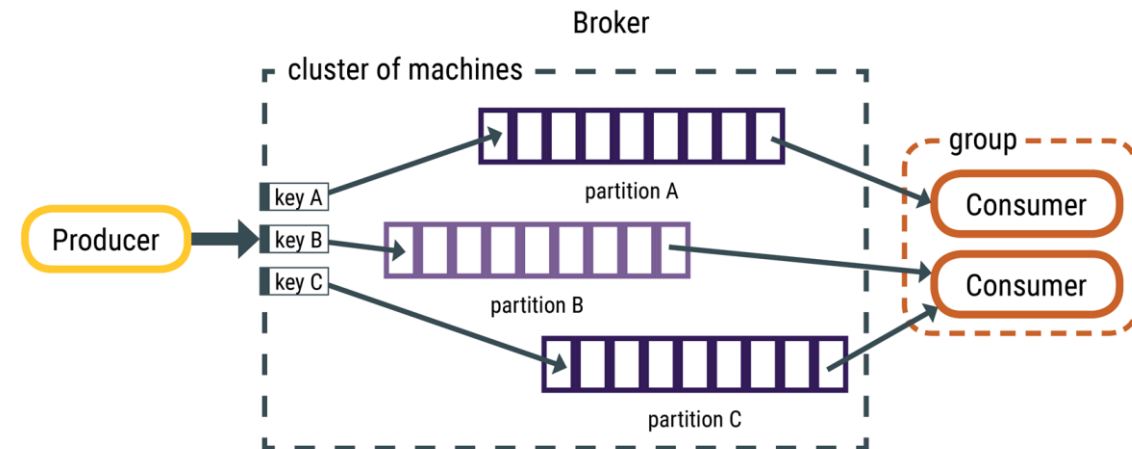
- Hægt að skipta upp event log straum
- Margir consumers í sama consumer group geta því skipt eventum á milli sín
- Getur virkað sem hálfgerð *gervi queue*
- Event með sama lykil fara á sama partition

- Strict Ordering

- Event innan partition eru í réttri röð

- Immutability

- Event í log eru óbreytanleg



Event Broker – Eiginleikar frh.

- Indexing

- Consumers geta valið hvar í log-inum þeir vilja byrja að consume-a
- Nýjir consumer-ar geta t.d. byrjað frá byrjun

- Infinite Retention

- Events eyðast ekki af log-inum eftir consumption
- Hægt að stilla hversu lengi events geymast (getur verið endalaust)

- Replayability

- Hægt að endurspila events
- Gamall consumer getur endurspilað events t.d. ef villa kemur upp



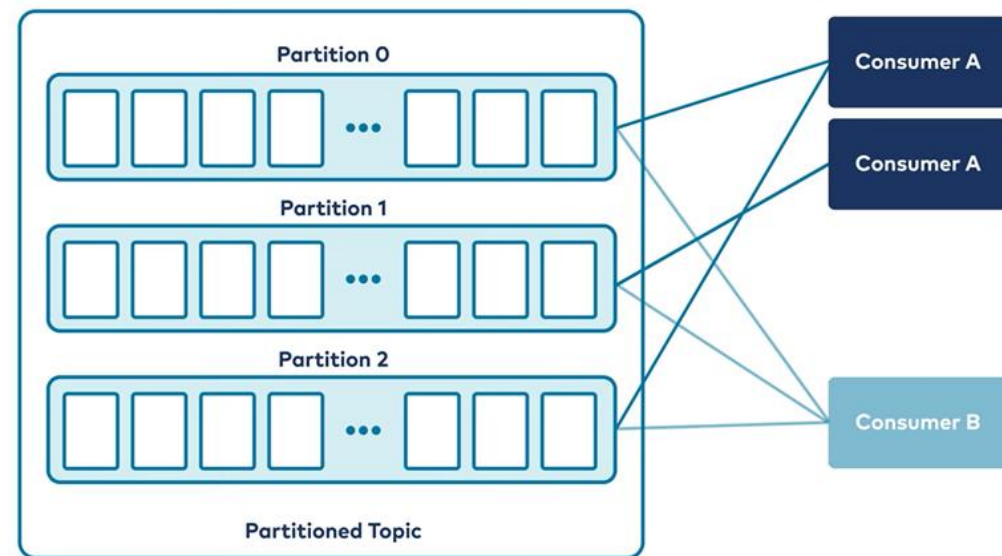
- Vinsælasti Event Broker-inn <https://www.confluent.io/>

- Topics

- Event skipt upp eftir Topics
- Hvert Topic er síðan partitioned
- T.d. Order Created Topic

- Partitions

- Hvert topic skipt upp í partitions
- Partitions skipt á meðal consumer groups



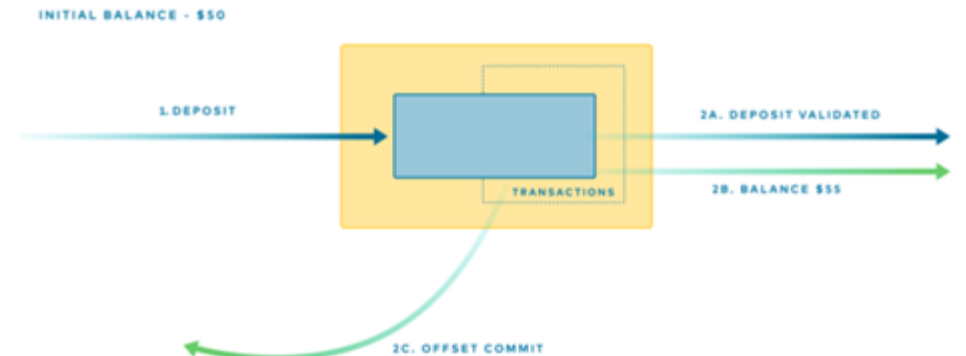
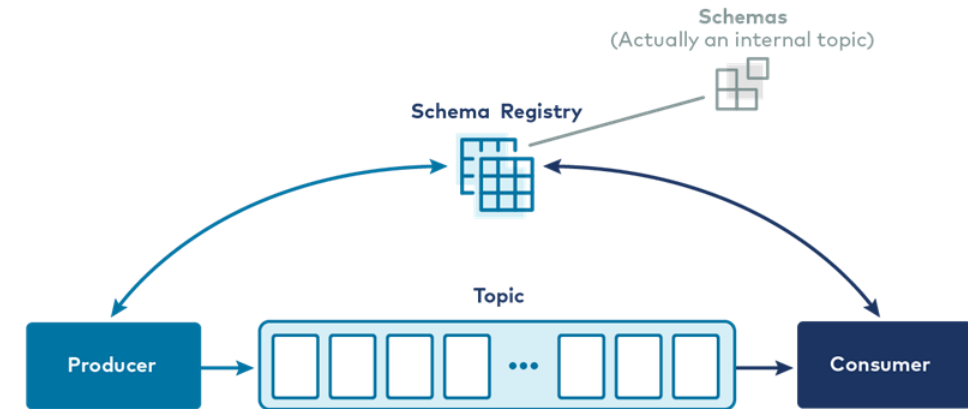


- Schema Registry

- Getum skilgreint schema fyrir topic
- Getum tryggt að event er á réttu formi
- Getum tryggt backward compatible breytingar
- Styður Json, Avro og Protobuf

- Transactions

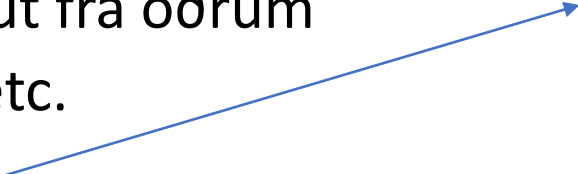
- Hægt að produce-a og consume-a events sem





- Kafka Streams

- Kafka býður upp á streaming virkni
- Hægt að aggregate-a streams
- Hægt að búa til streams út frá öðrum
- Hægt að filter-a, map-a etc.
- Hægt að nota SQL syntax



```
-- pq1
CREATE STREAM high_readings AS
  SELECT sensor,
         reading,
         UCASE(location) AS location
  FROM readings
  WHERE reading > 41
  EMIT CHANGES;
```

Kafka pros and cons



- Kostir

- Hátt performance
- Náttúrulegt observer pattern
- Náttúrulegt fyrir events
- Replayability
- Schema registry
- Transactions
- Event eyðast ekki

- Gallar

- Flóknara
- Ekki jafn góður stuðningur fyrir *queue* virkni

Kafka vs RabbitMQ

Samanburður	RabbitMQ	Kafka
Release Date	2007	2011
Message Ordering	Styður ekki á milli margra consumer-a en styður fyrir hvert queue	Styður ekki á milli partitions en styður fyrir hvert partition
Message Lifetime	Fara af queue þegar consumed	Eyðast ekki nema eftir stillanlegum tíma
Transactions	Ekki stutt	Stutt
Message Priority	Hægt	Ekki hægt
Performance	Minna	Meira
Event Sourcing	Ekki hægt	Hægt
Third party support	Fínt	Meiri
Schema	Lítið sem ekkert	Schema registry
Complex routing	Góður stuðningur	Ekki jafn góður en fínn
Broker tegund	Message Broker	Event Broker
Message Queue	Það sem RabbitMQ er ætlað fyrir	Ekki náttúrulegt en hægt að fá svipða virkni
Pub/sub	Ekki náttúrulegt en hægt	Það sem kafka
Durability/Persistence/replication	Stutt	Stutt

Order Brokers

- [Pulsar](#)
- [Azure Event Grid](#)
- [Azure Queue Storage](#)
- [Amazon EventBridge](#)
- [Amazon SQS](#)

The Pros

Great Degree of Decoupling

- Service eins vel decoupled og mögulega

Warehouse decoupled
frá Notifications og
Inventory

Warehouse veit ekki einu
sinni af Notifications og
Inventory

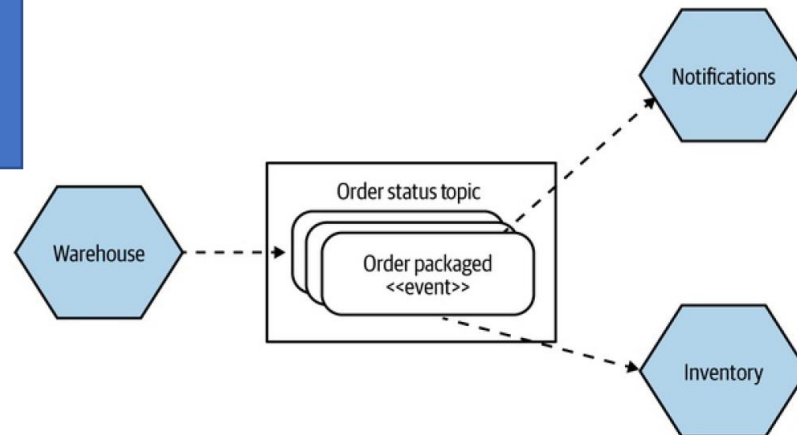
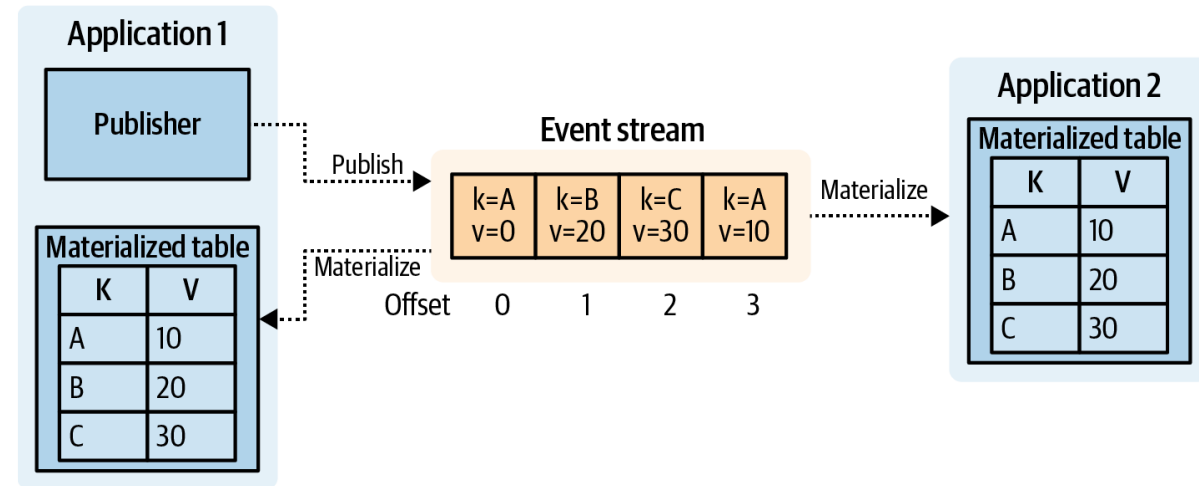


Figure 4-11. The Warehouse emits events that some downstream microservices subscribe to

Coupling snúin við,
Notifications og
Inventory núna coupled
við Order Packaged
event-ið

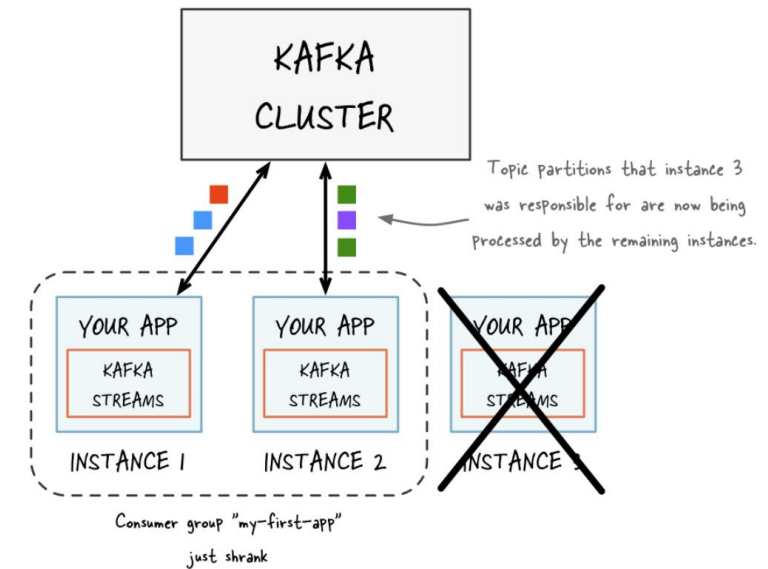
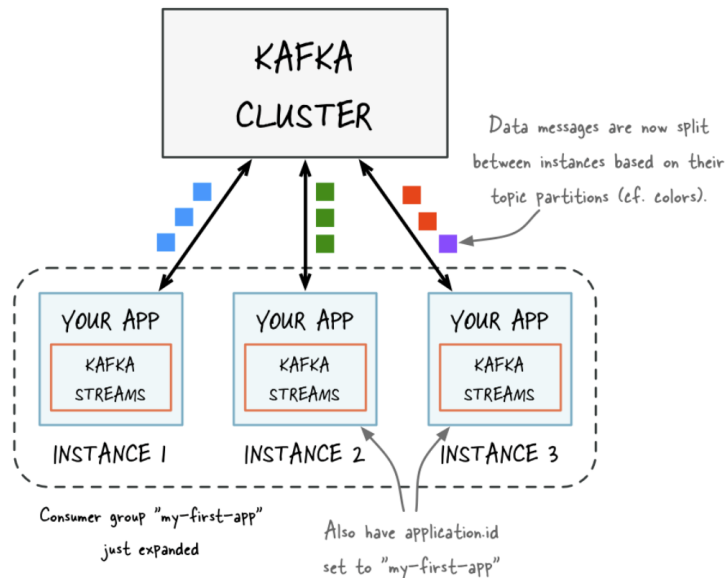
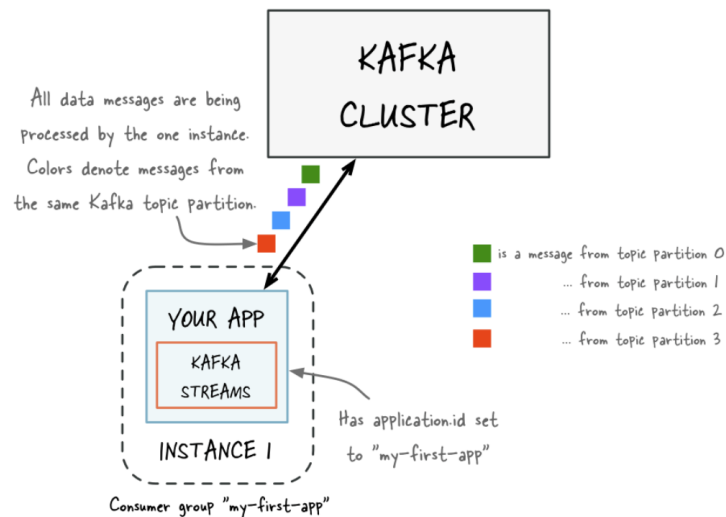
Ease of sharing Data

- Auðvelt að deila gögnum á milli service-a
- Auðvelt að duplicate-a gögn
- Service áhugasöm um gögn subscribe-a á topic fyrir þau gögn
- Event-in verða þannig **source of truth**
- Publisher-inn sá sem hefur **yfirráð yfir gögnum**
- *“Core business data should be easy to obtain”*



Scalability

- EDA hefur mjög hátt scalability
- EDA hefur mjög hátt throughput
- Hægt að bæta við fleiri instances til vinna event-in saman



Performance & Responsiveness

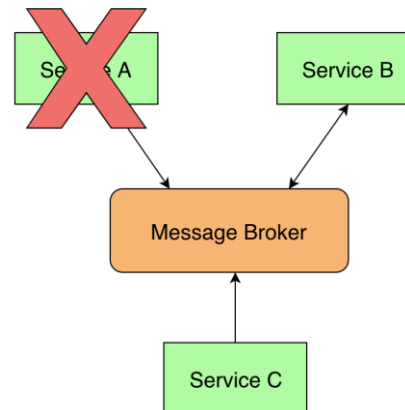
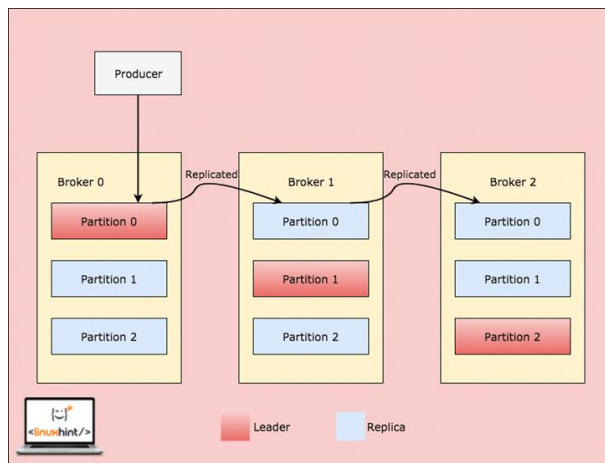
Scalability + throughput + async + observer pattern

=

performance & responsiveness

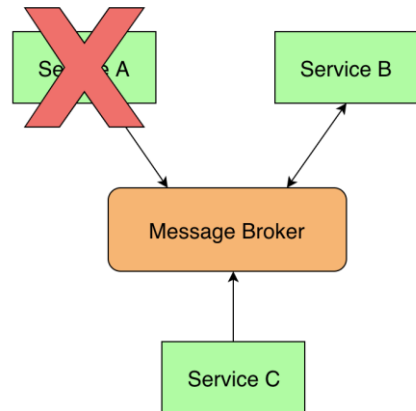
Resilience & Fault Tolerance

- Brokers samanstanda af mörgum einingum
- Ef eitt broker instance fer niður þá fer broker-inn / gögnin ekki með
- Hægt að senda event án þess að consumer er keyrandi



Guaranteed Delivery

- Brokers tryggja að *message* mun skila sér
- Ef consumer er niðri þá fær hann það þegar hann kemur upp
- Guaranteed Delivery \neq Guaranteed Processing

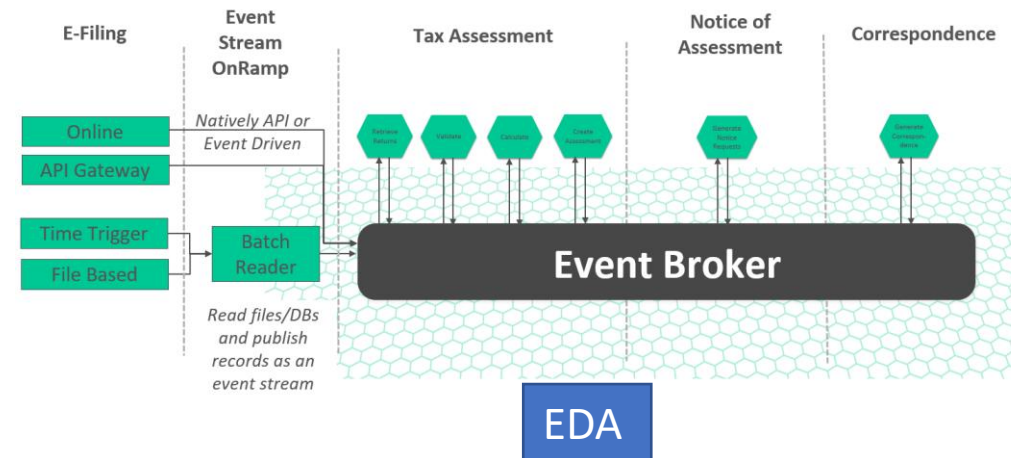
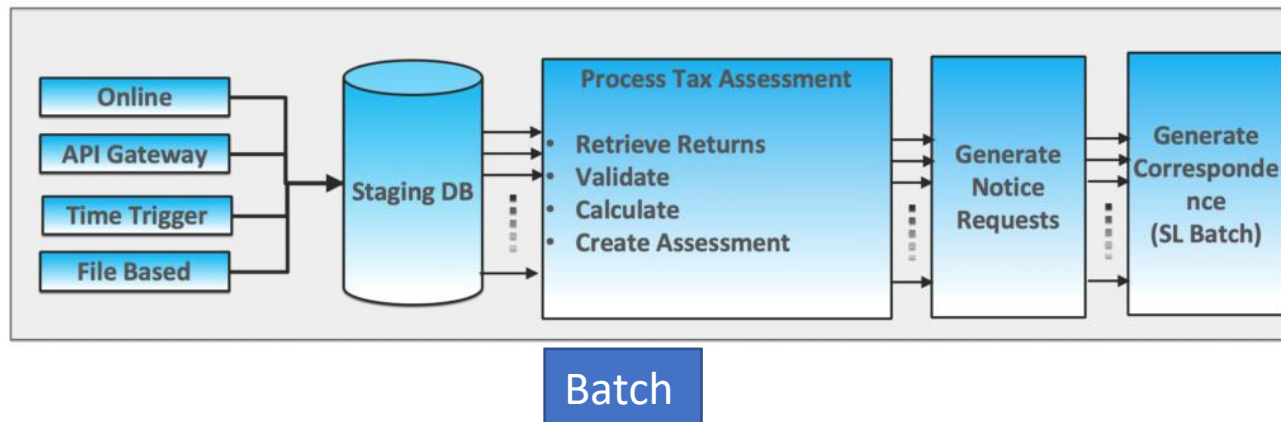


Flexibility & Adaptability

- Lágt coupling leiðir til flexibility og adaptability
- Auðvelt að bæta við nýjum service
- Auðvelt að breyta service
- Auðvelt að fjarlægja service

Batch processing no more

- Getum forðast Batch processing með EDA
- Fáum í staðinn *real-time* processing með event streaming



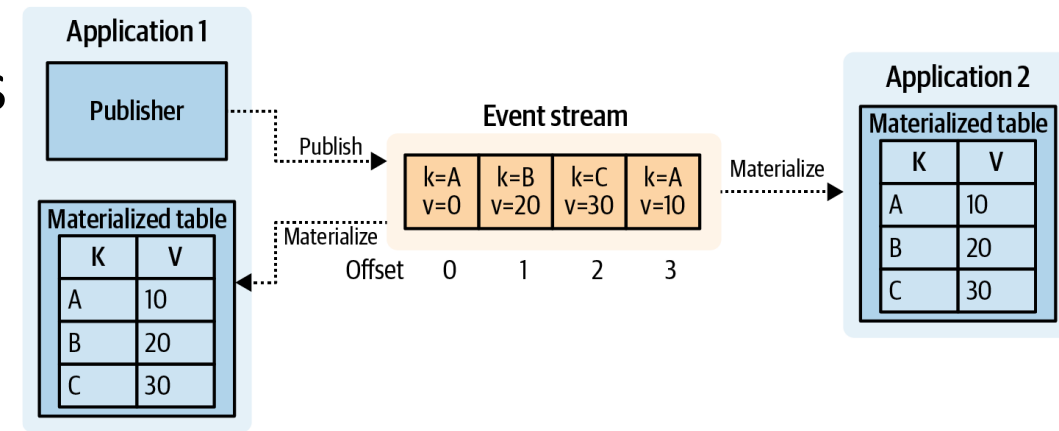
<The Cons>

Eventual Consistency

- Með distributed architecture fórnnum við data consistency / integrity

- Getum ekki lengur notað ACID transactions

- Sömu gögn oft dreifð um kerfið



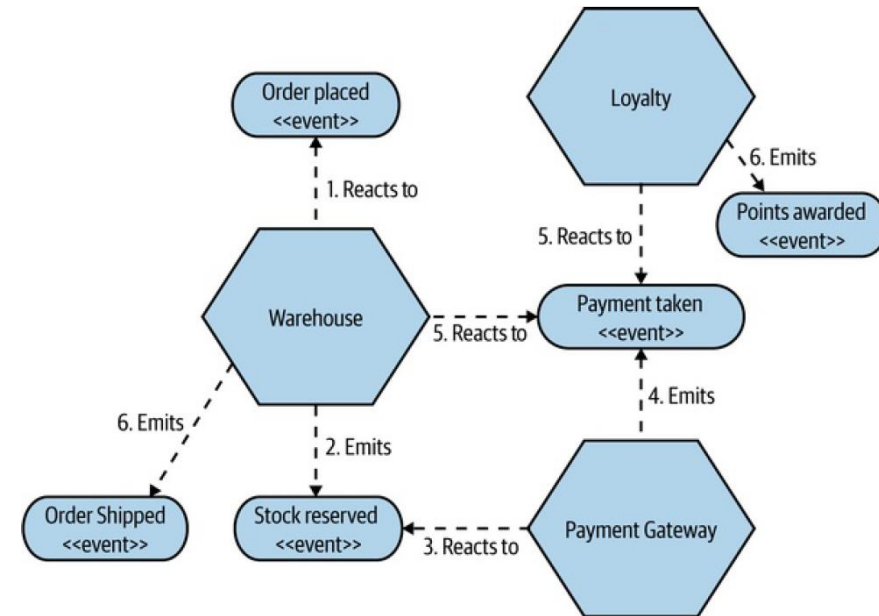
- Sættum okkur við að gögn verða **að lokum** í consistent state-l
- EDA hjálpa við eventual consistency (mætti mögulega telja sem kostur)

Testing

- Erfiðara að test-a event driven architecture
- Þarft að produce-a event til að test-a virkni í service sem á ekki event-ið

Minni verkflæðis stjórnun

- EDA leiðir oft til minni verkflæðis stjórnunar
- Oft ekkert explicit workflow
- Leiðir til minni innsýn og monitoring á verkflæðinu
- Ekki lengur point-to-point workflow tengingar
- Service bregðast við event-um og senda sín eigin
- Þetta á aðallega við um choreography pattern
- Hægt að fá meiri stjórn með orchestration pattern



Error handling

- Erfiðara að meðhöndla villur í EDA
- Producer fær ekkert svar til baka frá consumers
- Gætum þurft að senda annað event út til að tilkynna villu
- Stundum kemur villa sem leiðir til að ekki er hægt að senda út event
- Mest vandamál í choreography (en líka í orchestration)

Tracing & Monitoring

- Tracing og Monitoring erfiðara í EDA
- Erfiðara að tengja saman verkflæði
- Erfiðara að fylgjast með verkflæði

</The Cons>

Workflows

- Verkflæði er mengi af aðgerðum sem saman mynda business ferli
- Verkflæði krefst oft vinnu frá mörgum service-um
- Tvö helstu verkflæði mynstrin eru [choreography](#) og [orchestration](#)

Choreography

- Choreography á aðallega við um EDA
- Í choreography er **engin miðlæg verkflæðis stjórnun**
- Verkflæði logic er dreifð á milli service-a
- Service hlusta eftir og senda event sem önnur bregðast við
- Service vita ekkert um þau service sem þau senda á
- „dancers all finding their way and reacting to others around them in a ballet.“

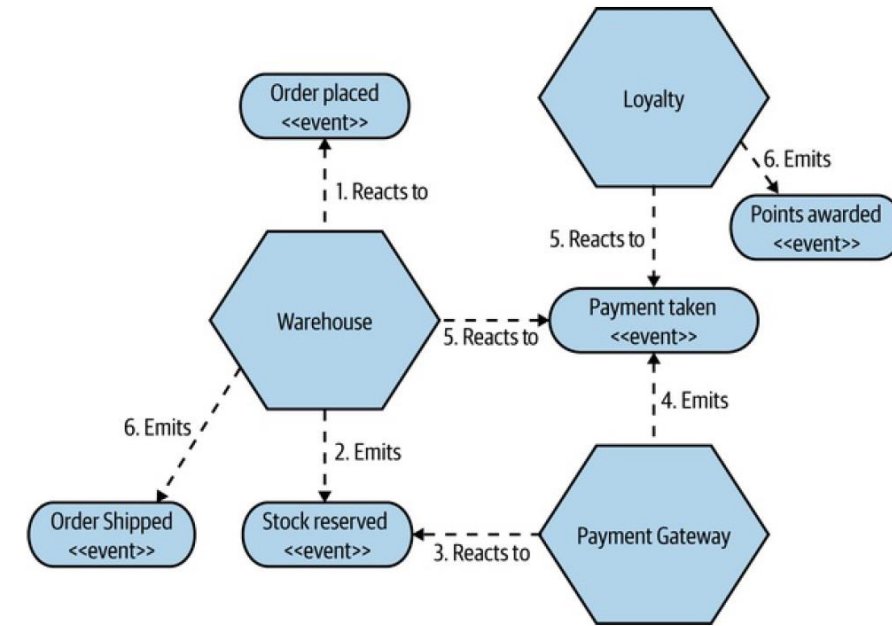


Figure 6-10. An example of a choreographed saga for implementing order fulfillment

Choreography frh.

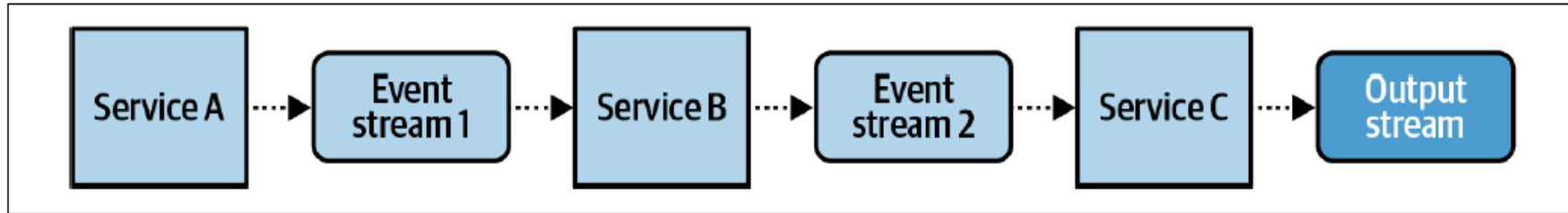


Figure 8-1. Simple event-driven choreographed workflow

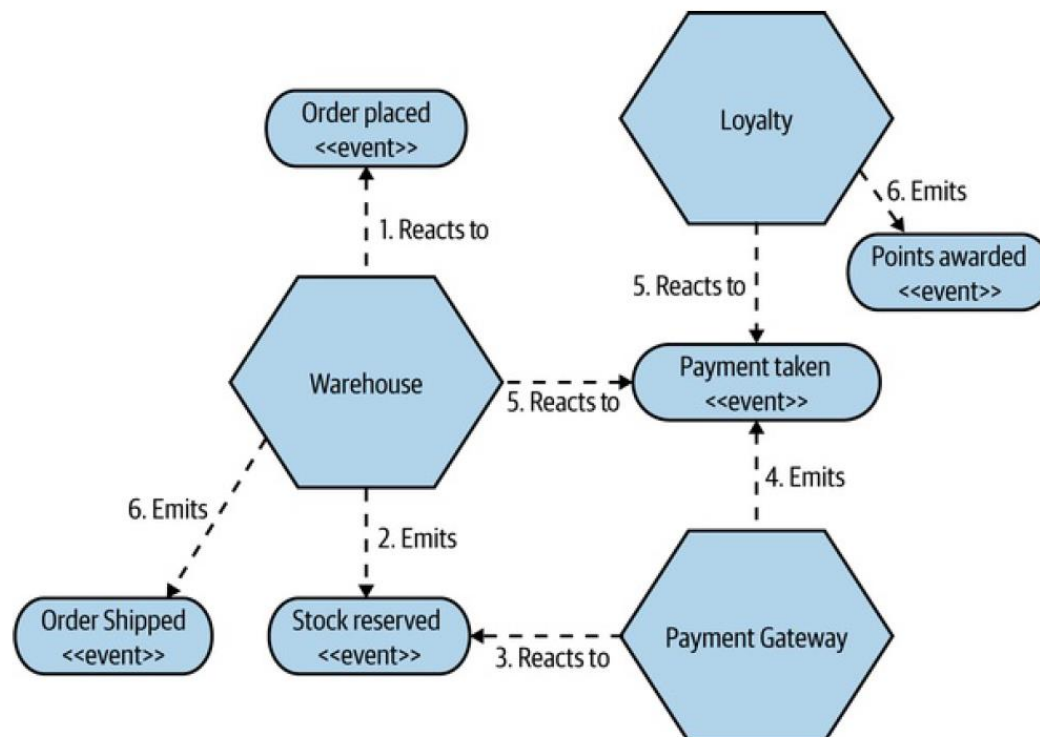


Figure 6-10. An example of a choreographed saga for implementing order fulfillment

Choreography frh.

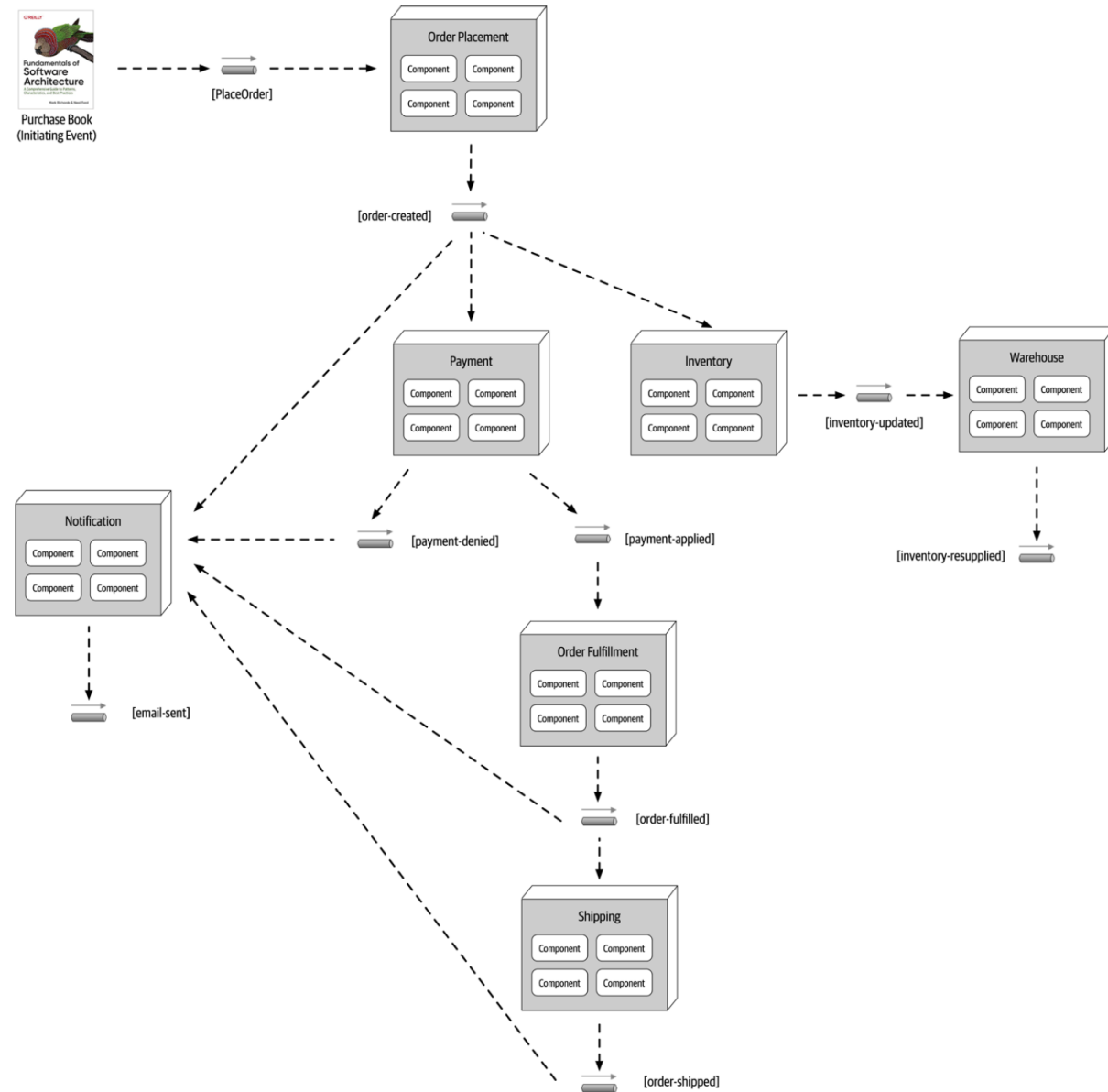
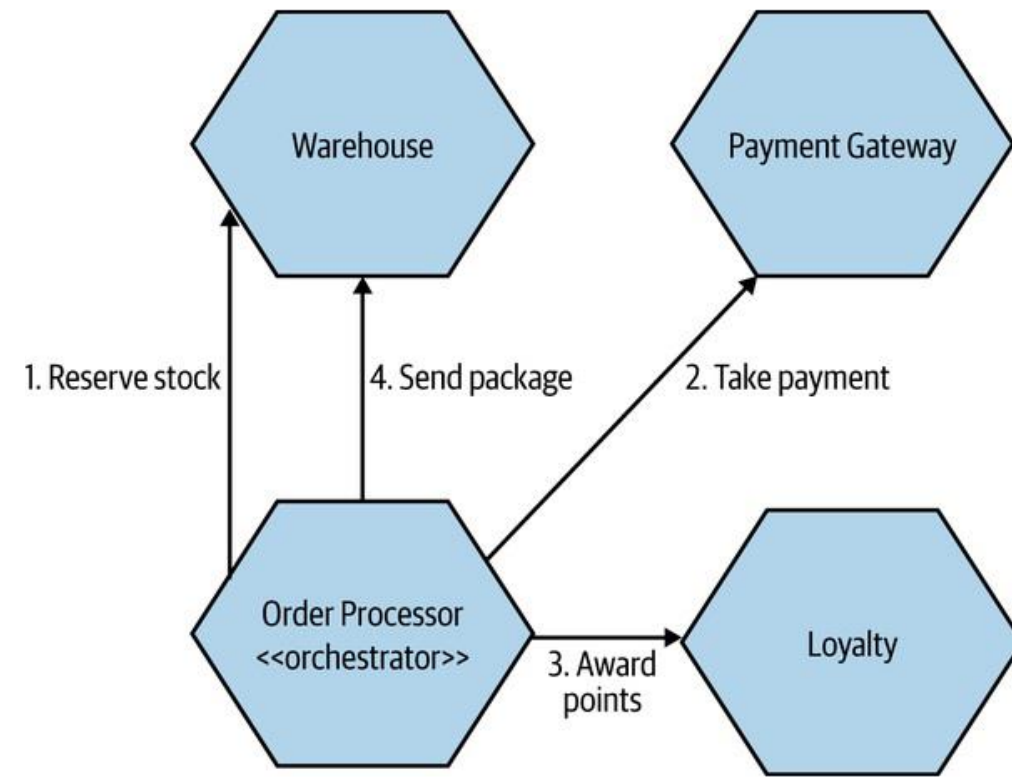


Figure 14-4. Example of the broker topology

Orchestration frh

- Á við um bæði EDA og request-response
- Miðlægur *orchestrator* sem sér um verkflæðið
- Allt verkflæðið skilgreint innan eins service
- Orchestrator ræður hvað gerist og hvenær
- Orchestrator bounded context-ið ætti eingöngu að vera verkflæðið sjálft
- “Like a single conductor commanding the musicians during a performance”



Orchestration

Getur bæði verið event driven og request-response

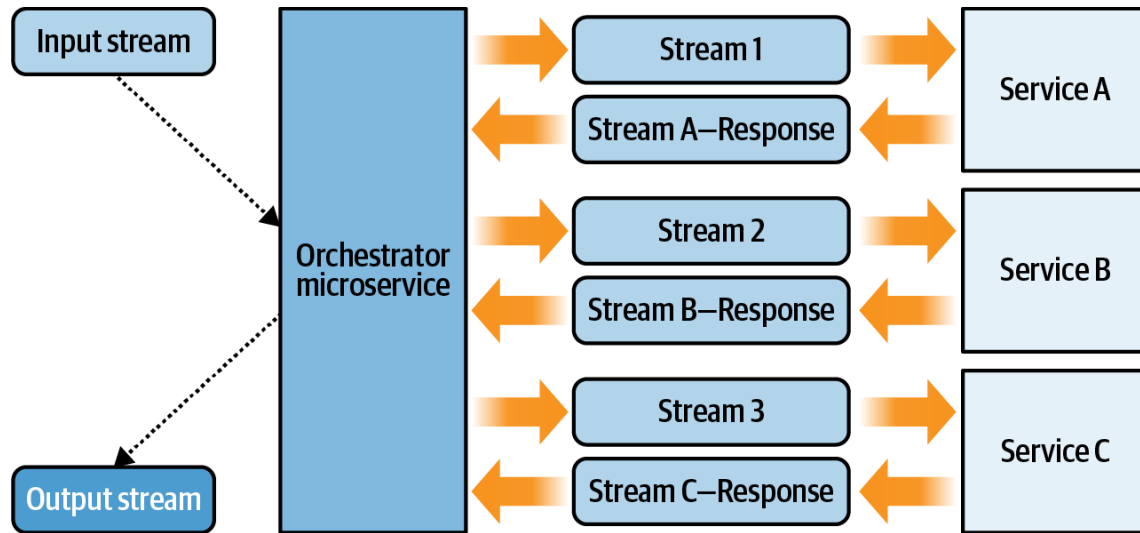
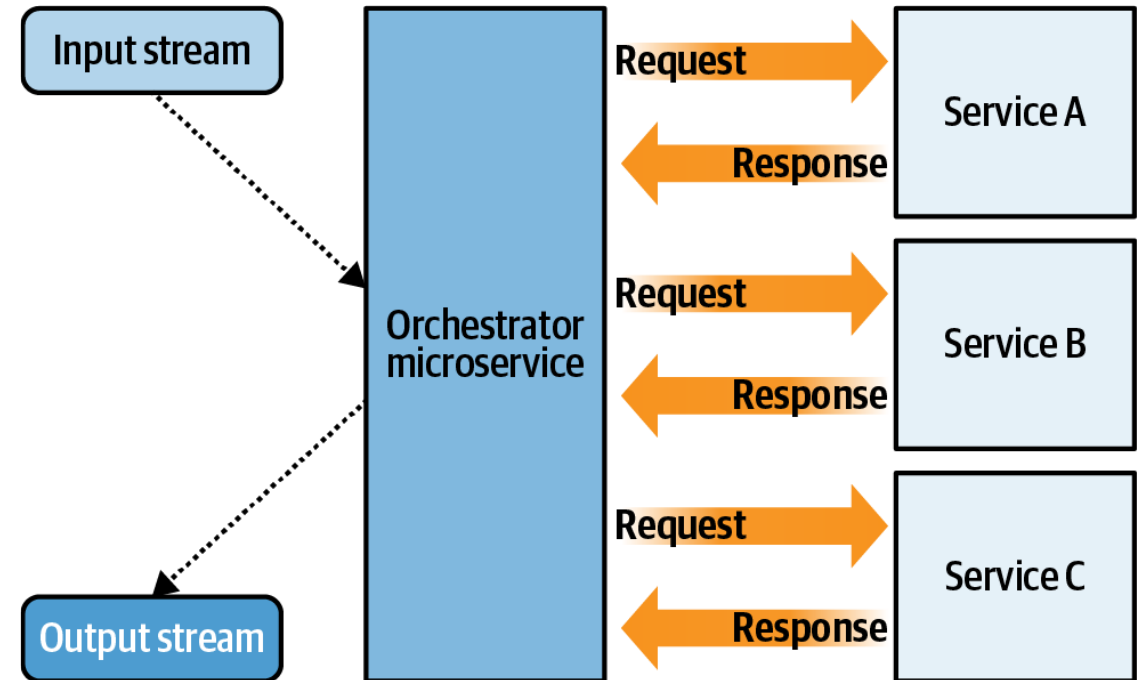


Table 8-1. Materialization of events issued from orchestration service

Input event ID	Service A	Service B	Service C	Status
100	<results>	<results>	<results>	Done
101	<results>	<results>	Dispatched	Processing
102	Dispatched	null	null	Processing



Orchestration dæmi

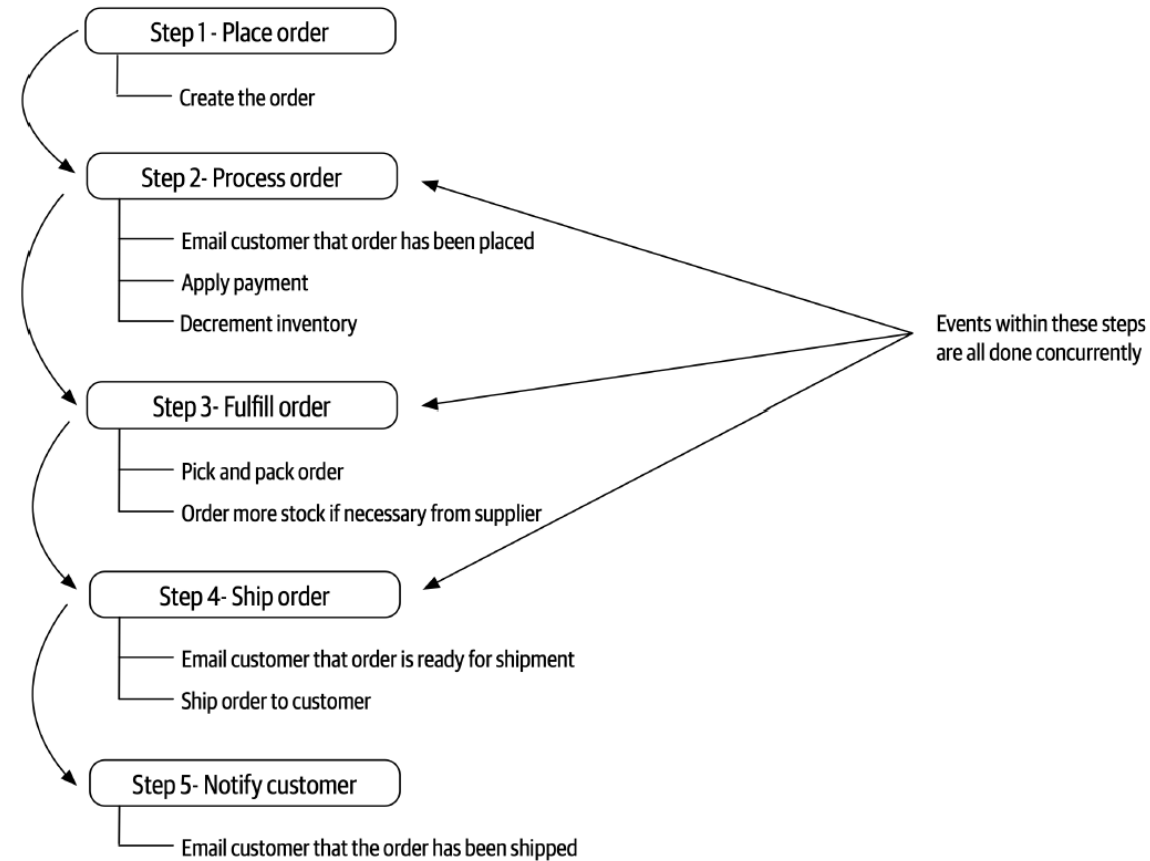


Figure 14-7. Mediator steps for placing an order

Orchestration dæmi frh.

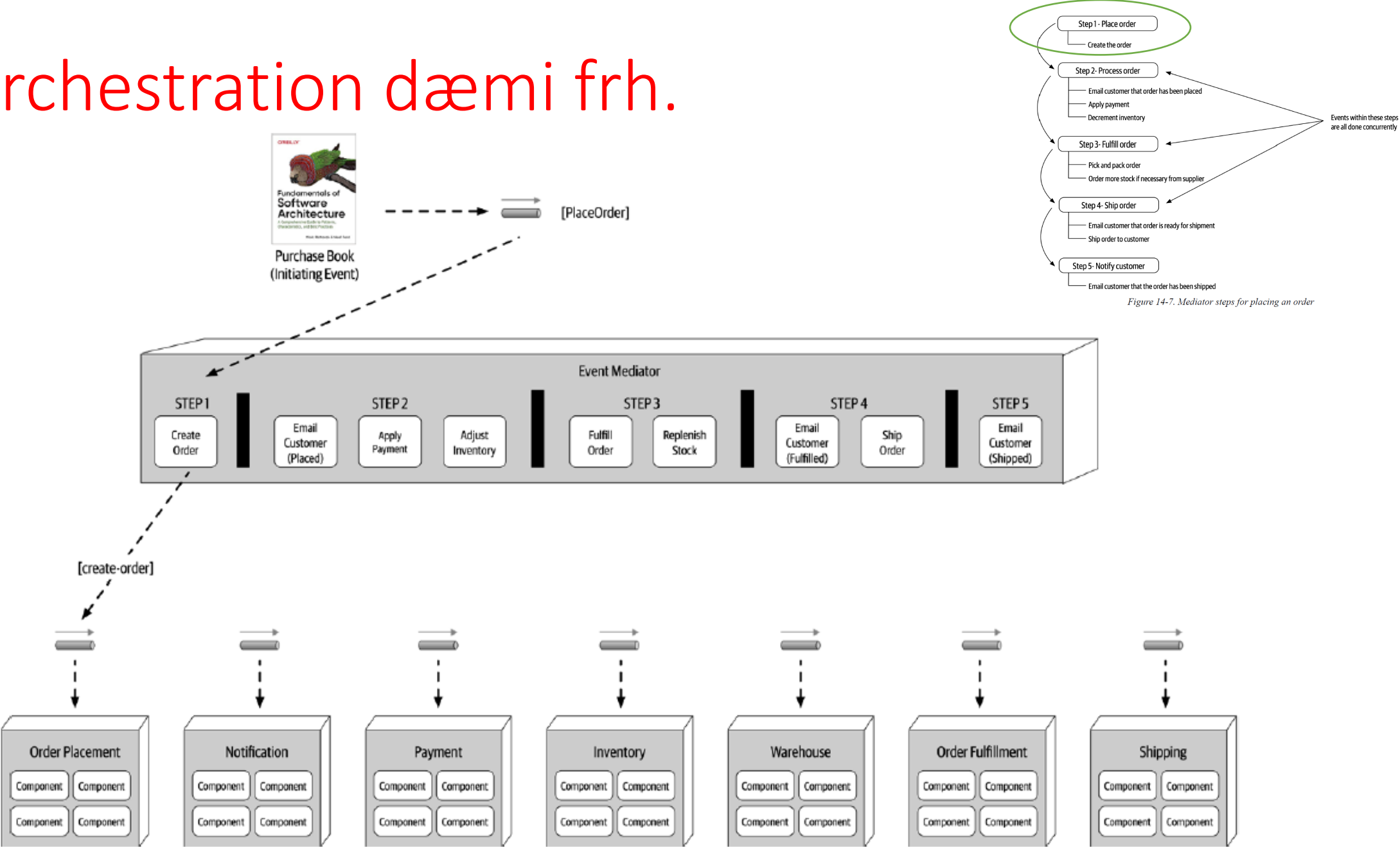


Figure 14-8. Step 1 of the mediator example

Orchestration dæmi frh.

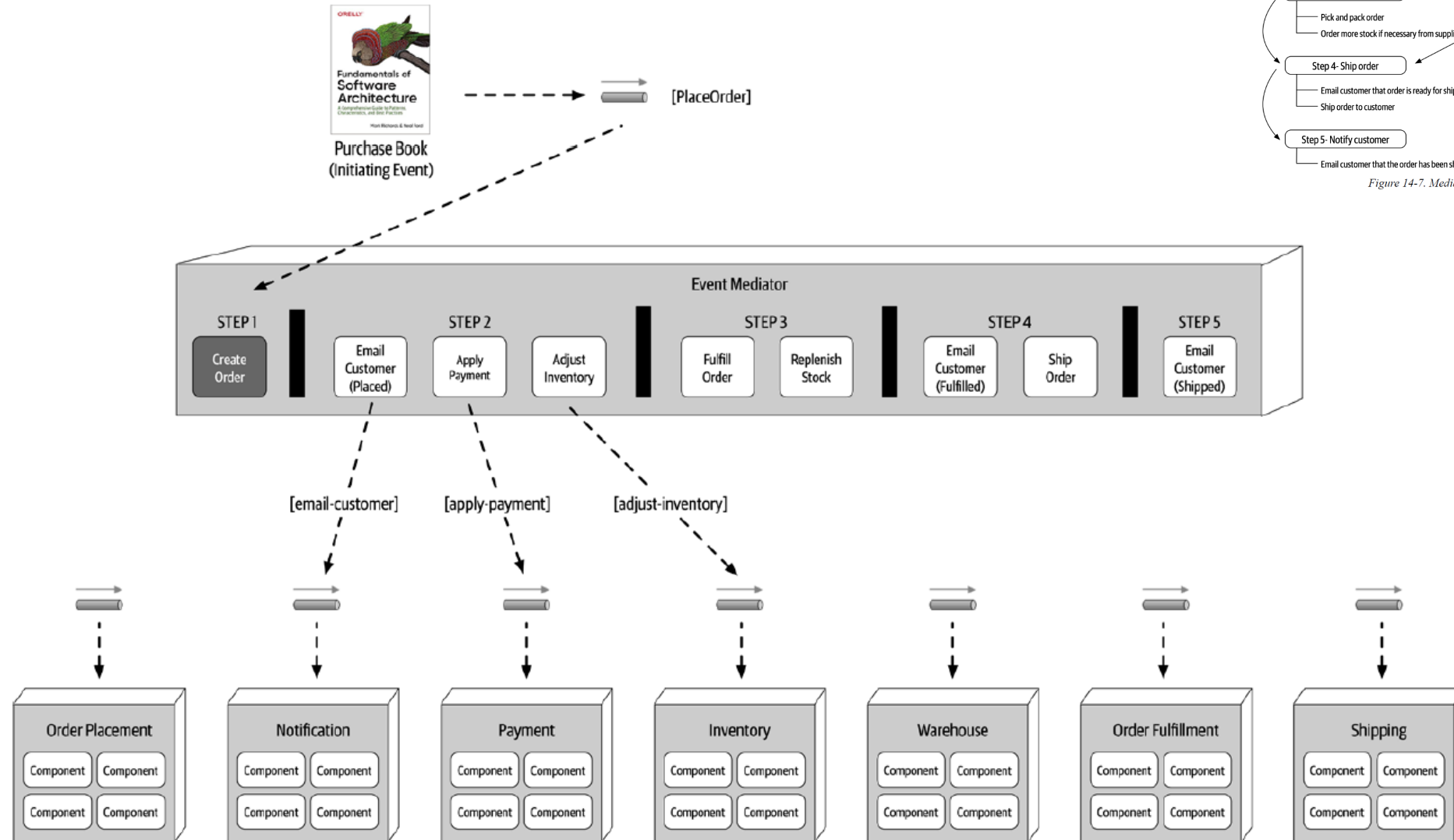


Figure 14-9. Step 2 of the mediator example

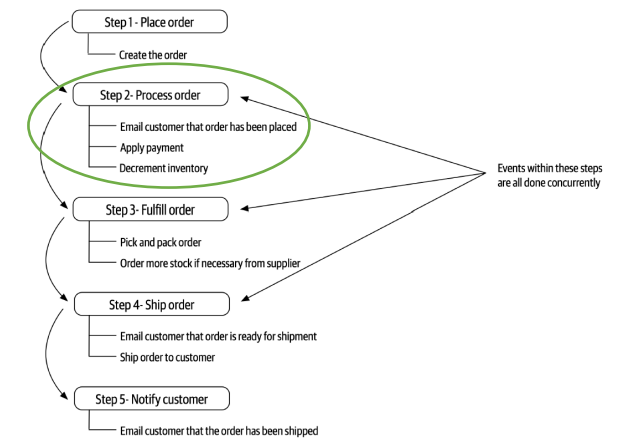


Figure 14-7. Mediator steps for placing an order

Orchestration dæmi frh.

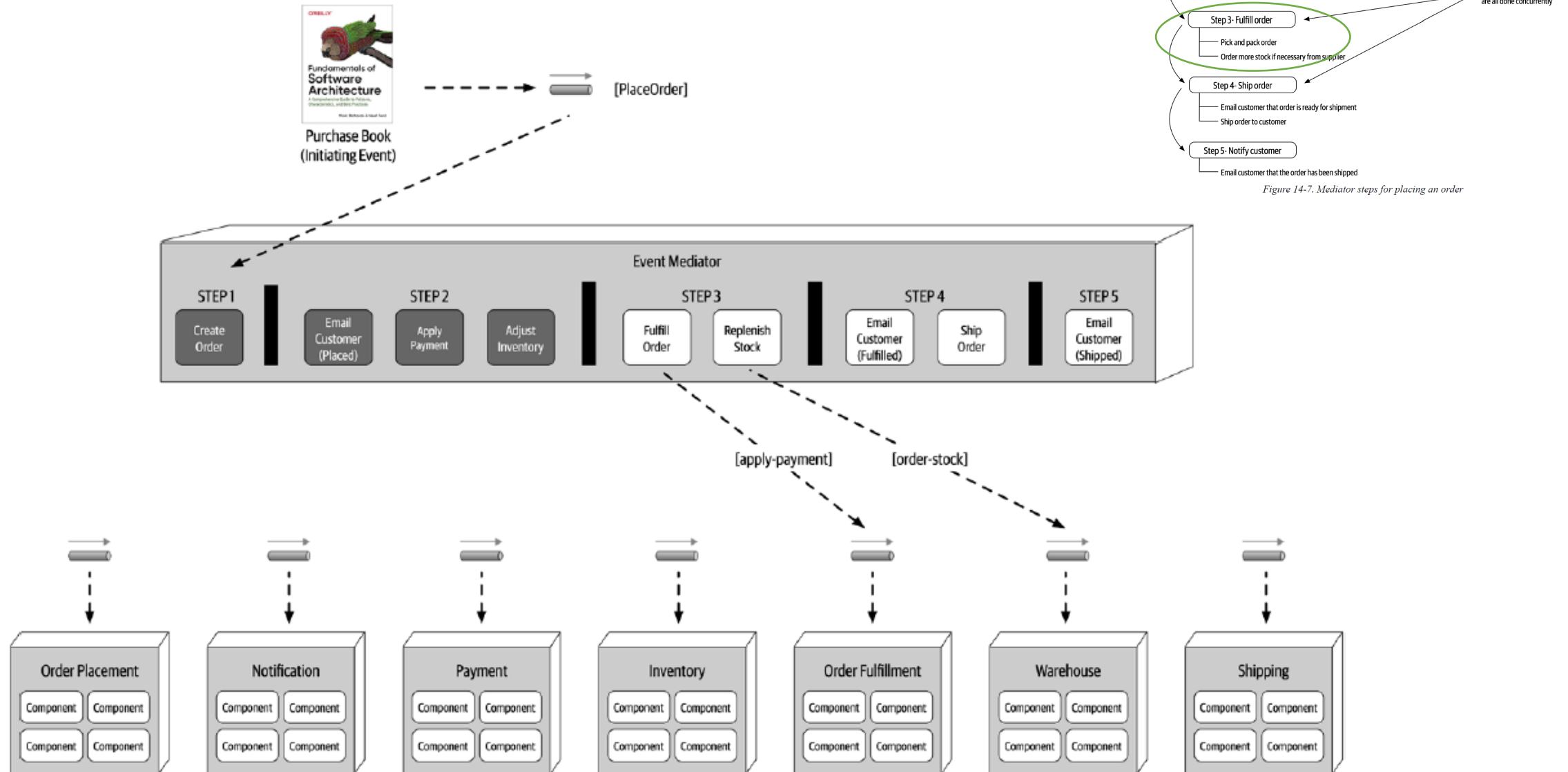


Figure 14-10. Step 3 of the mediator example

Orchestration dæmi frh.

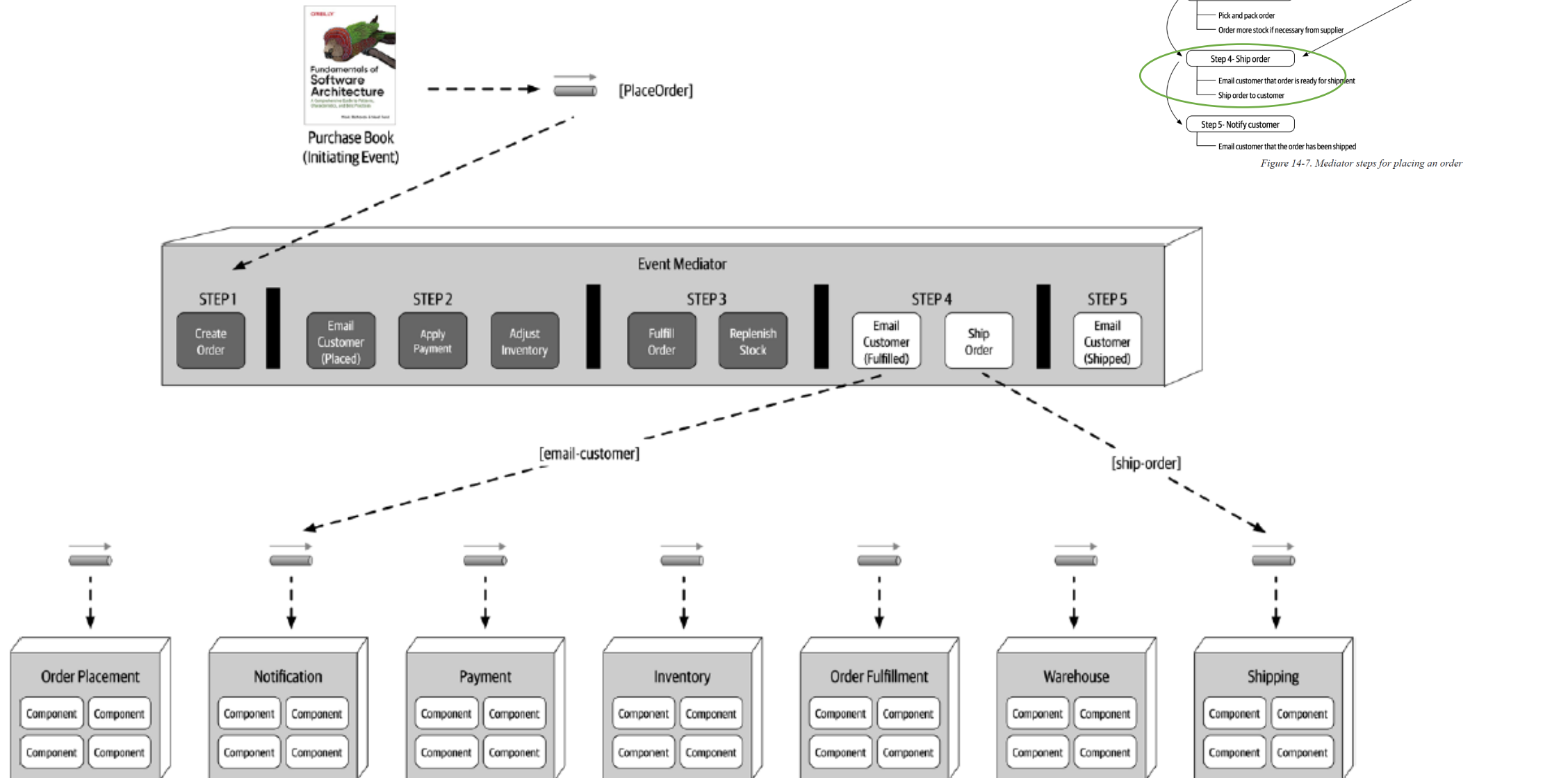


Figure 14-11. Step 4 of the mediator example

Orchestration dæmi frh.

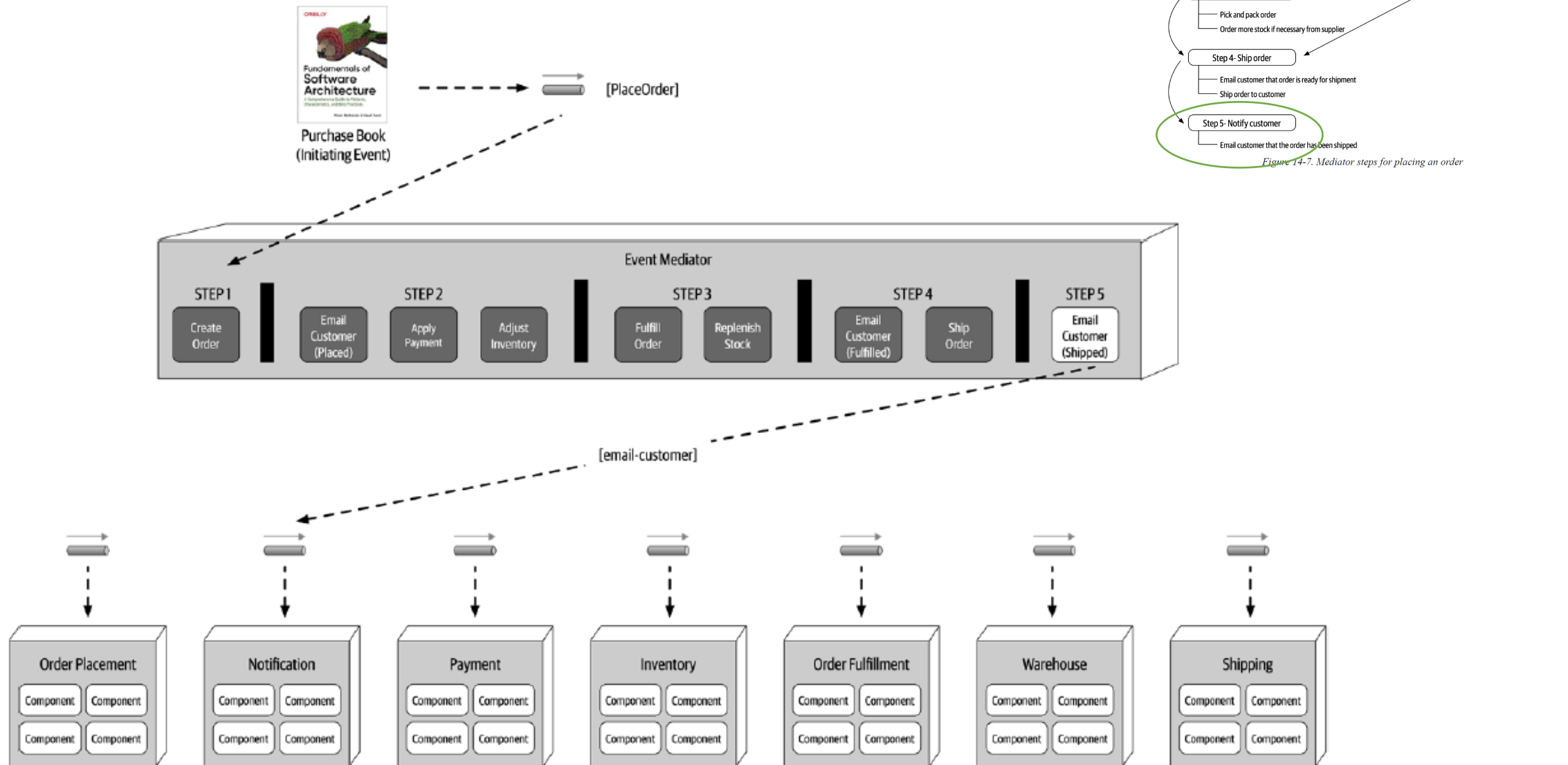


Figure 14-12. Step 5 of the mediator example

Choreography pros and cons

- Kostir
 - Oft einfaldara (nema fyrir flókin verkflæði)
 - Mun meira decoupled
 - Scalability, performance, throughput
 - Adaptability, flexibility
 - Góð dreifing á logic (ekkert god service)

Choreography pros and cons

Kostir

- Oft einfaldara
 - nema ef flókið verkflæði
- Mun meira decoupled
- Scalability, performance, throughput, responsiveness
- Adaptability, flexibility
- Góð dreifing á logic
 - Ekkert *god* service

Gallar

- Minni verkflæðis stjórnun
- Minni innsýn í verkflæðið
 - Verkflæði ekki explicit
 - Erfitt að vita í hvaða stöðu verkflæði er
 - Erfiðara að vita að villa kom upp í verkflæðinu
- Erfiðara að meðhöndla villur
- Verkflæðis breytingar erfiðari
- Erfiðara að sjá um distributed transactions

Orchestration pros and cons

Kostir

- Explicit skilgreining á verkflæði
- Auðveldar fyrir flókin verkflæði
- Auðveldar að breyta verkflæði
 - Allt verkflæði innan eins service
- Auðveldar að meðhöndla villu í verkflæðinu
- Auðvelt að sjá stöðu verkflæðis

Gallar

- Meiri coupling
- Hætta á of mikilli logic í orchestrator
 - *God service*
- Minna flexibility
- Single Point of Failure

Orchestrator eða Choreography?

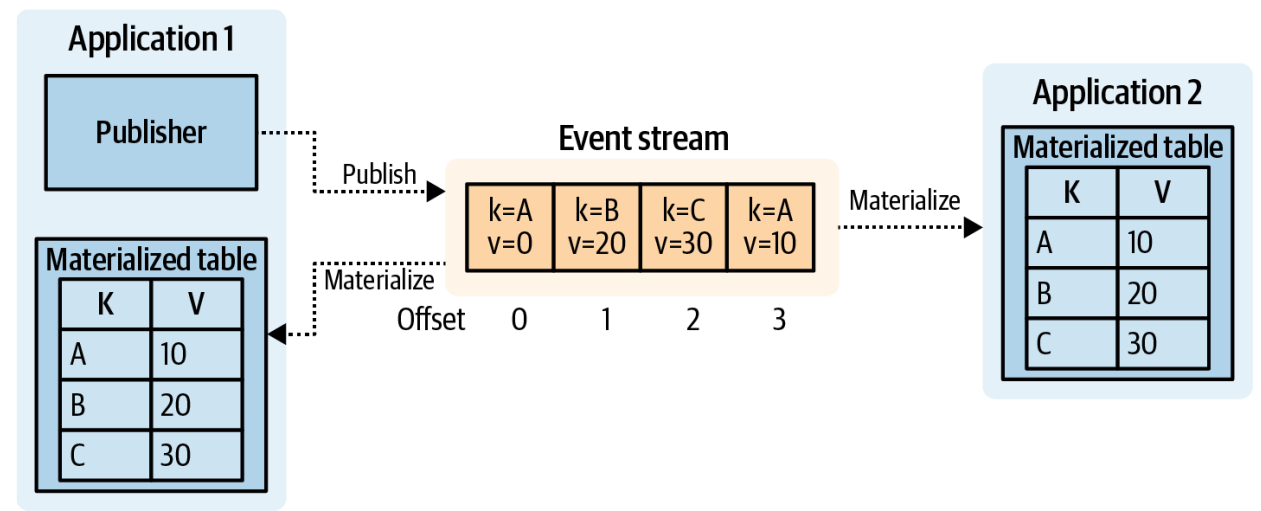
- Hægt að nota bæði í sama kerfi
- Notum það sem hentar best hverju sinni
- Choreography hentar betur þegar verkflæði teygist yfir teymi
 - Út af loose coupling
- Í EDA er þumalputtaregla oft að hugsa fyrst um choreography

The Contents and Structure of Events

- Yfirleitt **key / value** par
- **Headers**
 - Timestamp
 - Source id
 - Schema url
 - Triggering locic
 - **Trace id**
- Ættu að innihalda gögn sem þú **værir glaður að deila í gegnum API**

Event streams as the single source of truth

- Event ættu að vera óbreytanleg staðreynd
- Ættu að innihalda upplýsingar til að lýsa nákvæmlega hvað gerðist
- Gæta þarf að ekki myndast ágreiningur / ósamræmi á milli gagna
- Ef þú vilt gögn þá færðu það frá event-um
- Vista í grunn út frá event-um



Avoid Events as Semaphores or Signals

- Event ætti ekki eingöngu að vera merki um að eitthvað gerðist
- Event ætti að segja alla söguna um niðurstöðurnar og afleiðingann
- Viljum upplýsingar um ákveðna vinnu sem var unnin
- Ekki bar að vinna hafi verið unnin
- Ef þessu er ekki fylgt þarf consumer að leita að niðurstöðunum með request

Single Writer Principle

- Hvert event stream hefur eitt og aðeins eitt producing service
- Það service er eigandi straumsins, event-ana og schema
- Sama ástæða og af hverju service deila ekki gagnagrunnum
- Ef fleiri en einn þá myndast coupling, óskýr mörk og hætta á vilum

Use Schemas

- Event streymi / topic ættu að hafa schema
- Schema er explicit strúktúr á hvernig event má líta út
- Hvert streymi hefur eitt og aðeins eitt schema
- Ef þetta er ekki er auðvelt að publish-a event-i sem brýtur clients
- Kemur í veg fyrir backward incompatible breytingar
- Ekki allir broker-ar styðja Schemas (t.d. RabbitMQ)

Use a Singular Definition per Stream

- Ættum eingöngu að publish-a einni event tegund í straum
- Consumer ætti ekki að þurfa að filter-a eftir mismunandi tegundum
- T.d. ekki order created og buyer created event í sama topic
- Getum tryggt þetta með [Schemas](#)

Keep Events Single-Purpose

- Viljum ekki blanda tveimur event-um saman
- SRP
- Viljum ekki að client brotni út af gögnum sem hann þarf ekki
- Ekki gera

Keep Events Single-Purpose frh.

TypeEnum: Book, Movie
ActionEnum: Click, Bookmark

~~ProductEngagement {
 productId: Long,
 productType: TypeEnum,
 actionType: ActionEnum,
 //Only applies to productType=Movie
 watchedPreview: {null, Boolean},
 //Only applies to productType=Book,actionType=Bookmark
 pageId: {null, Int}
}~~

```
MovieClick {  
  movieId: Long,  
  watchedPreview: Boolean  
}
```

```
BookClick {  
  bookId: Long  
}  
  
BookBookmark {  
  bookId: Long,  
  pageId: Int  
}
```

Consumers Should be Idempotent

- Idempotent
 - consumer getur unnið sama event mörgum sinnum og staða kerfisins / niðurstaðan ætti að vera sú sama
- Minni hættu á duplicate events
- Auðvelt að setja offset til baka x mikið upp á error recovery
- Þetta guideline auðveldara sagt en gert
- Stuðlar að *effectively once processing*

Effectively Once Processing

- Líka kallað *Exactly Once Processing*
 - En það er statement sem er ekki hægt / erfitt að tryggja
- Producer ætti að styðja við Idempotent writes
 - Stutt af Kafka
- Producer ætti að synchronously skrifa á topic-ið
- Consumer ætti að styðja við Idempotent reads
 - Oft erfiðara sagt en gert
- Consumer ætti að reyna að commit-a offset-i ásamt að producer-a processing result sem part af transaction
 - Stutt af Kafka
- Ef við getum þetta ekki þurfum við að velja á milli *at most once* og *at least once* processing

Effectively Once Processing Compromise

At least once processing

```
def consume_loop(consumer, topics):
    try:
        consumer.subscribe(topics)

        msg_count = 0
        while running:
            msg = consumer.poll(timeout=1.0)
            if msg is None: continue

            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
                msg_process(msg)
                msg_count += 1
                if msg_count % MIN_COMMIT_COUNT == 0:
                    consumer.commit(asynchronous=False)

    finally:
        # Close down consumer to commit final offsets.
        consumer.close()
```

Látum við að við höfum unnið
event-ið eftir að við höfum unnið
það tryggir að við vinnum það
amk einu sinni.

Effectively Once Processing Compromise

At most once processing

```
def consume_loop(consumer, topics):
    try:
        consumer.subscribe(topics)

        while running:
            msg = consumer.poll(timeout=1.0)
            if msg is None: continue

            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
                consumer.commit(asynchronous=False)
                msg_process(msg)

    finally:
        # Close down consumer to commit final offsets.
        consumer.close()
```

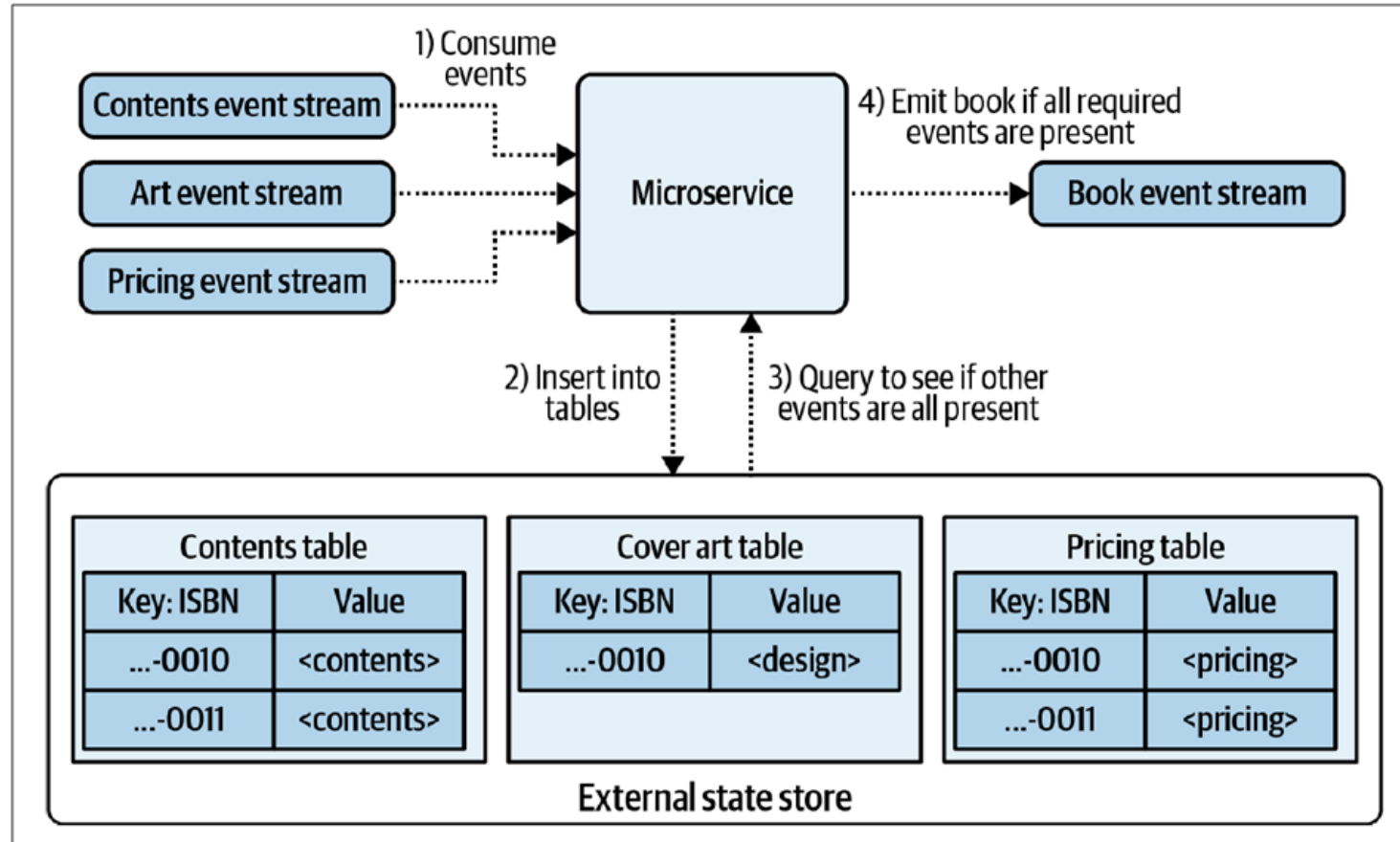
Látum við að við höfum unnið event-ið áður en við höfum unnið það tryggir að við vinnum það í mesta lagi einu sinni.

Some Patterns

Gating Pattern

Býður þangað til öll event eru komin áður en event er produced

Agent í orchestrator



Dead Letter Queues

- Einnig kallað Message Hospital
- Getum sent event í Dead Letter Queue ef vinnsla á því fail-ar
- Mögulega hægt að laga event-in og setja þau aftur í straumin
- Mögulega þarf að finna út hvað fór úrskeiðis

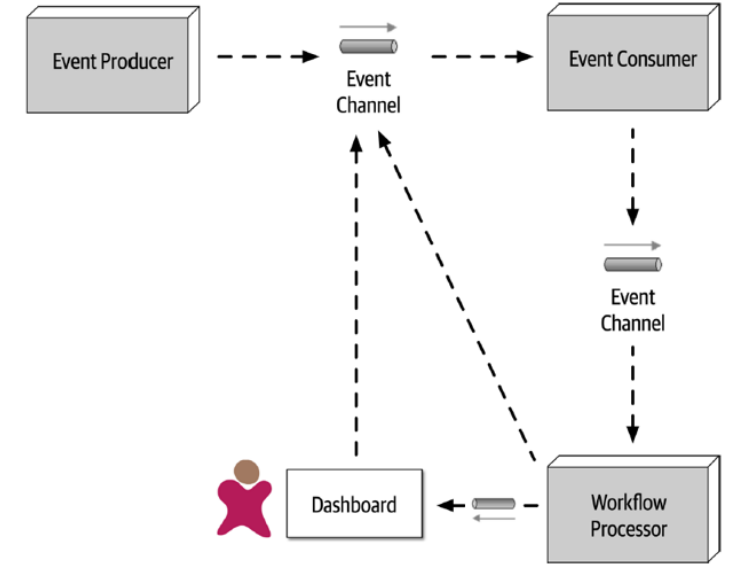


Figure 14-14. Workflow event pattern of reactive architecture

Dead Letter Queues frh

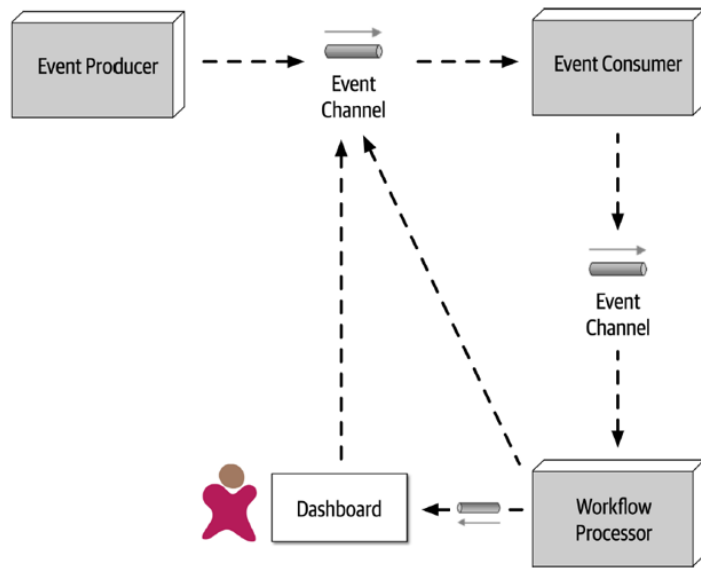
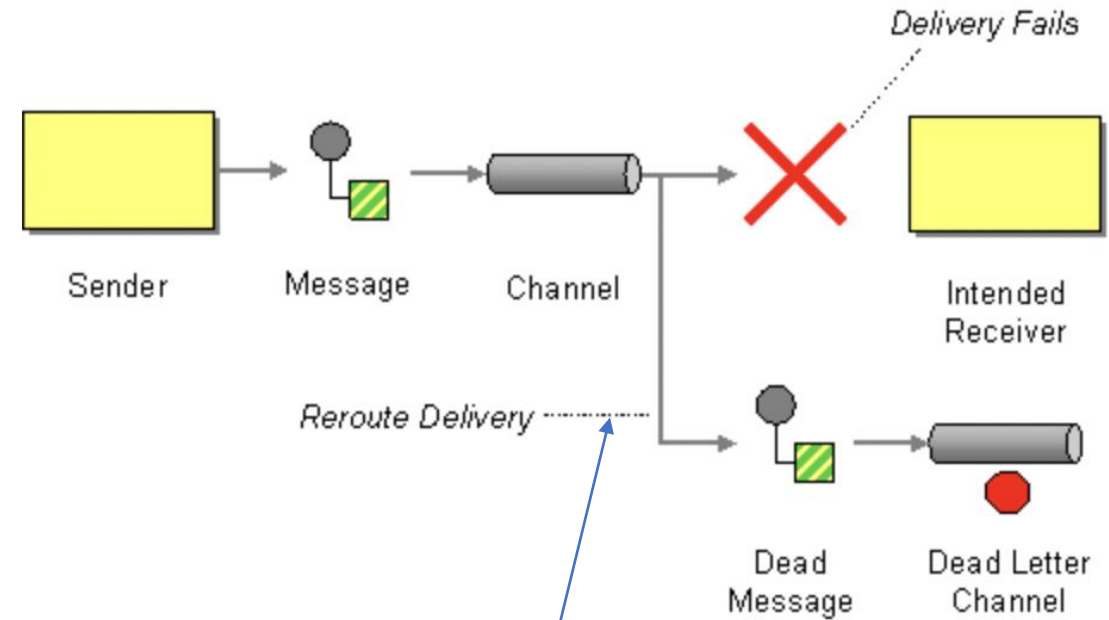


Figure 14-14. Workflow event pattern of reactive architecture

Getum sjálf útfært dead letter queue ef vinnsla fail-ar



Brokers geta oft verið stilltir til að höndla svona hluti fyrir okkur

Request-Reply í Event Driven Architecture

Hægt að gera Request-Reply í EDA með svokölluðu *pseudosynchronous communications*

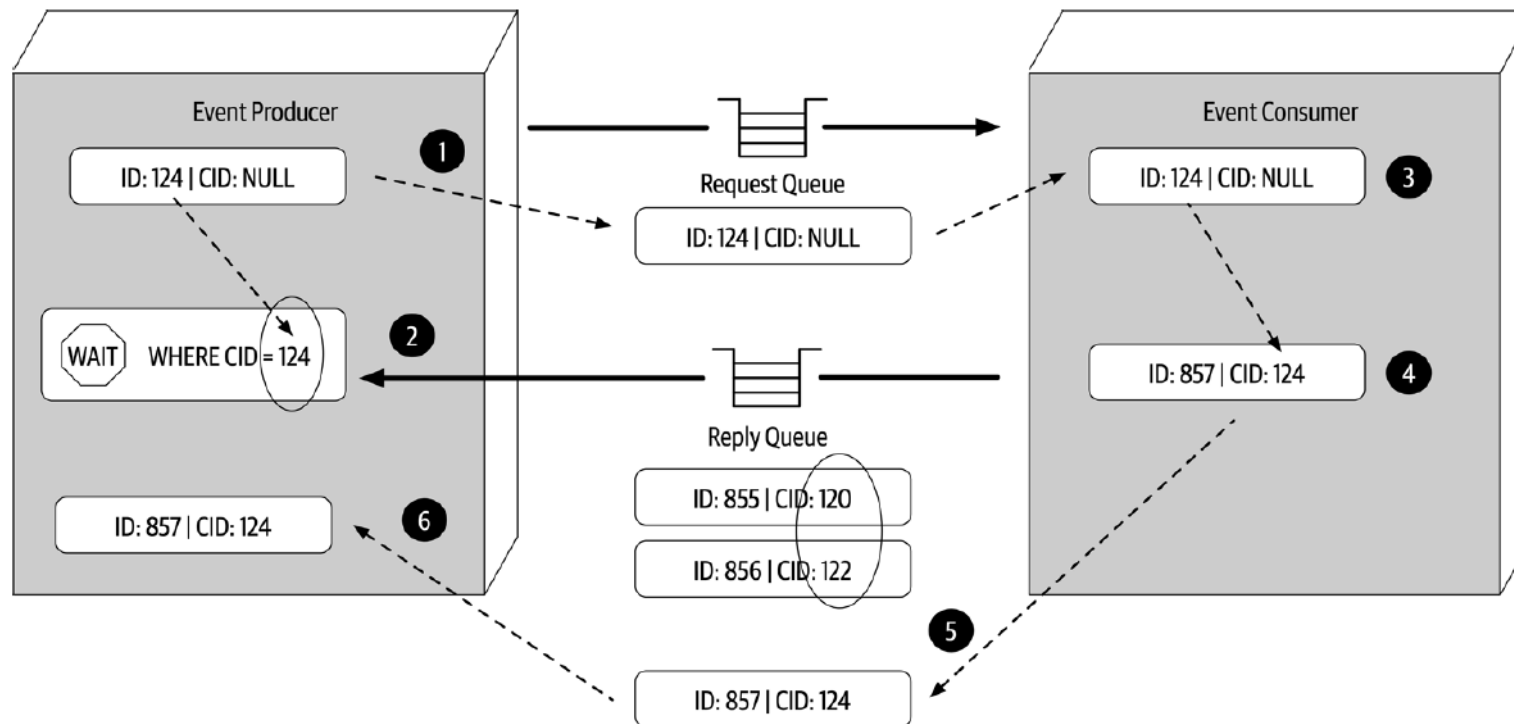


Figure 14-20. Request-reply message processing using a correlation ID

Sagas

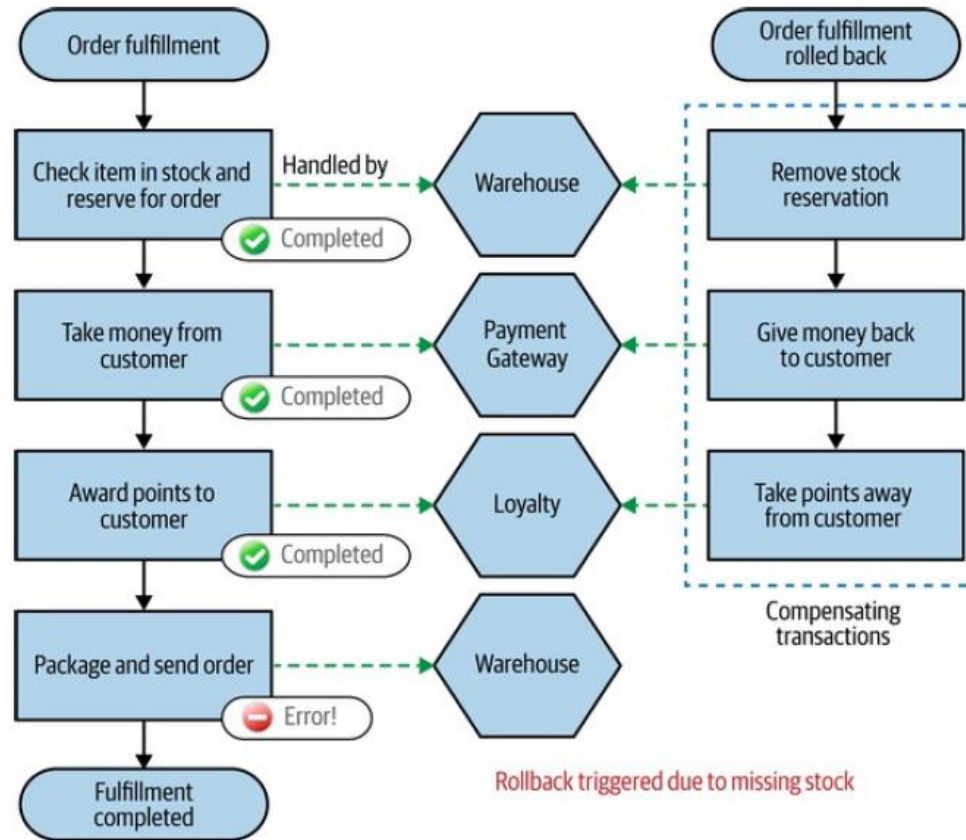
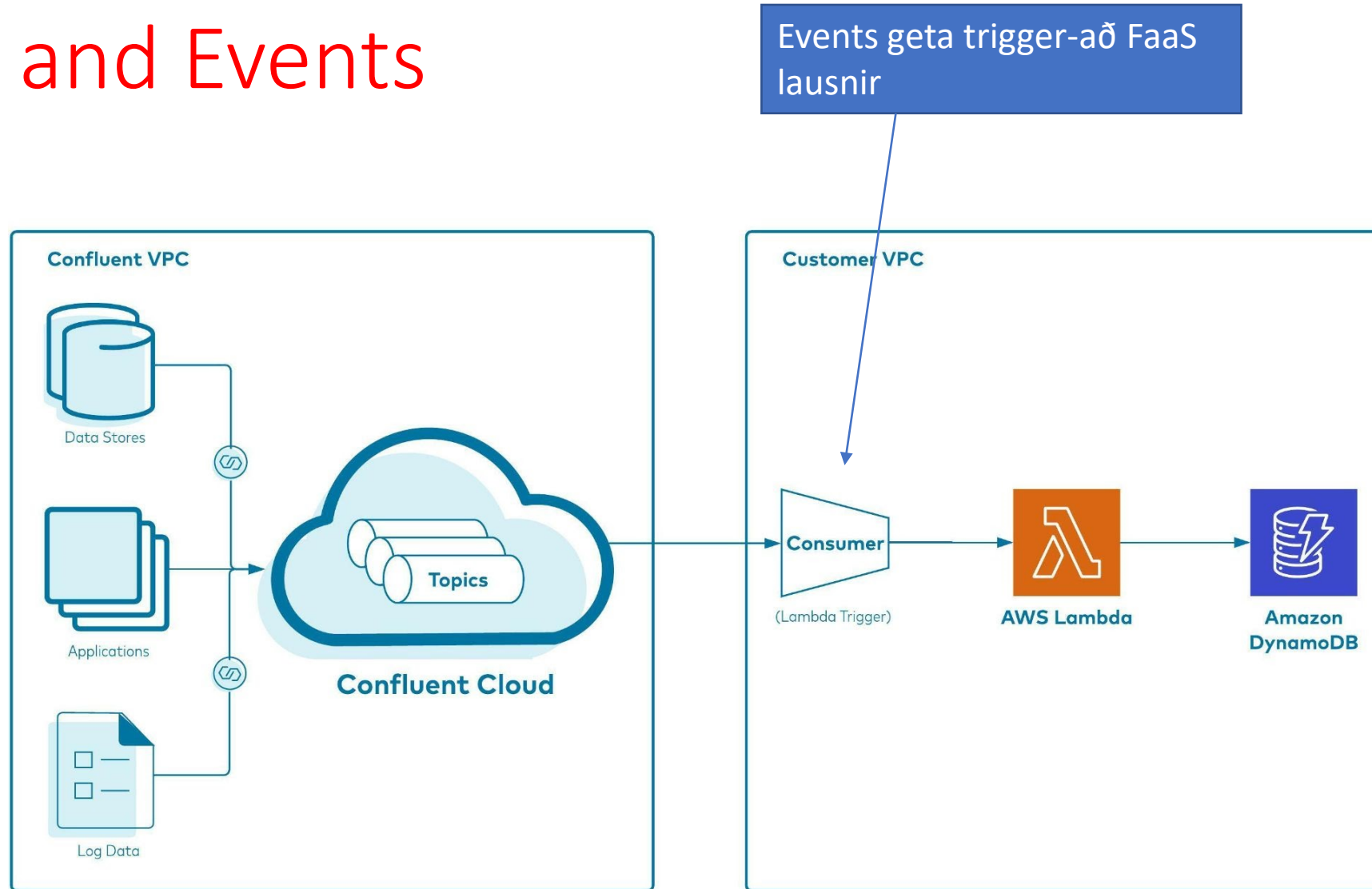


Figure 6-7. Triggering a rollback of the entire saga

FaaS and Events



Hybrid Architecture

- Sjaldan sem kerfi er eingöngu event driven
- Oft sem request-response samskipti henta betur
 - Tracing þarfir
 - Auðkenningu
 - Etc.
- Notum það sem hentar best hverju sinni