ProSiebenSat.1 Media SE

REFERENCE ARCHITECTURE

MESSAGE-BASED DATA REPLICATION

Implementation of Martin Fowler's ECST Pattern (Event-Carried State Transfer)

White paper: https://medium.com/@max.zalota/event-carried-state-transfer-reference-architecture-26ef49186c44

By Maxim Zalota, Principal Engineer

ProSiebenSat.1 Tech & Services

ABOUT ME

- Industries:
  - Media/Broadcasting – ProSiebenSat1: Munich, Germany
  - Insurance – Travelers: Hartford, CT, USA
  - Start-up founder – SalonsOn.Net: CT, USA
  - IT Consulting – Dassault Systems: MA, USA

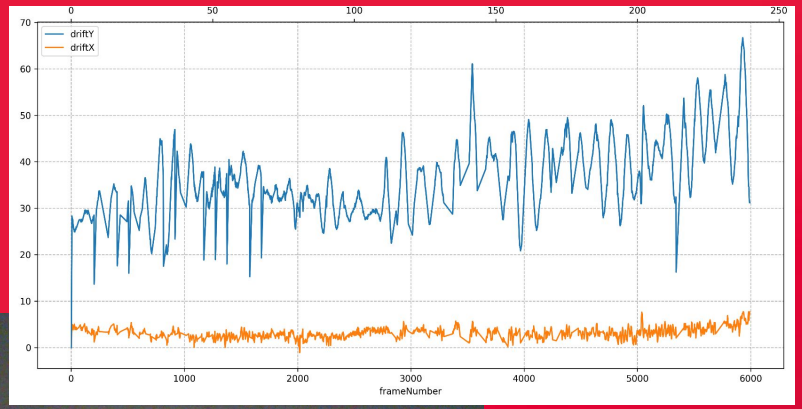
- Roles:
  - Software Engineer
  - Solutions Architect
  - Engineering Manager

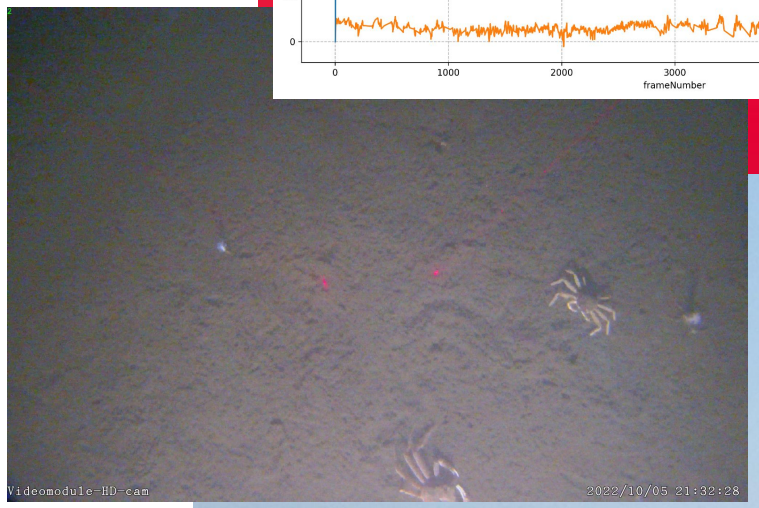
- Education:
  - Computer Science - Bachelors: University of Michigan, USA
  - Signal Processing - Masters: University of Michigan, USA
  - Business Administration - Masters: Oxford University, UK

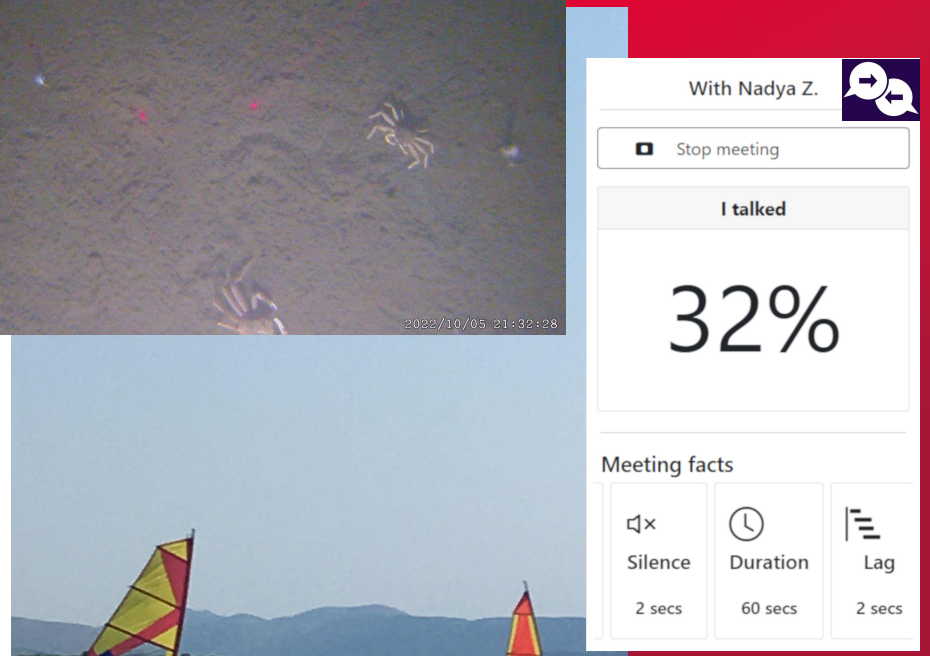
- Side projects:
  - WhoTalks.app - make every one-on-one meeting more engaging and productive.
  - Scientific analysis of ocean floor videos - constructing geometry and motion.
  - Refactoring Katas - https://github.com/mzalota/katas

- Hobbies: Travelling, Windsurfing

- Interests: Space exploration, Physics, Macroeconomics, Green energy
  https://www.goodreads.com/user/show/50303934-max-zalota





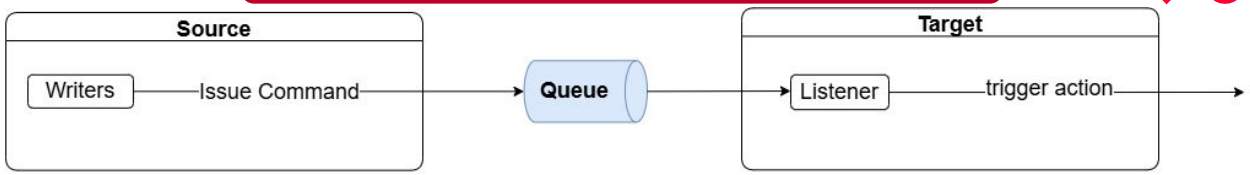


MESSAGING PATTERNS - OVERVIEW

Listeners act on a single message - immediately

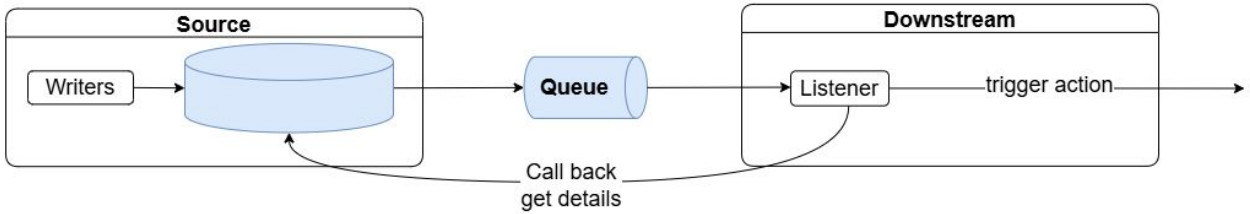
Command

Message-based remote call (fire-and-forget)



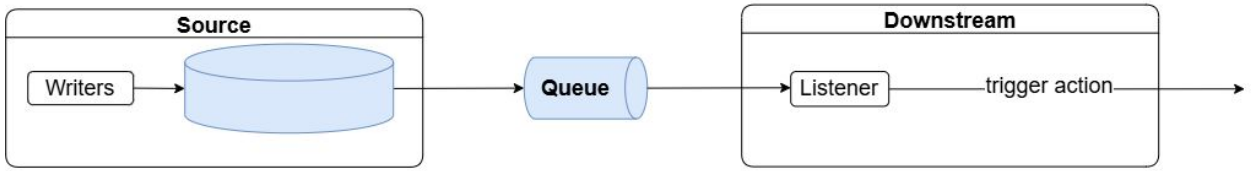
Event notification with callback

Message with minimal payload (id+action)



Event notification with full payload

Message with full payload

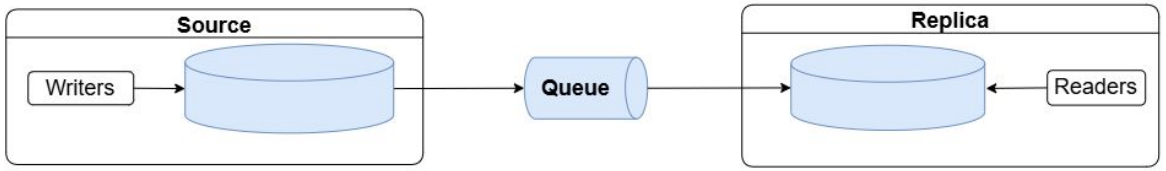


Readers act on multiple messages - later

Event-Carried State Transfer (ECST)

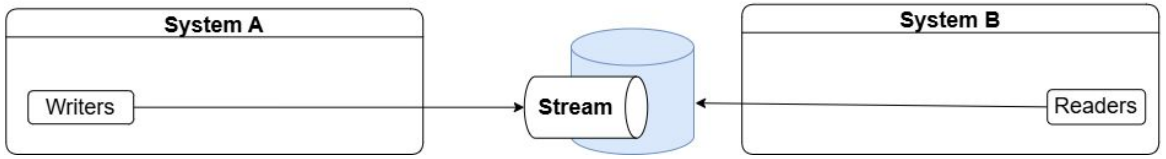
Data replication

Message-based
data replication



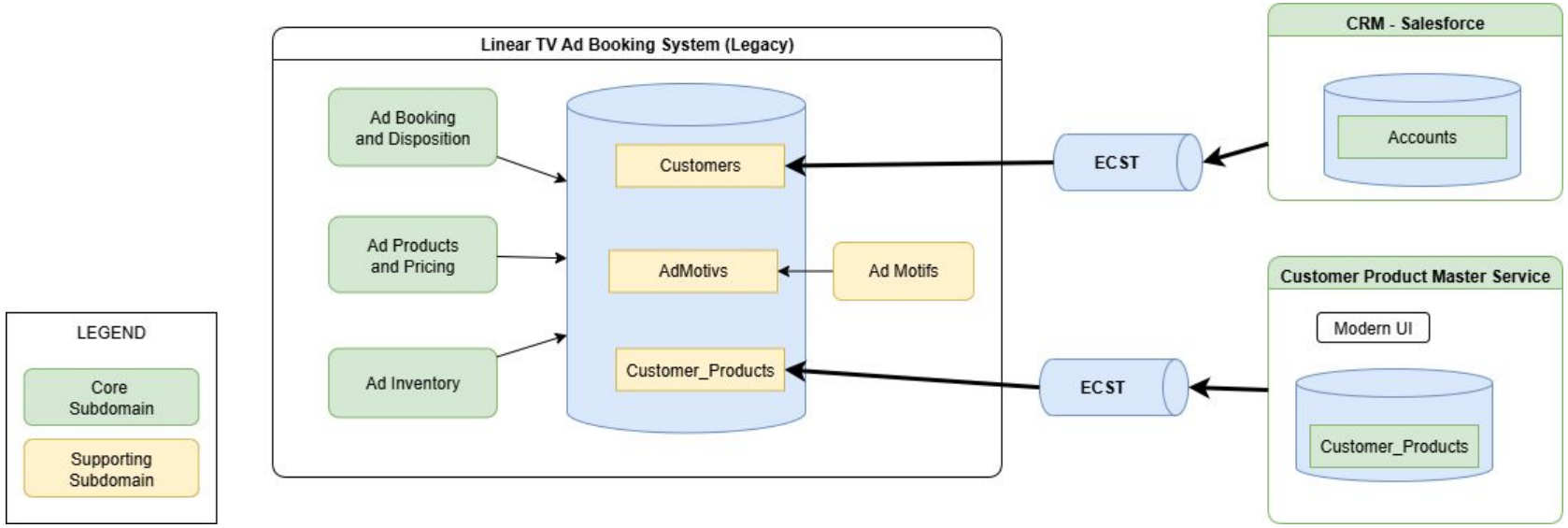
Event Sourcing

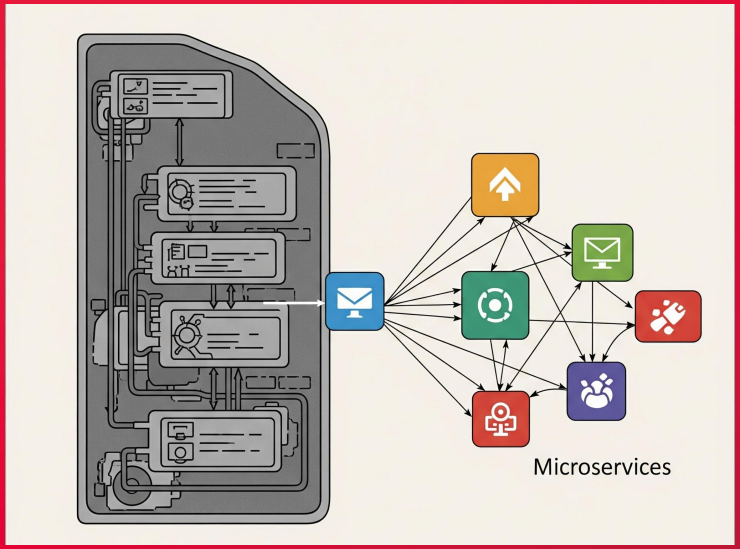
"Inside-out" database (immutable change log)

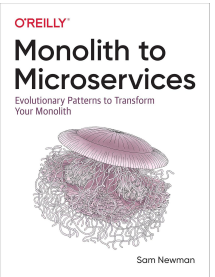


OUR USE-CASE

- Refactoring a Monolith into Microservices
  - New microservice becomes the Source - processes Reads and Writes.
  - Monolith keeps the Replica - keeps existing table as read-only. It's too expensive to rewrite all SQL Joins.





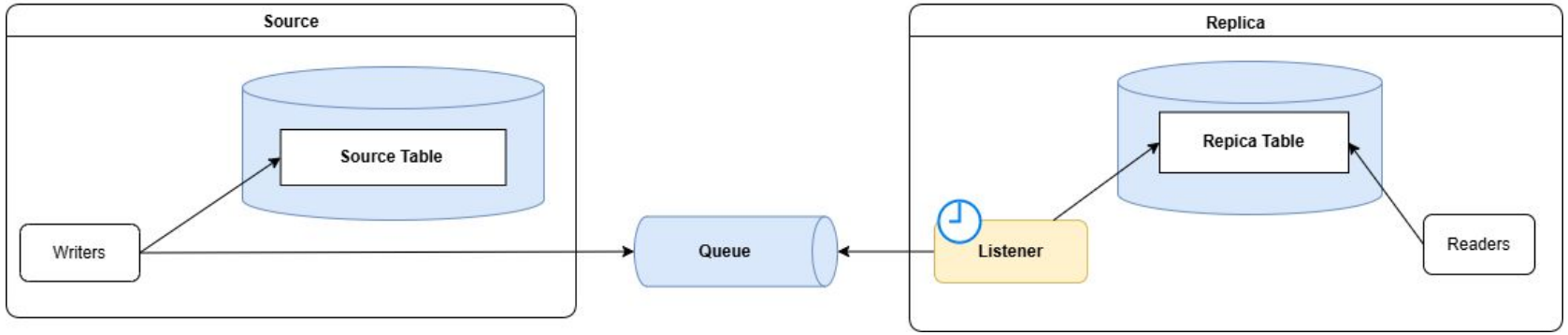


References in the „Monolith-to-Microservices" book:

- Pattern: Change Data Capture (page 120)
- Pattern: Tracer Write (page 149)

Make sure your use case is worth the complexity

QUICK IMPLEMENTATION IS NOT RESILIENT



1. Writers cannot simultaneously commit to the DB and publish to the Queue:

Implementation Option A

```
10 var record = new Record(userInputs)
20 database.insert(record)
30 queue.send(record)
40 database.commit()
```

Event is sent, but record is not saved to DB.

Crash

Implementation Option B

```
10 var record = new Record(userInputs)
20 database.insert(record)
30 database.commit()
40 queue.send(record)
```

Record is saved to DB, but event is not sent

2. Parallel Listeners may encounter race conditions leading to data corruption, for example a Delete message is processed before Create message

Failure modes:

1. Crashes and network failures

2. Race conditions

Not resilient enough for valuable data

Architectural Requirements

1. RESILIENCE AGAINST INFRASTRUCTURE FAILURES

- Kubernetes can restart container any time

- Examples of technical issues:

  1. Kubernetes can restart container any time
  2. Server, operating system, container or process crashes.
  3. Network interruptions: Request is lost or Response is lost.
  4. Database exceptions, such as deadlocks or cursor failure.
  5. Etc

- Operations Engineers (SREs) should be able to restart any component without the fear of causing a permanent data inconsistency.

Anything that can go wrong, will go wrong (Murphy's law)

2. RESILIENCE AGAINST RACE CONDITIONS

Two messages with the same Entity_ID could arrive in the wrong order:

→ “updated_on” timestamp at the milliseconds granularity.

Scenario 1: Two Updates out-of-order

→ Payload of the Update messages cannot be “delta”.

Scenario 2: Update before Create

→ Treat every Update message as an Upsert (Update or Insert).

Scenario 3: Delete before Update/Create

→ Persist Enity_IDs of deleted messages at Replica

Caused by parallelized Publishers and Listeners

3. RESILIENCE AGAINST “POISON PILL” MESSAGES

“Poison Pill” is a record that fails upon consumption, no matter the number of attempts.

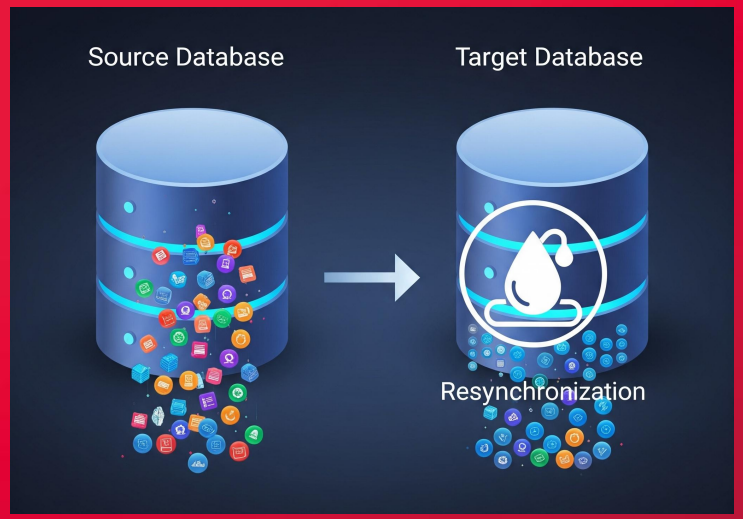
- For example:
  - A column in the Replica table is defined as a small integer, but a message contains a large number.
  - A column in the Replica table is defined as integer, but message contains an empty space string that the Listener fails to interpret as null.

- A failure will cause an infinite retry – no other messages can be processed

- Resolution will probably take the long time. Workarounds will be risky.

No quick fix or a safe workaround

4. ROBUST DATA RE-SYNCHRONIZATION



- Examples, where you will end up needing a “re-sync” capability:
  1. Initial rollout of the solution —Replica table is empty and needs to be filled out.
  2. Additional Listener+Replica is added
  3. A “one-shot” data change took place on the Source table bypassing outbox table. These changes now need to be propagated to the Replica table.
  4. A bug was introduced with a new release that compromised solution presented in this article, for example: the Listener swallows certain Exceptions
  5. Source table was restored from Production to Test environment.

- Re-synch must run reliably in parallel with normal messages

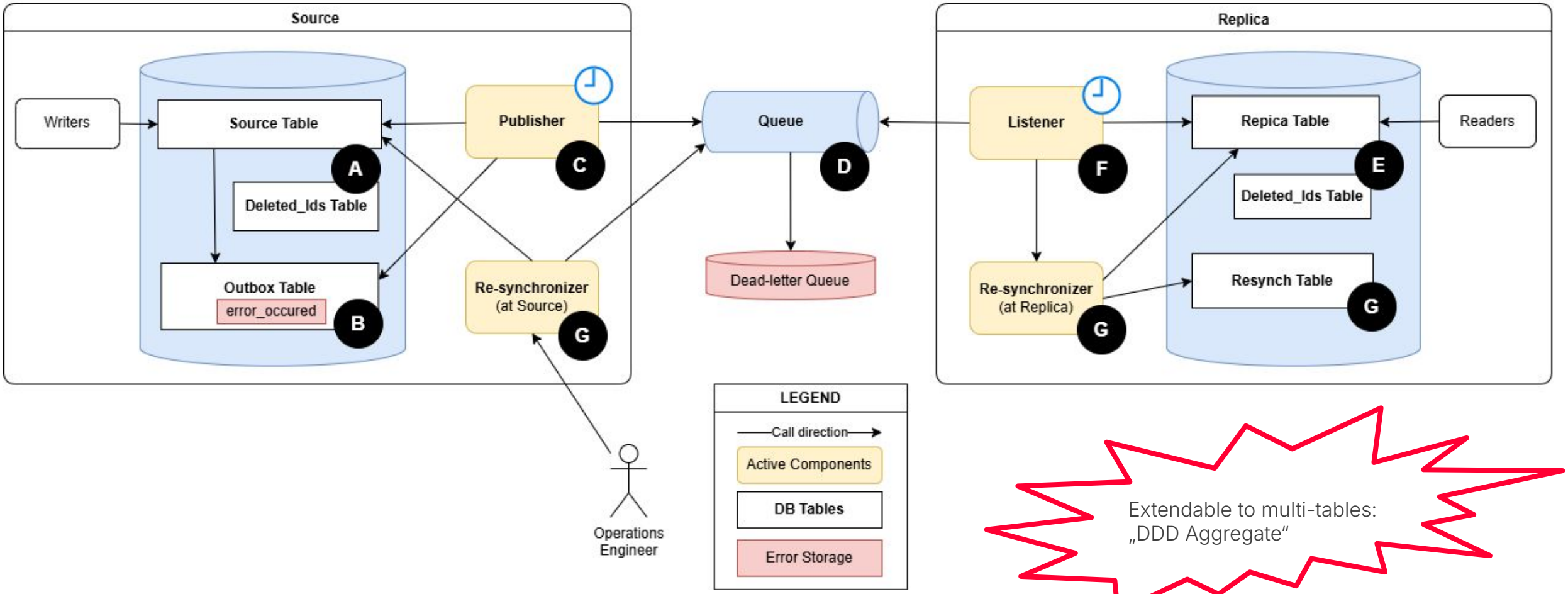
1) Press a button
2) Wait a bit

Data is consistent!

Solution
(Design Template)

Consider using **Debezium** CDC:
off-the-shelf and open-source

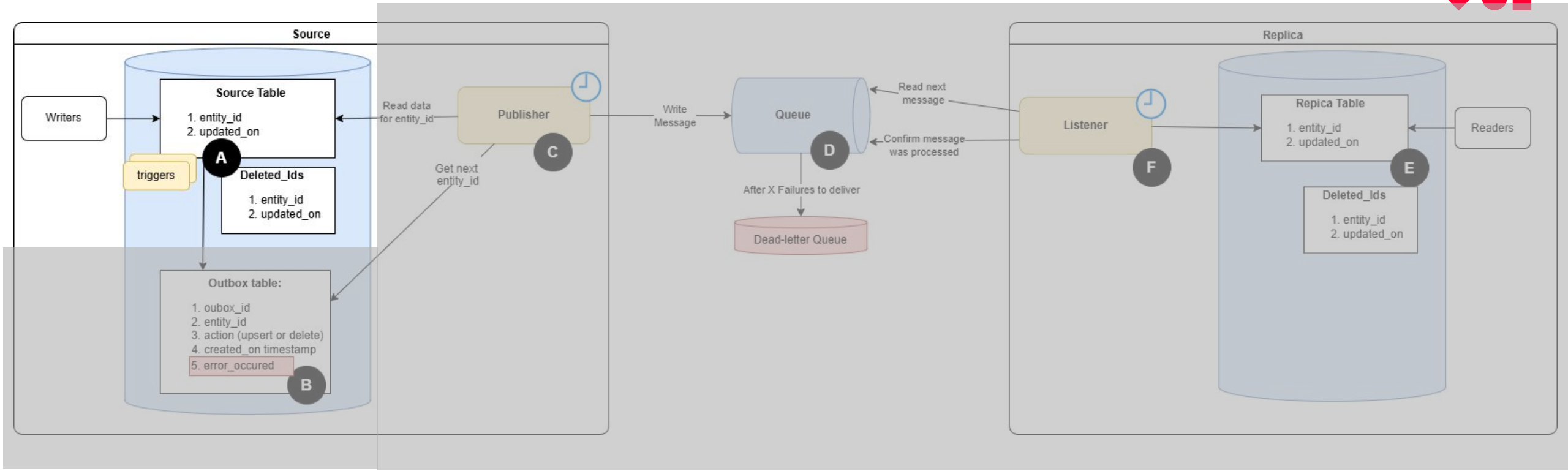
7 COMPONENTS TO GET RIGHT



Extendable to multi-tables:
„DDD Aggregate“

Arrows represent direction of control (not data flow)

A. SOURCE TABLES



„Source“ table structure

1.Entity_ID – primary key, auto-gen
2.Updated_on:
- millisecond granularity
- use database's clock, not writers'

„Deleted_Ids“ table structure

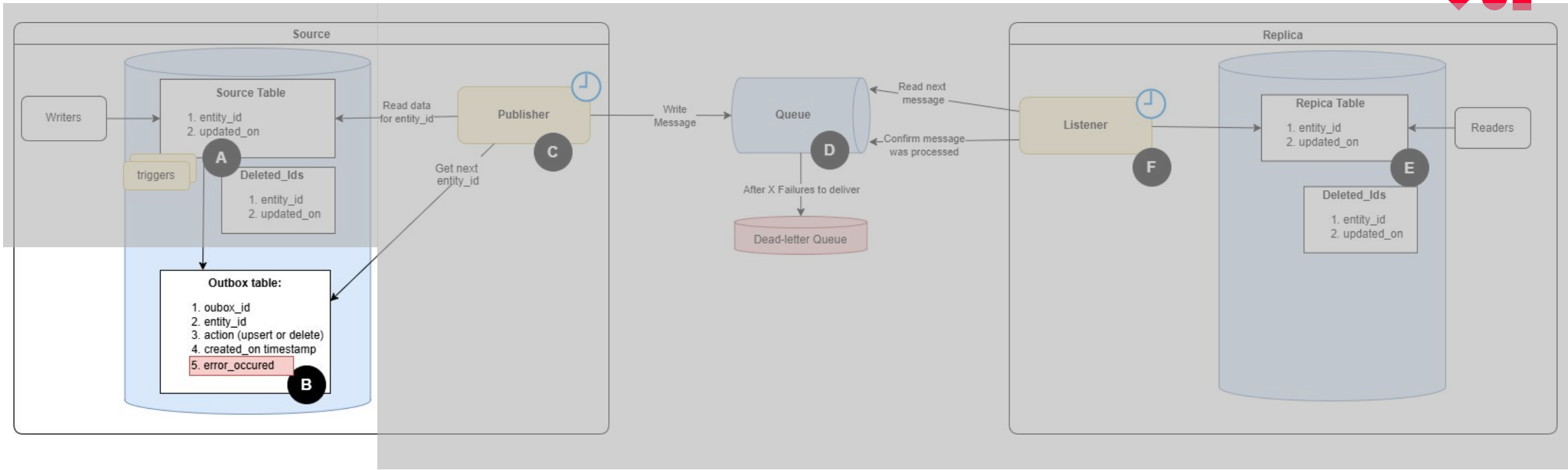
1. Entity_ID – unique key
2. Updated_on – from source table

Is used by Re-synchronization process.

Data model in Replica is identical

Use DB Triggers/Stored Procs:
1) to set value of **updated_on**
2) to write to **Deleted_Ids** table

B. OUTBOX TABLE



Outbox table structure:

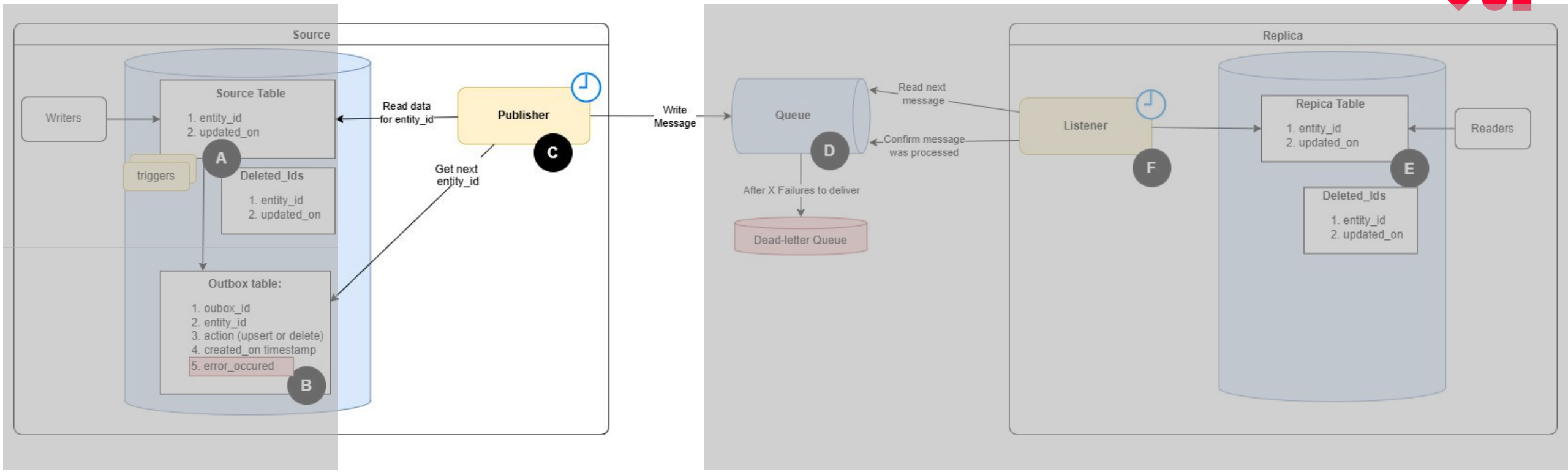
1. ID – auto-gen
2. Entity_ID – from source table
3. Updated_on – from source table
4. Created_on – current timestamp
5. Error_occured (boolean)

Notes:

1. If using AWS AuroraDB consider using Event Bridge as Outbox+Publisher
2. Reference: https://microservices.io/patterns/data/transactional-outbox.html

Use DB Trigger to insert into outbox table

C. PUBLISHER



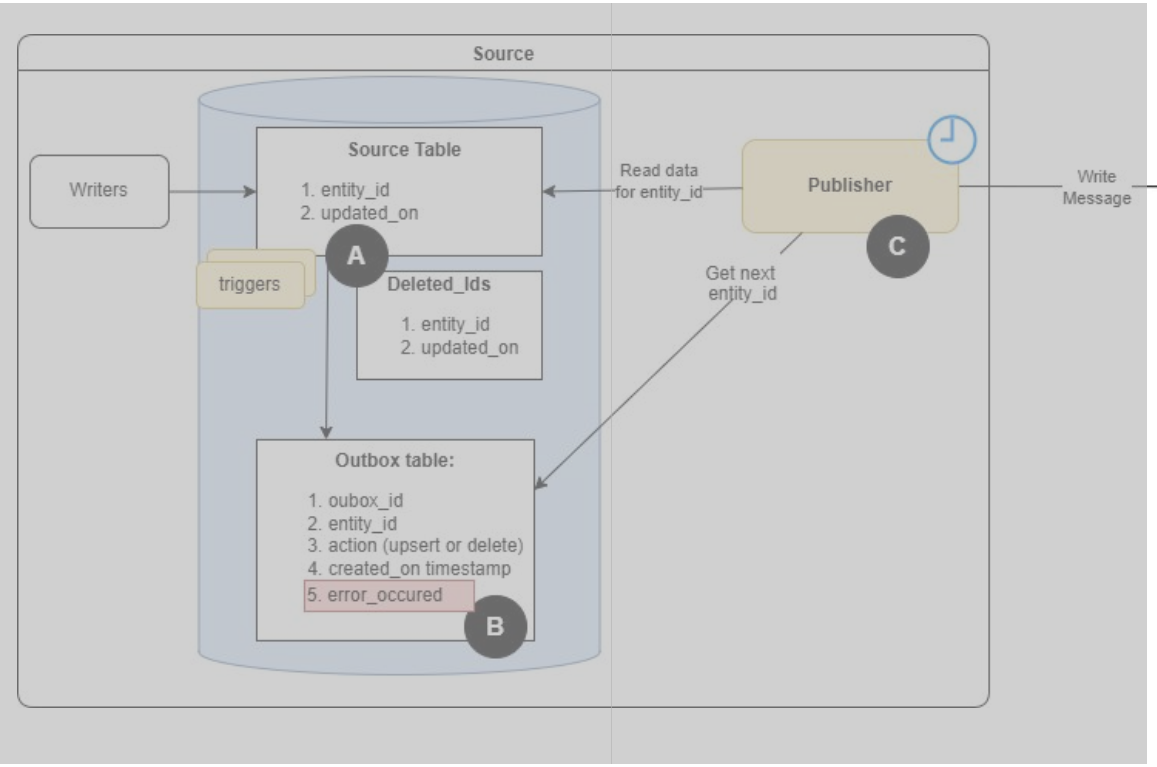
Wake up every X seconds and process rows in Outbox

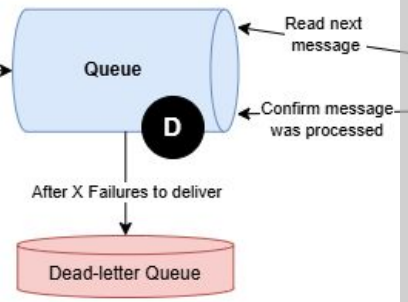
If Error_occured retry Y times, then:
-set error_occured=True,
-notify Operations Engineers,
-move to the next record

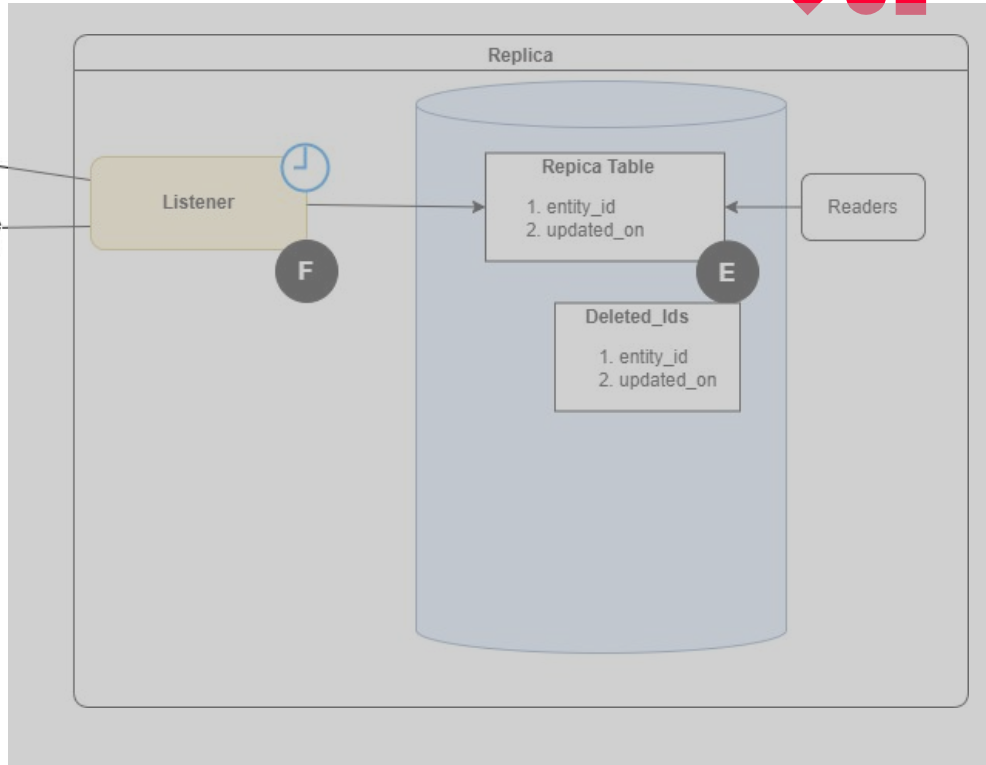
Set Isolation Level to „Repeatable Read“

Set Lock Mode Not Wait

D. QUEUE







Message content:

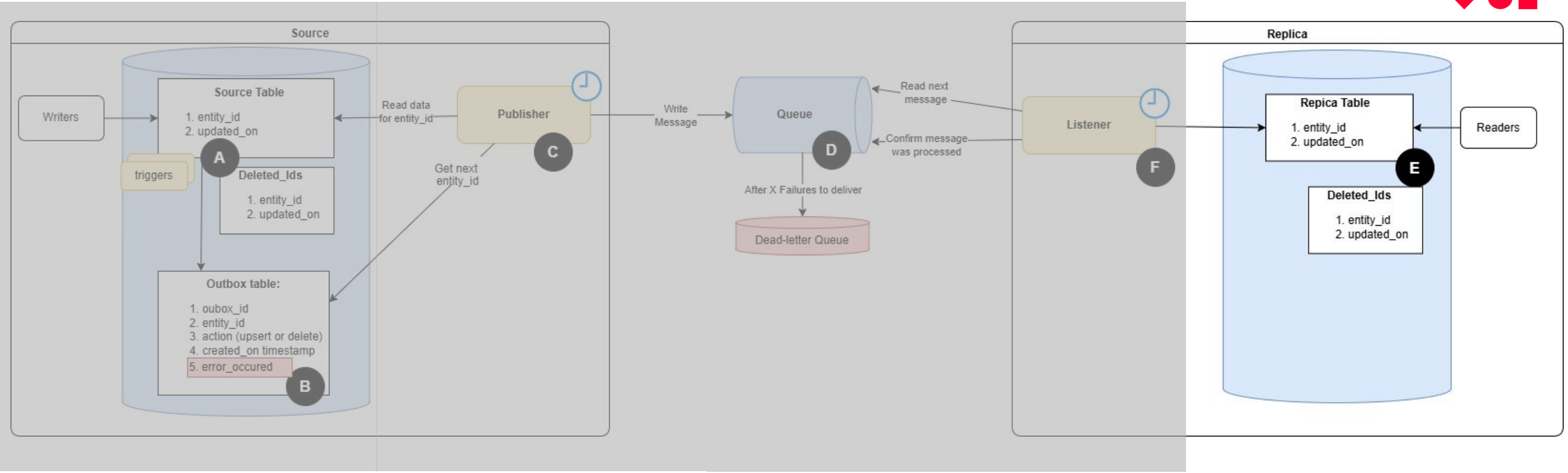
1. Entity_ID
2. Updated_on
3. Action (Upsert or Delete)
4. Full payload – every field.
   a. If a field is missing – it's value must be set to null in Replica.

Notes:

In AWS, SQS is a better choice than Kinesis:
- Easer to scale Listeners – no new Shard needed.
- „Dead Letter Queue" is built-in.
- Easy @JMS integration in Java. Easy to mock.
- Permission-handling is easier.

Queue broker should offer
Dead Letter Queue
to prevent outages from
„poison pills"

E. REPLICA



Replica table structure:

1. Entity_ID – unique constraint
2. Updated _on – not null
3. All other columns are nullable

No other constraints on the Replica table.

Deleted Entity_IDs can never be reused.

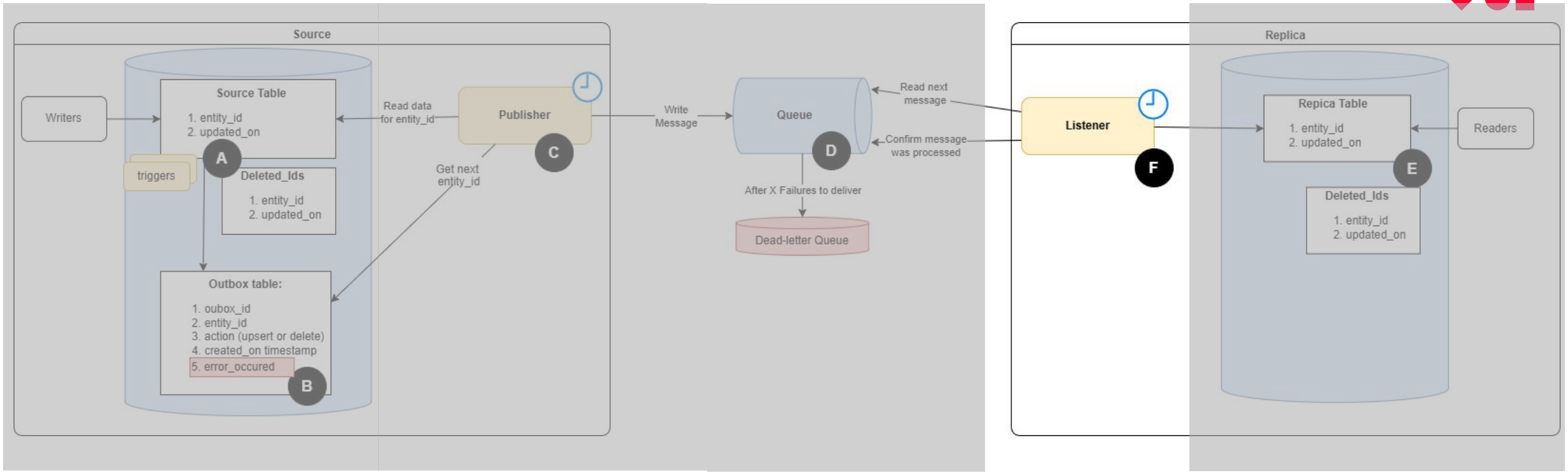
Notes:

Updated_on column is need to avoid „out-of-order Upsert“ scenario.

Deleted_Ids table is needed to avoid „Create-after-Delete“ scenario.

Only Listener can write to Replica

F. LISTENER

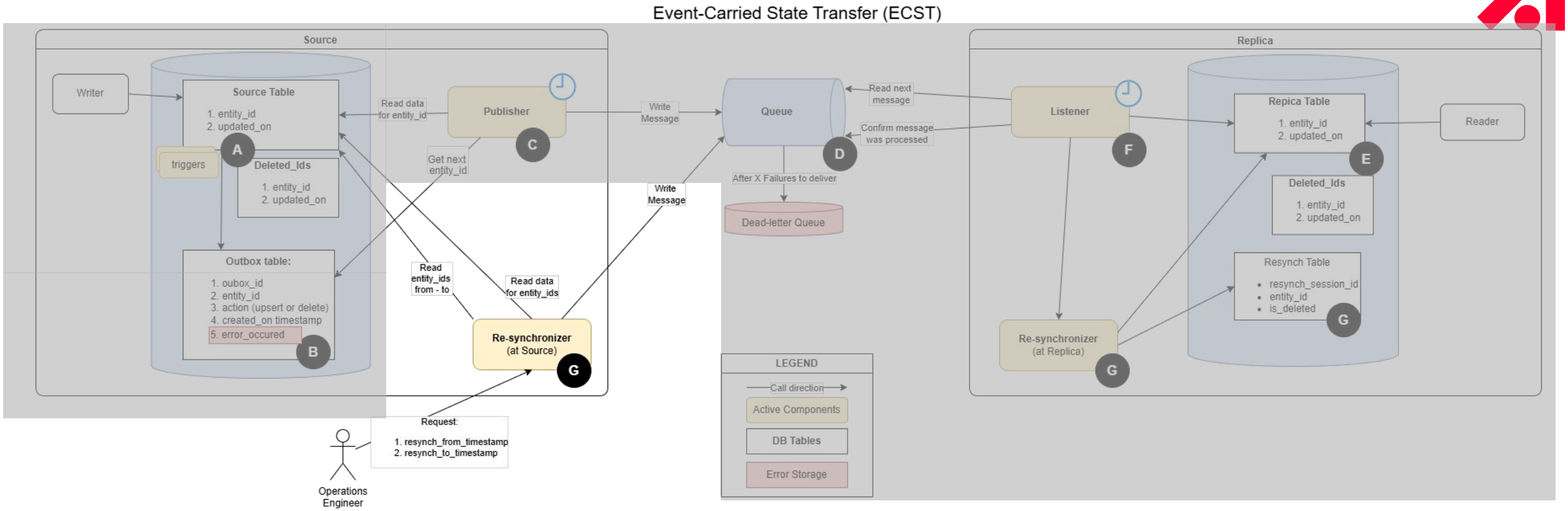


Notes:

- Important to avoid „Phantom record“ scenario while processing „Create-after-Delete“ case.
- Additional logic when processing Re-sync messages
- Different handling for different error scenarios:
  - „exponential backoff“, vs
  - simple „3-times retry“

SERIALIZABLE
Isolation Level

G. RE-SYNCHRONIZER (AT SOURCE)



For details refer to the white paper on Medium.com:

Event-Carried State Transfer — reference architecture

At least implement “At Source” resync

REFERENCES

White Paper - main reference:

- https://medium.com/@max.zalota/event-carried-state-transfer-reference-architecture-26ef49186c44

ECST:

- Martin Fowler: https://martinfowler.com/articles/201701-event-driven.html
- ITNext: https://itnext.io/the-event-carried-state-transfer-pattern-aae49715bb7f

Poison pill:

- https://medium.com/lydtech-consulting/kafka-poison-pill-e146b87c1866

At-least-once message delivery:

- https://bravenewgeek.com/you-cannot-have-exactly-once-delivery/

Outbox pattern:

- https://microservices.io/patterns/data/transactional-outbox.html
- https://medium.com/the-tech-collective/outbox-pattern-providing-reliable-messaging-2603bdfa097f

Domain-Driven-Design Aggregate pattern:

- https://martinfowler.com/bliki/DDD_Aggregate.html

DB Transaction Isolation Level and SQL Query hints:

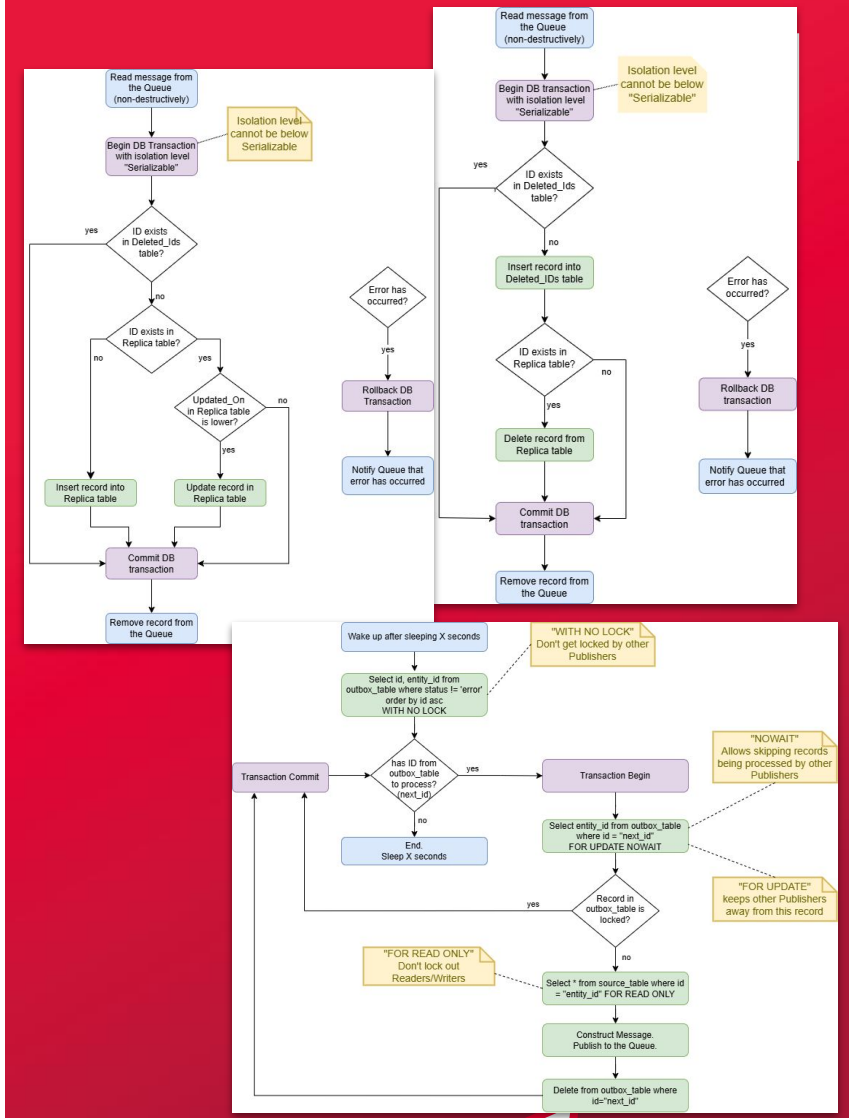
- Informix: https://www.ibm.com/docs/en/informix-servers/14.10.0?topic=statement-read-only-clause
- https://en.wikipedia.org/wiki/Isolation_(database_systems)

Dead letter queue:

- https://www.enterpriseintegrationpatterns.com/patterns/messaging/DeadLetterChannel.html
- https://theburningmonk.com/2023/12/the-one-mistake-everyone-makes-when-using-kinesis-with-lambda/

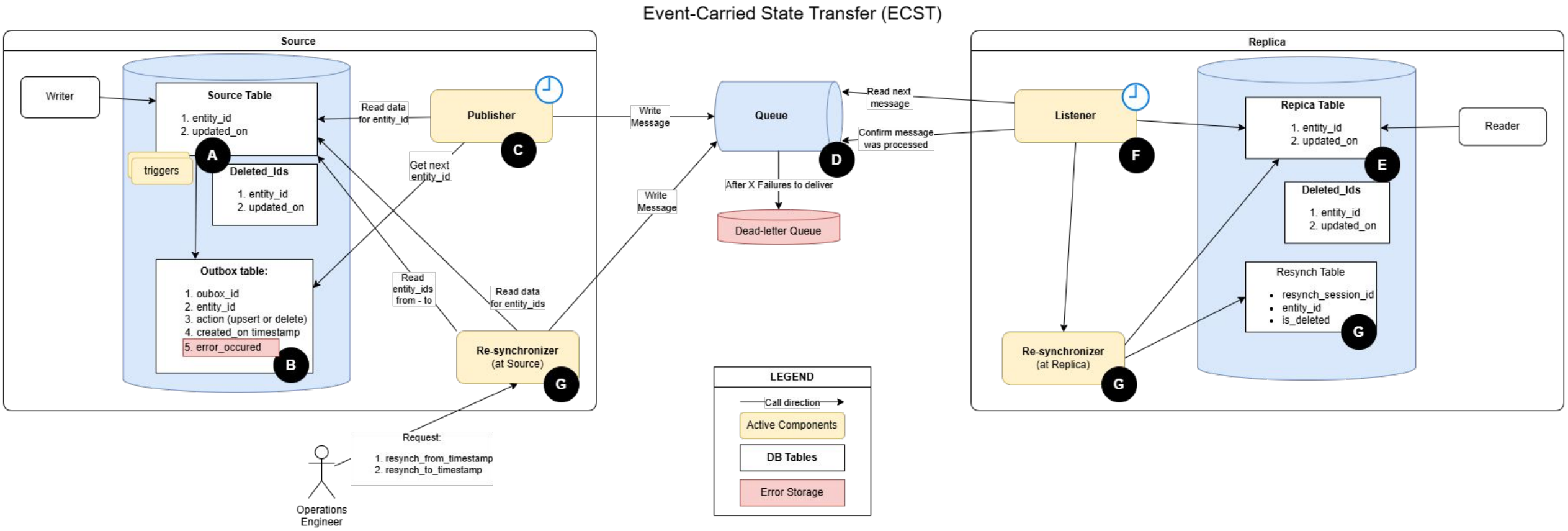
Resynchronization:

- https://softwareengineering.stackexchange.com/questions/401936/event-carried-state-transfer-and-initial-state-synchronization



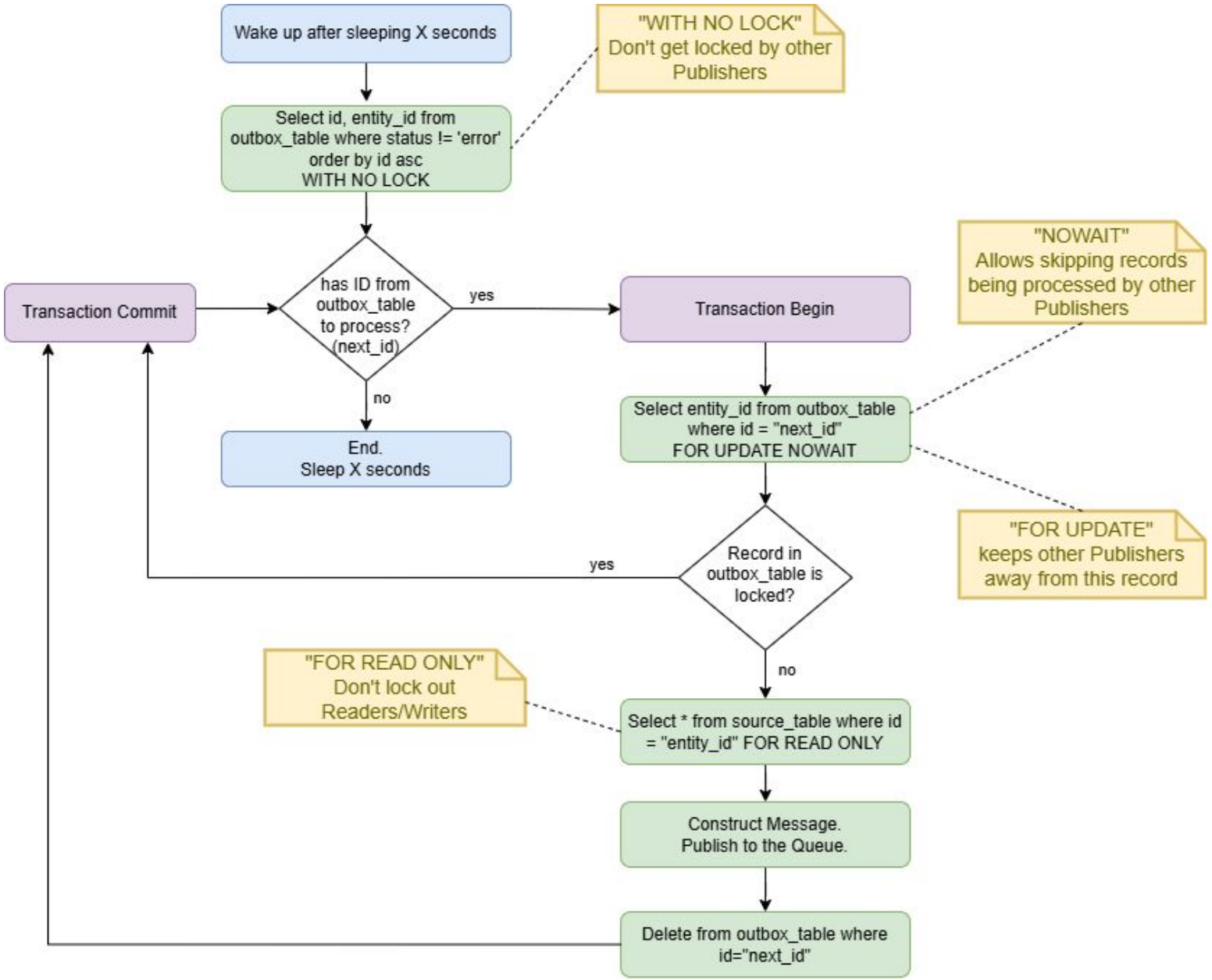
More details in my white paper on medium.com

Q&A


Event-Carried State Transfer (ECST)
Source
Writer
Source Table
1. entity_id
2. updated_on
triggers
A
Deleted_Ids
1. entity_id
2. updated_on
Outbox table:
1. oubox_id
2. entity_id
3. action (upsert or delete)
4. created_on timestamp
5. error_occured
B
Read data for entity_id
Publisher
C
Get next entity_id
Read entity_ids from - to
Read data for entity_ids
Re-synchronizer (at Source)
G
Write Message
Queue
D
Read next message
Confirm message was processed
After X Failures to deliver
Dead-letter Queue
Write Message
Replica
Listener
F
Repica Table
1. entity_id
2. updated_on
E
Reader
Deleted_Ids
1. entity_id
2. updated_on
Resynch Table
resynch_session_id
entity_id
is_deleted
G
Re-synchronizer (at Replica)
G
Request:
1. resynch_from_timestamp
2. resynch_to_timestamp
Operations Engineer
LEGEND
Call direction
Active Components
DB Tables
Error Storage

Appendix

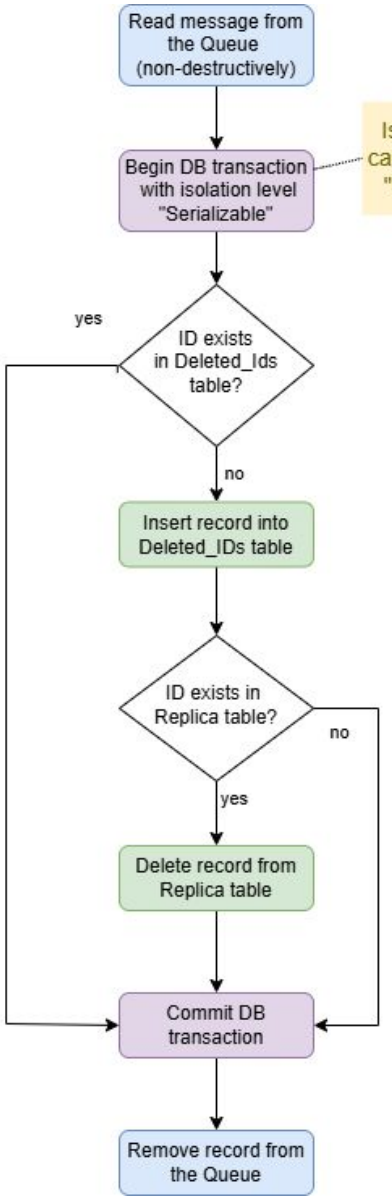
C. PUBLISHER ALGORITHM

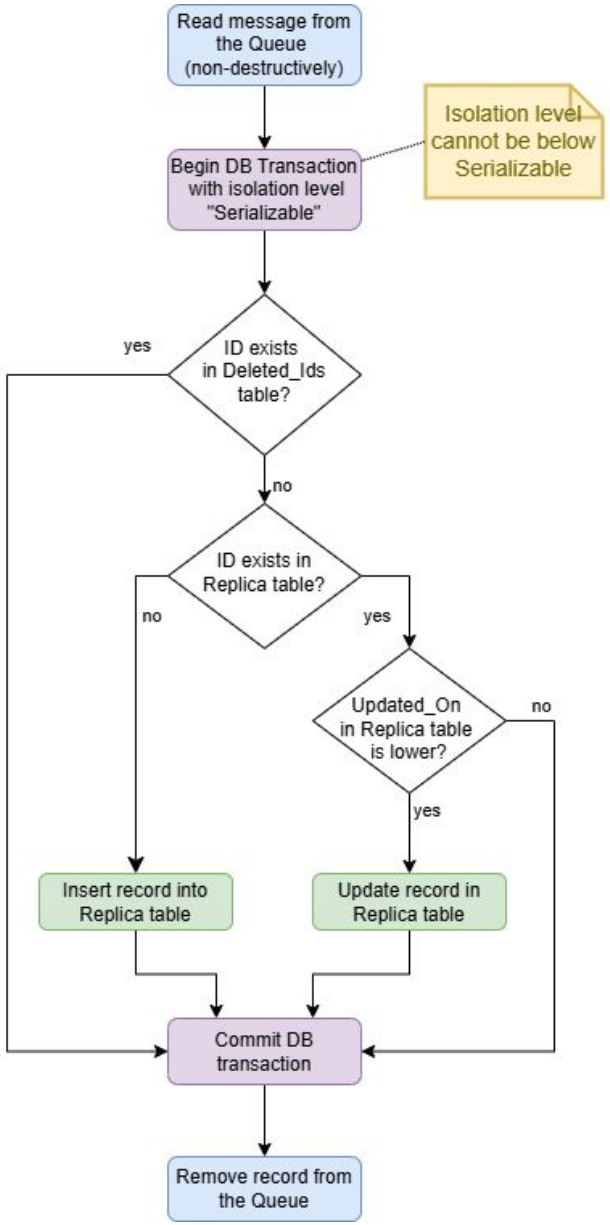


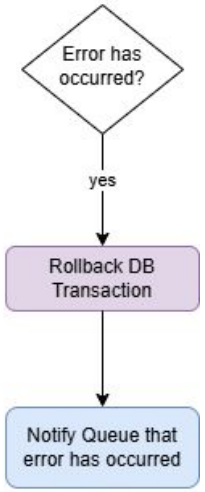
Make use of SQL query hints

Sending a delete message is a bit simpler

F. LISTENER ALGORITHM







1) Delete event

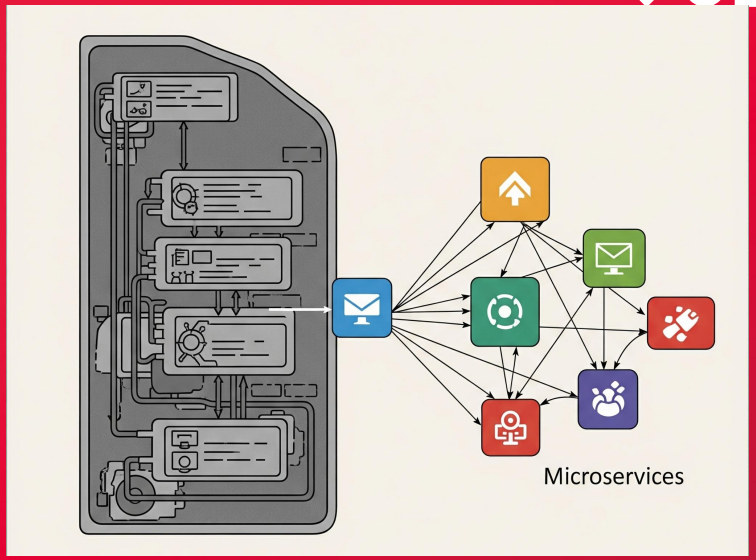
2) Upsert event

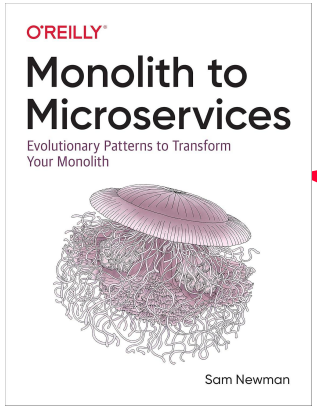
3) On Error

SERIALIZABLE
Isolation Level

OTHER POTENTIAL USE-CASES

1. Refactoring a Monolith into Microservices
2. Micro-batching into Data Warehouse
3. Cross-microservices „JOIN searches“, but GraphQL is too slow.
4. Different view of data, sub-schema.
   - For example, CRM has all accounts.
   - Individual microservices have views relevant for their bounded-contexts: prospects, debtors, support contacts
5. Cache for other system‘s UI for high availability, responsiveness.





Make sure your use case is worth the complexity