Beyond the Happy Path: Failure Recovery with Go and Kafka

Haseeb Majid · Go Labs

Introduction

- Haseeb Majid
  - Backend Software Engineer at Nala
  - https://haseebmajid.dev
- Loves cats 🐱
- Avid cricketer 🏏
- 🌱 Amateur Gardener

TODAY'S ROUTE



The story

Why events?

A payout. A slow bank. A queue.

The foundations

Kafka + Go

Topics, partitions, consumers.

Handling failure

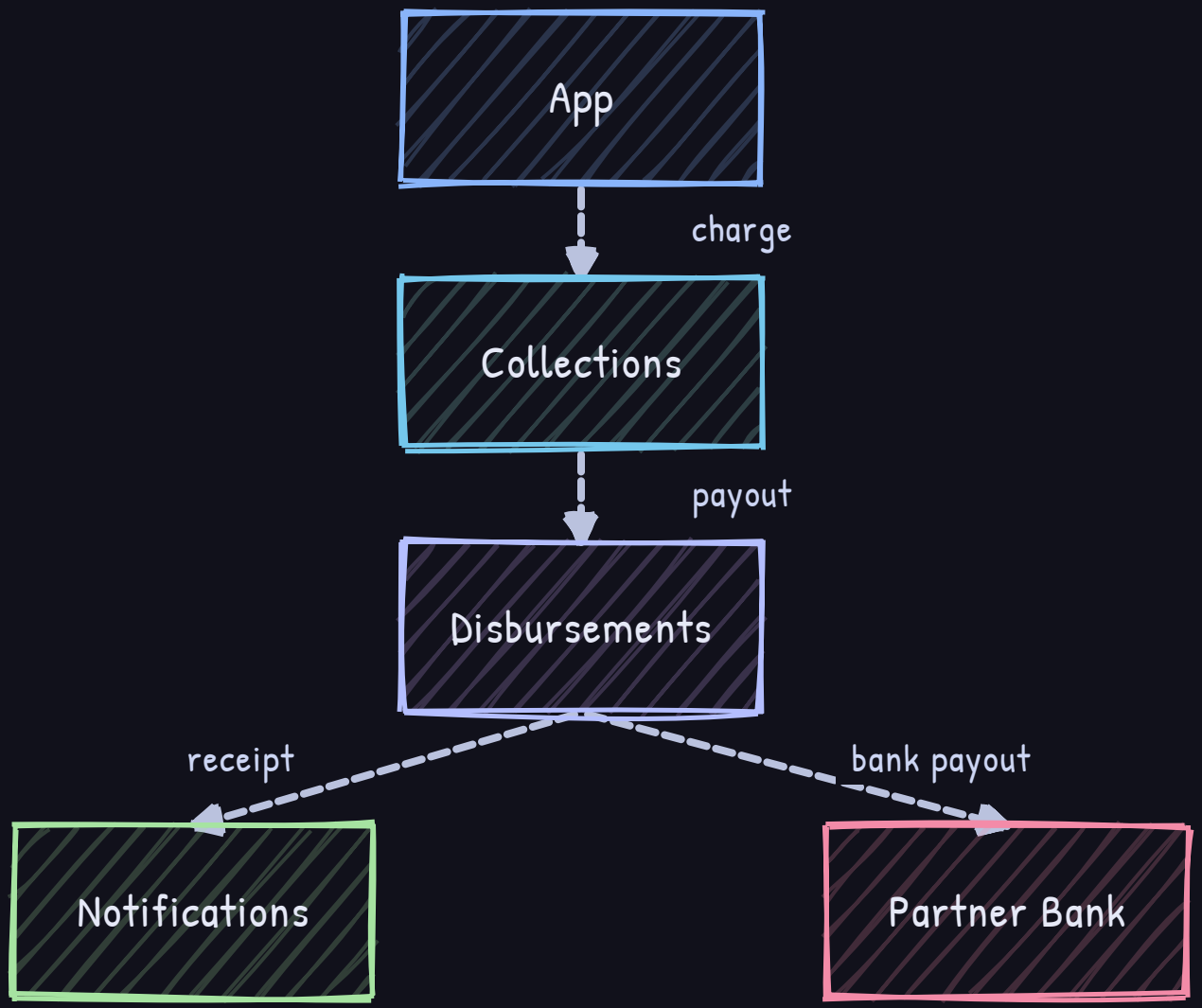
When things fail

Crashes, duplicates, retries.

Recovery

How we can recover

Dead letters, replay, visibility.



1. Charge

Collections charges the customer.

2. Payout

Disbursements requests the bank payout.

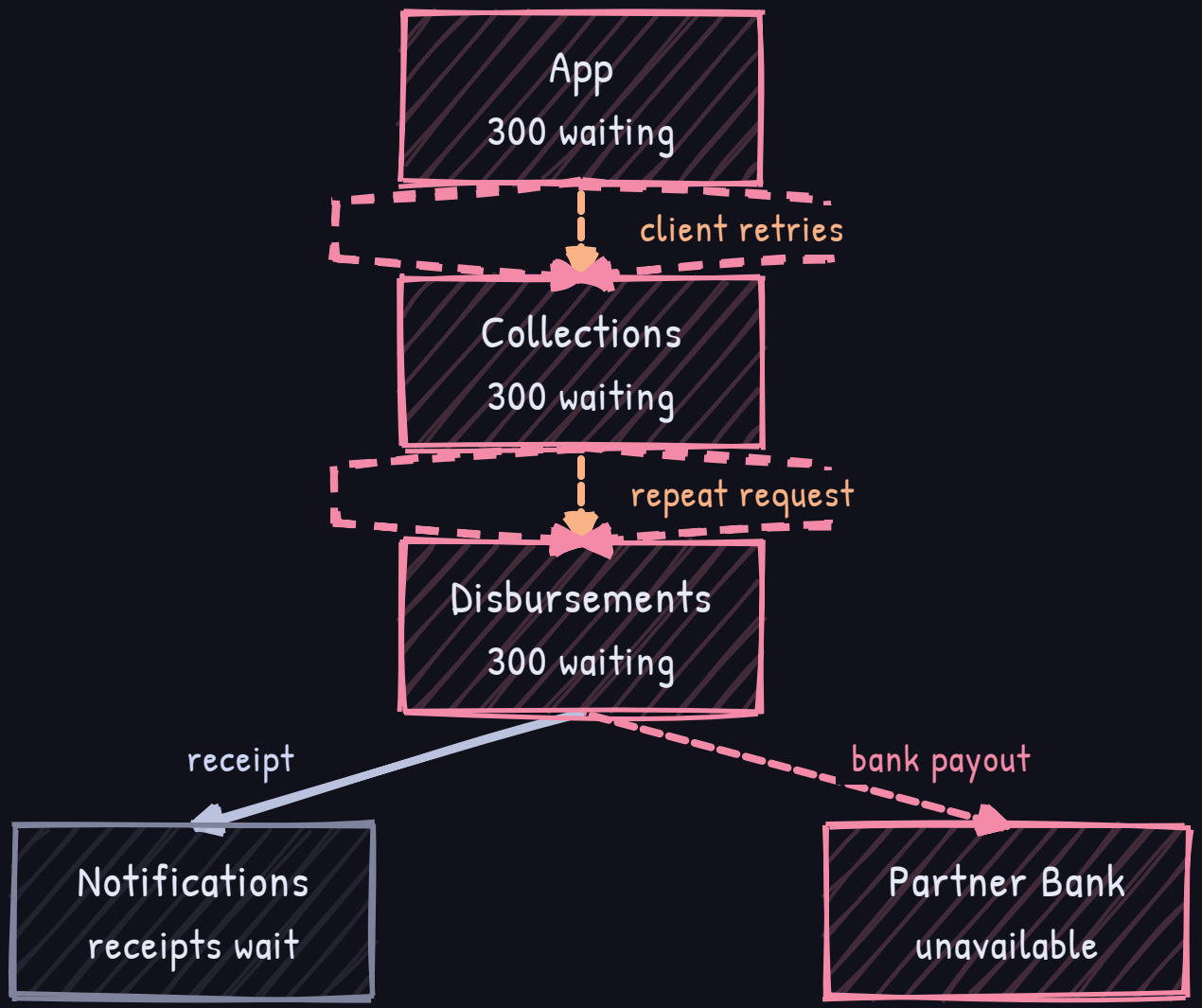
3. Receipt

Notifications sends the receipt.

🎄 Holiday rush

Imagine 1,000 payouts waiting.



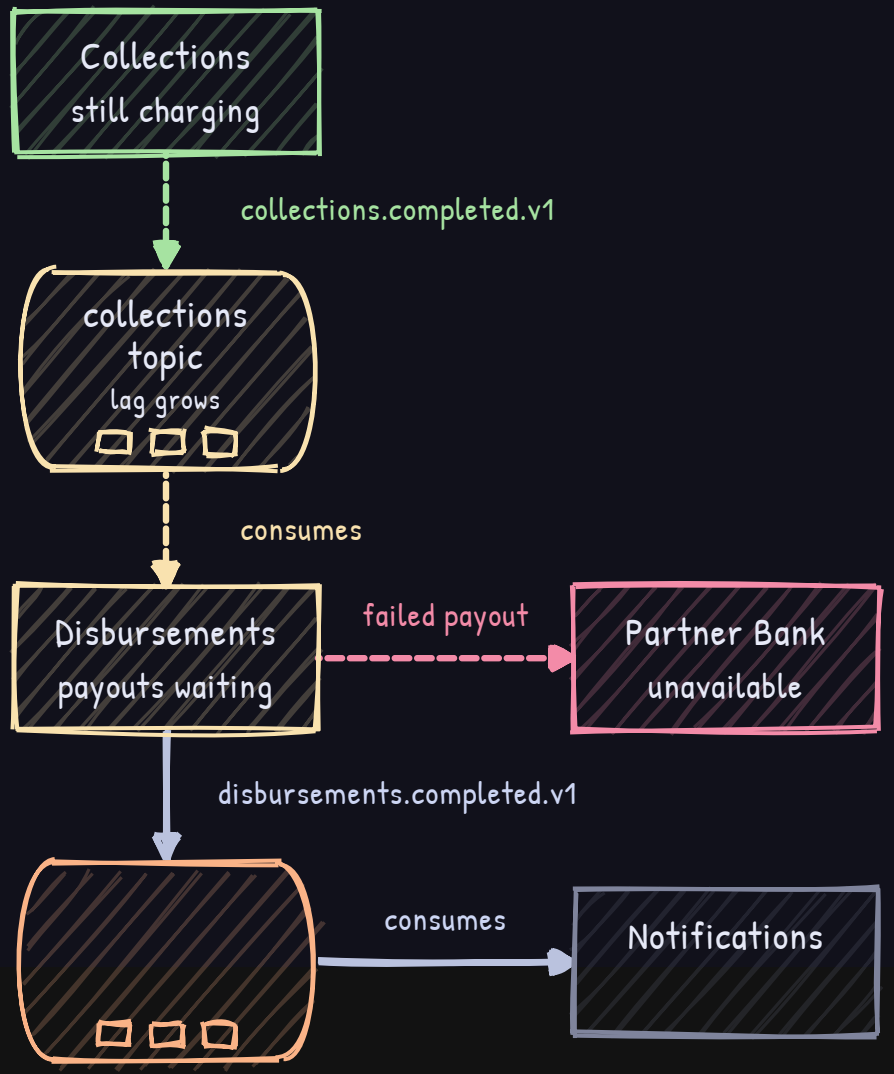


3. Retry storm

Client retries add load and can repeat the same payment request.


FAILURE

Event-driven architecture



4. Bank down

Collections keeps working. Payouts and receipts wait.



Kafka Crash Course

without Otters 

Event: a fact that happened in the past.

```go
type Event struct {
    ID            uuid.UUID       `json:"id"`
    Type          string          `json:"type"`
    Payload       json.RawMessage `json:"payload"`
}

const (
    TypeCollectionCompleted = "collections.completed.v1"
)
```

collections.completed.v1

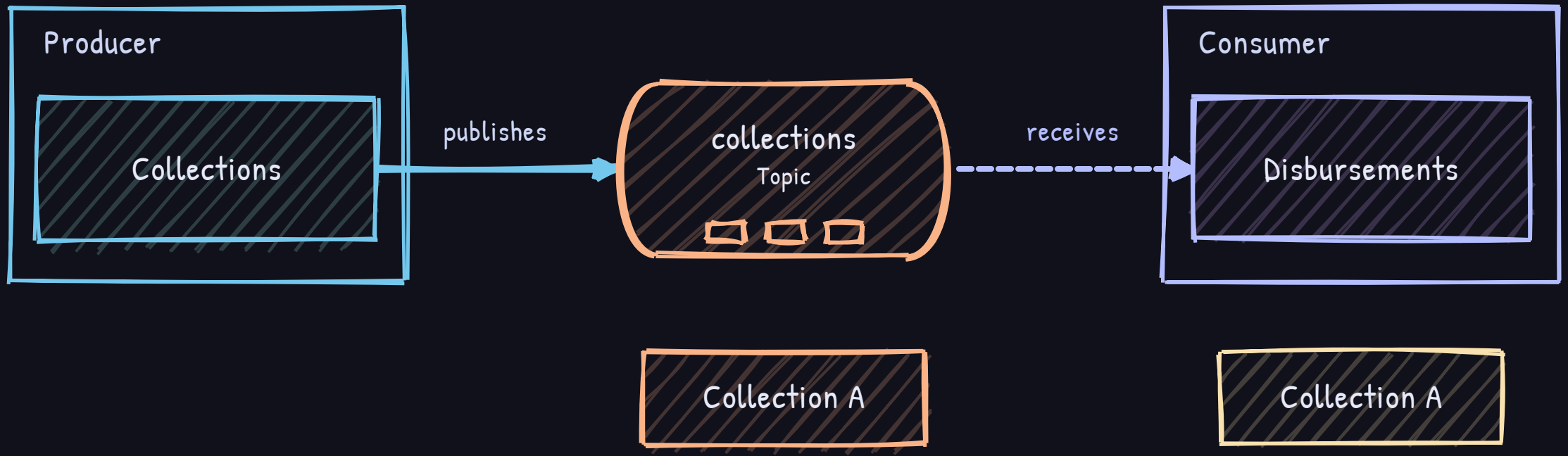
```
{
  "id": "92bf4e5a-3a1e-4d6f-8db8-1f7d865bd46a",
  "type": "collections.completed.v1",
  "source": "collections",
  "correlation_id": "collection-a",
  "time": "2026-06-01T10:30:00Z",
  "schema_version": 1,
  "payload": {
    "collection_id": "collection-42",
    "amount_minor": 10000,
    "currency": "GBP"
  }
}
```

Topics: How events are organised


Topic: collections
Collection A
collections.created.v1
Collection A
collections.updated.v1
Collection A
collections.completed.v1

Producer: publishes event to topic

Consumer: receives events from a topic


Producer
Collections
publishes
collections
Topic
receives
Consumer
Disbursements
Collection A
Collection A



```
go get github.com/twmb/franz-go
```



Why franz-go?

- **Pure Go**: No CGO
- **Idiomatic API**: contexts and errors
- **Well Supported**: KIPs Implemented

Also useful

- Manual commits
- Idempotent producer by default
- Prometheus / OpenTelemetry hooks

```
err := client.ProduceSync(ctx, &kgo.Record{
    Topic: "collections",
    Key:   []byte(collection.ID.String()),
    Value: eventJSON,
}).FirstErr()
```

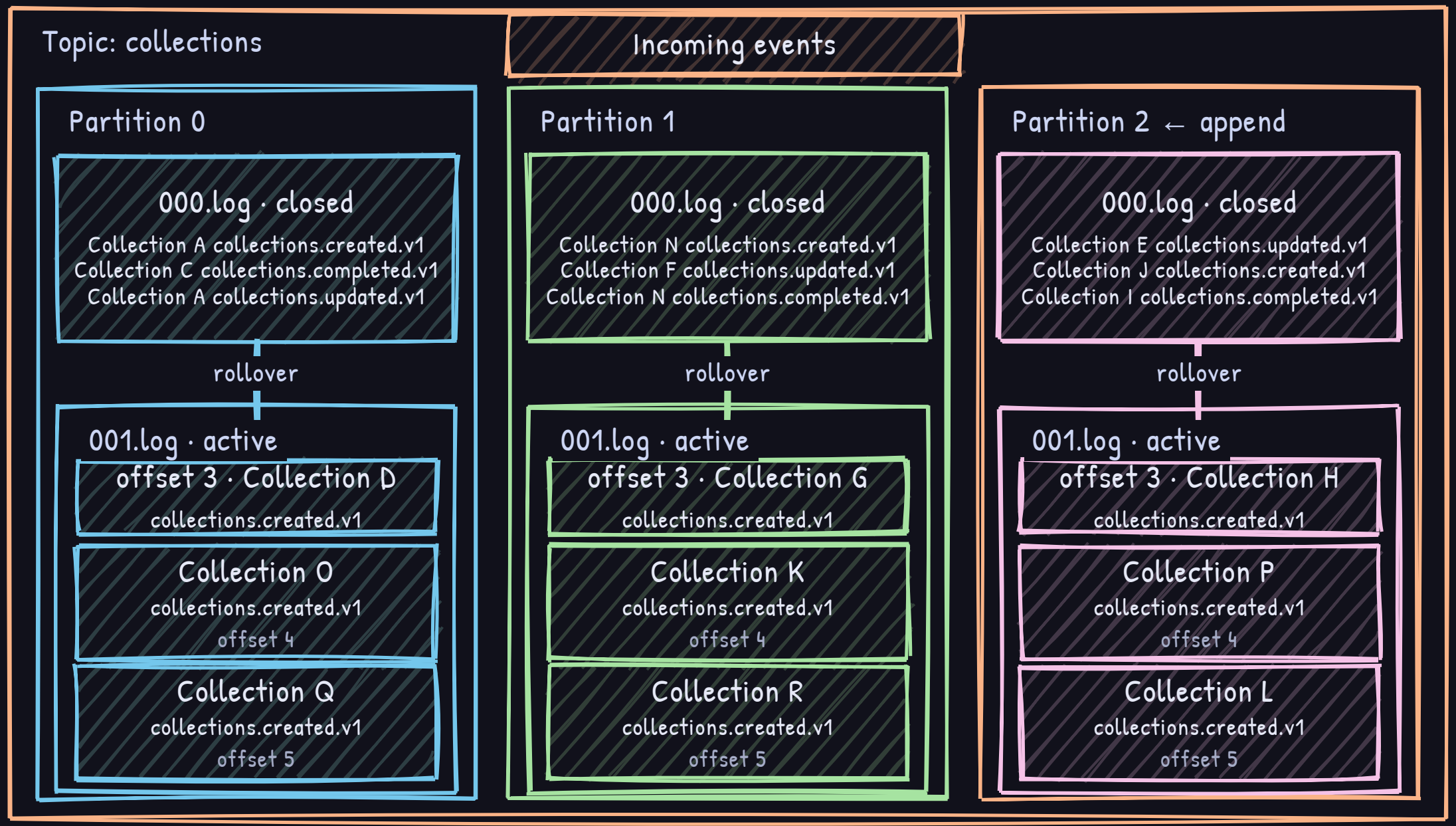
```
func handleCollectionCompleted(
    ctx context.Context, r *kgo.Record,
) error {
    collection, err := decodeCollection(r.Value)
    if err != nil { return err }

    _, err = bank.CreatePayment(ctx, PaymentRequest{
        CollectionID: collection.ID,
        Amount:       collection.Amount,
    })
    return err
}
```

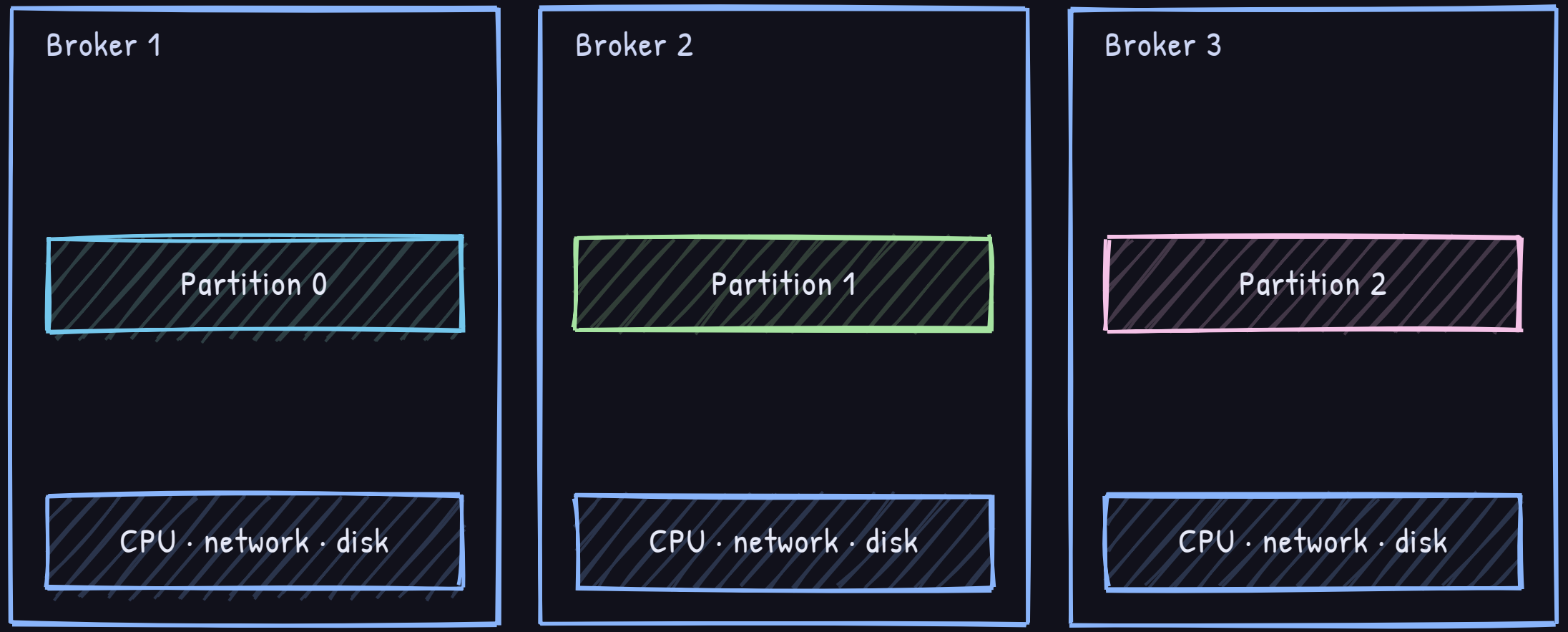

Kafka · collections
Collection A
Collection B
Collection C
Consumer
PollFetches(ctx)
Returned records
Handle one at a time
Collection A ✓
Collection B ✓
Collection C ✓

Partition: an ordered, append-only log within a topic.

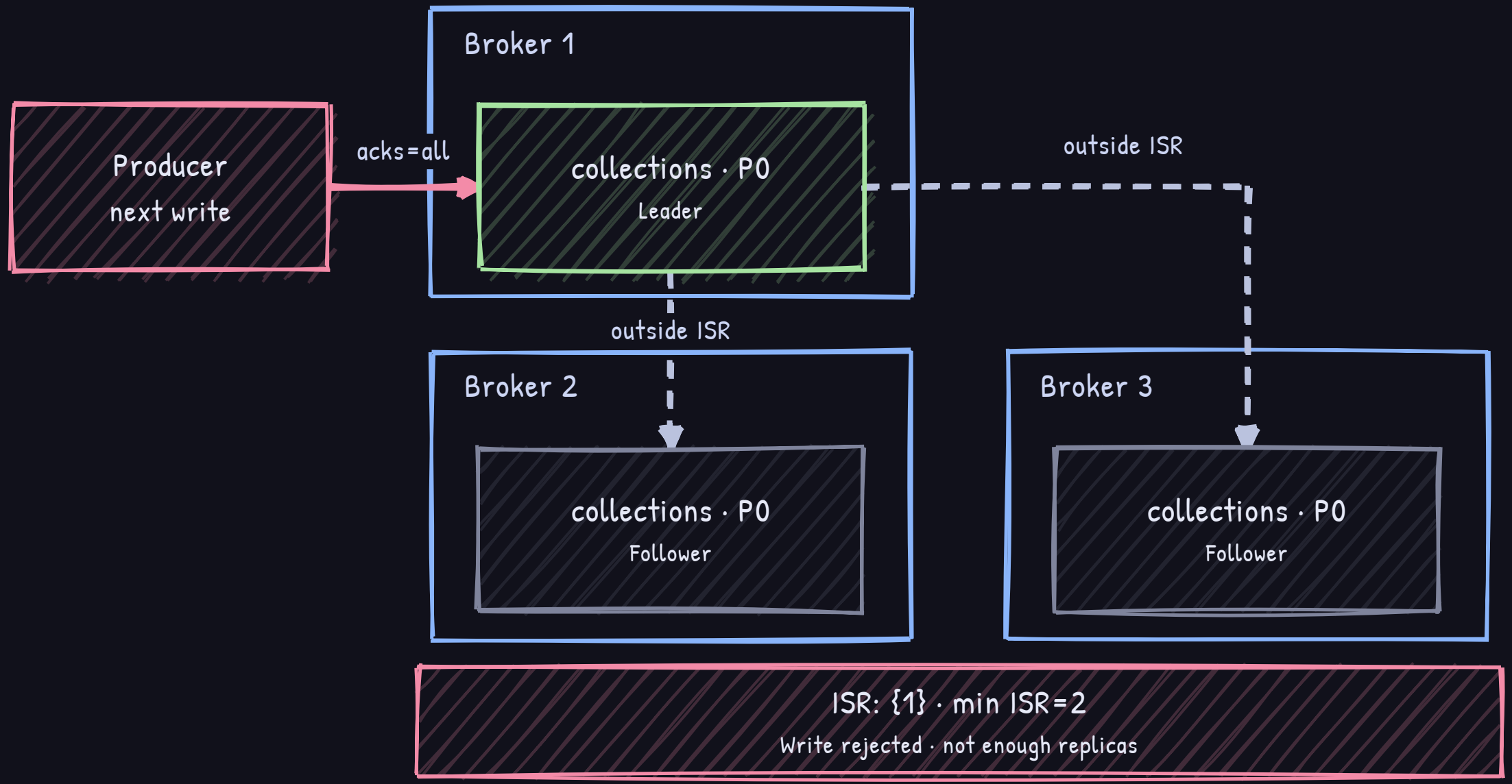

Topic: collections
Incoming events
Partition 0
Collection A
offset 0
Collection C
offset 1
Collection A
offset 2
Collection D
offset 3
Collection O
offset 4
Collection Q
offset 5
Partition 1
Collection N
offset 0
Collection F
offset 1
Collection N
offset 2
Collection G
offset 3
Collection K
offset 4
Collection R
offset 5
Partition 2
Collection E
offset 0
Collection J
offset 1
Collection I
offset 2
Collection H
offset 3
Collection P
offset 4
Collection L
offset 5


Topic: collections
Incoming events
Partition 0
000.log · closed
Collection A collections.created.v1
Collection C collections.completed.v1
Collection A collections.updated.v1
rollover
001.log · active
offset 3 · Collection D
collections.created.v1
Collection O
collections.created.v1
offset 4
Collection Q
collections.created.v1
offset 5
Partition 1
000.log · closed
Collection N collections.created.v1
Collection F collections.updated.v1
Collection N collections.completed.v1
rollover
001.log · active
offset 3 · Collection G
collections.created.v1
Collection K
collections.created.v1
offset 4
Collection R
collections.created.v1
offset 5
Partition 2 ← append
000.log · closed
Collection E collections.updated.v1
Collection J collections.created.v1
Collection I collections.completed.v1
rollover
001.log · active
offset 3 · Collection H
collections.created.v1
Collection P
collections.created.v1
offset 4
Collection L
collections.created.v1
offset 5


IT ALWAYS HAS BEEN
WAIT KAFKA IS JUST LOGS
imgflip.com


Broker 1
Partition 0
CPU · network · disk
Broker 2
Partition 1
CPU · network · disk
Broker 3
Partition 2
CPU · network · disk

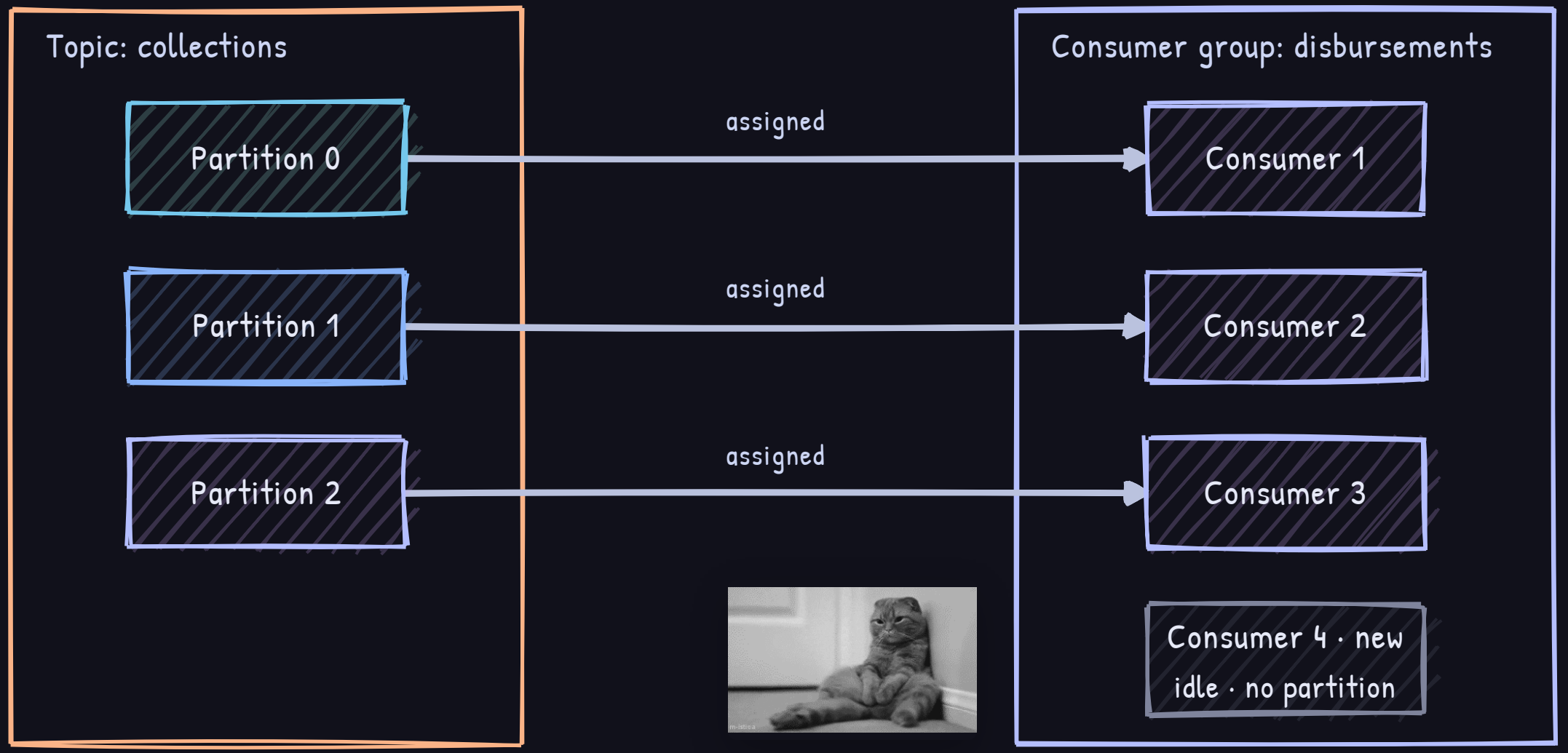
- **ISR (In-Sync Replicas)**: replicas of a partition that are caught up with its leader

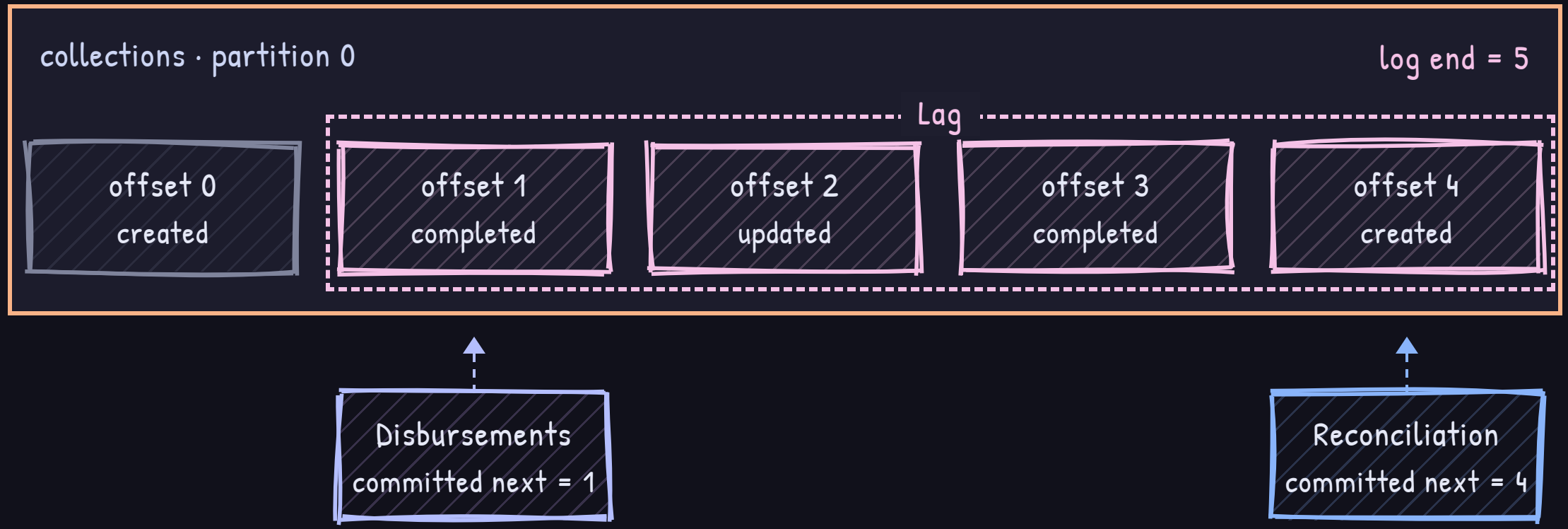

Broker 1
Producer
next write
acks=all
collections · P0
Leader
outside ISR
outside ISR
Broker 2
collections · P0
Follower
Broker 3
collections · P0
Follower
ISR: {1} · min ISR=2
Write rejected · not enough replicas

```
admin := kadm.NewClient(client)
minISR := "2"

_, err = admin.CreateTopic(ctx,
    3, // partitions
    3, // replication factor
    map[string]*string{"min.insync.replicas": &minISR},
    "collections",
)
if err != nil { return err }
```

Consumer Group: a set of consumers that share a topic's partitions.

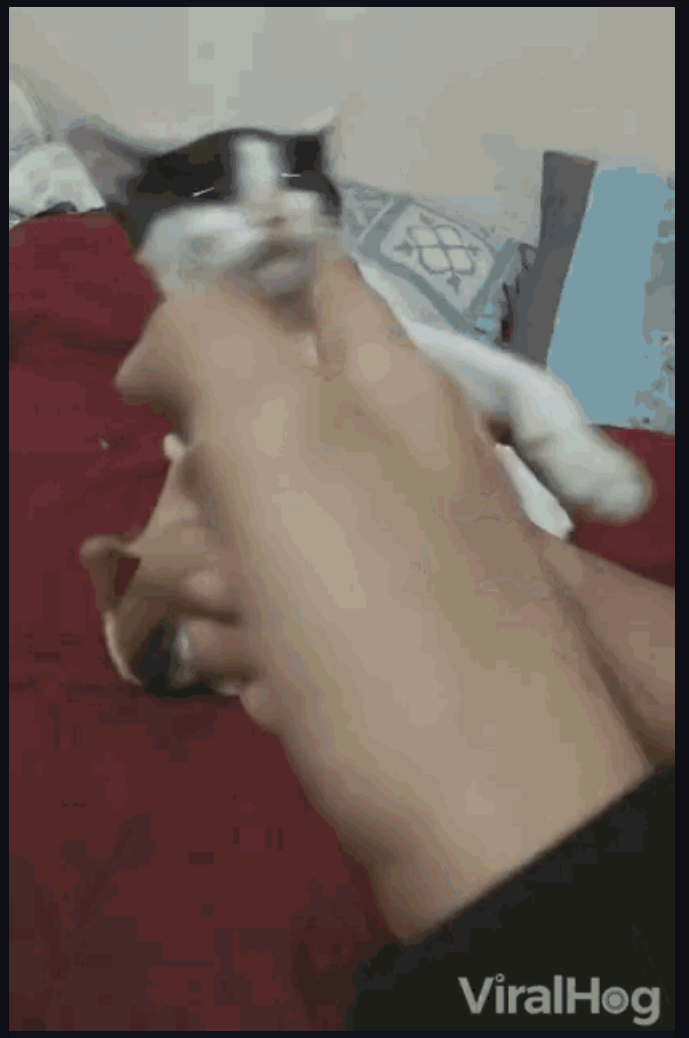

Topic: collections
Partition 0
Partition 1
Partition 2
assigned
assigned
assigned
Consumer group: disbursements
Consumer 1
Consumer 2
Consumer 3
Consumer 4 · new
idle · no partition
No work for me...

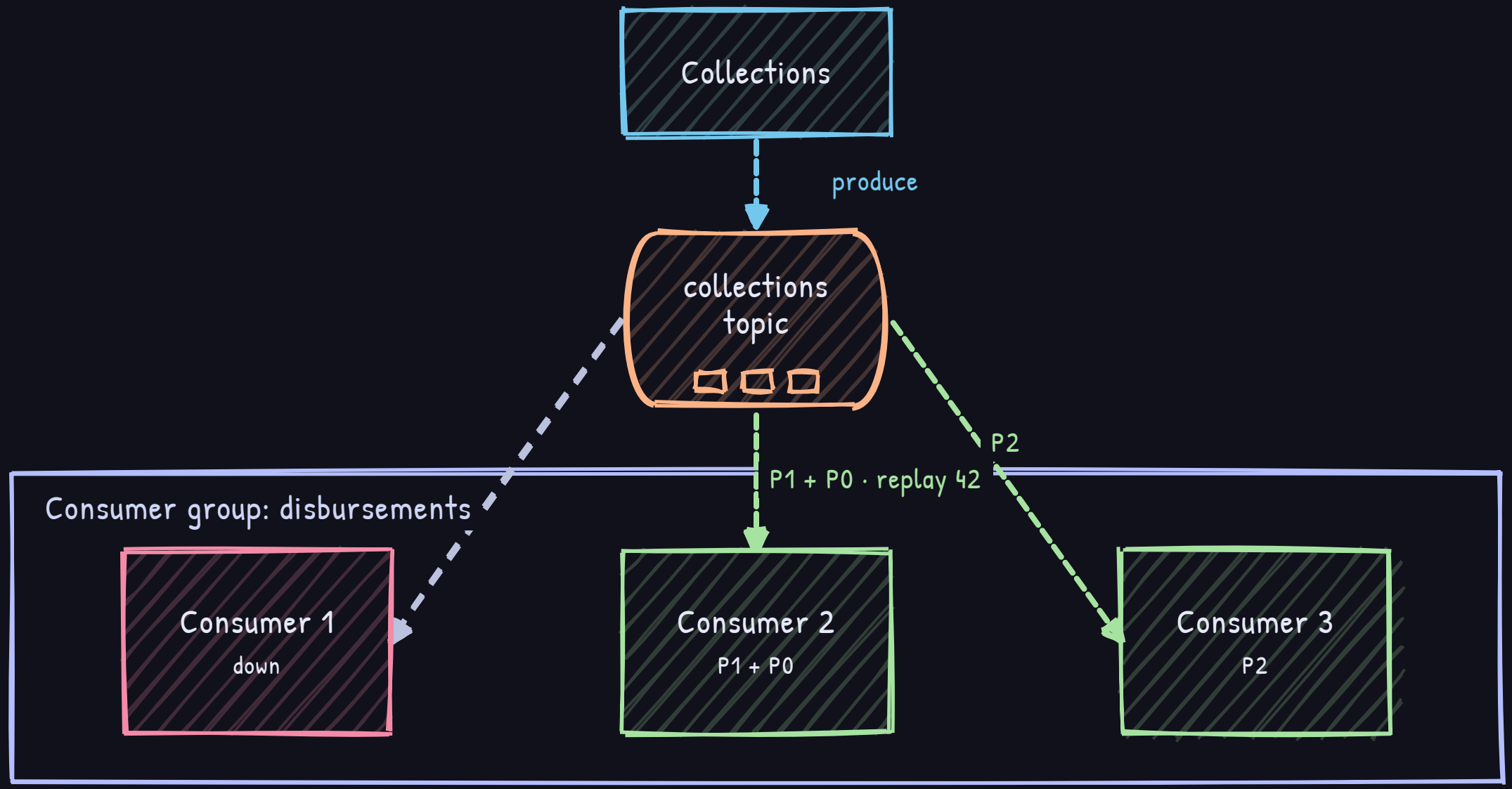

collections · partition 0
log end = 5
Lag
offset 0
created
offset 1
completed
offset 2
updated
offset 3
completed
offset 4
created
Disbursements
committed next = 1
Reconciliation
committed next = 4

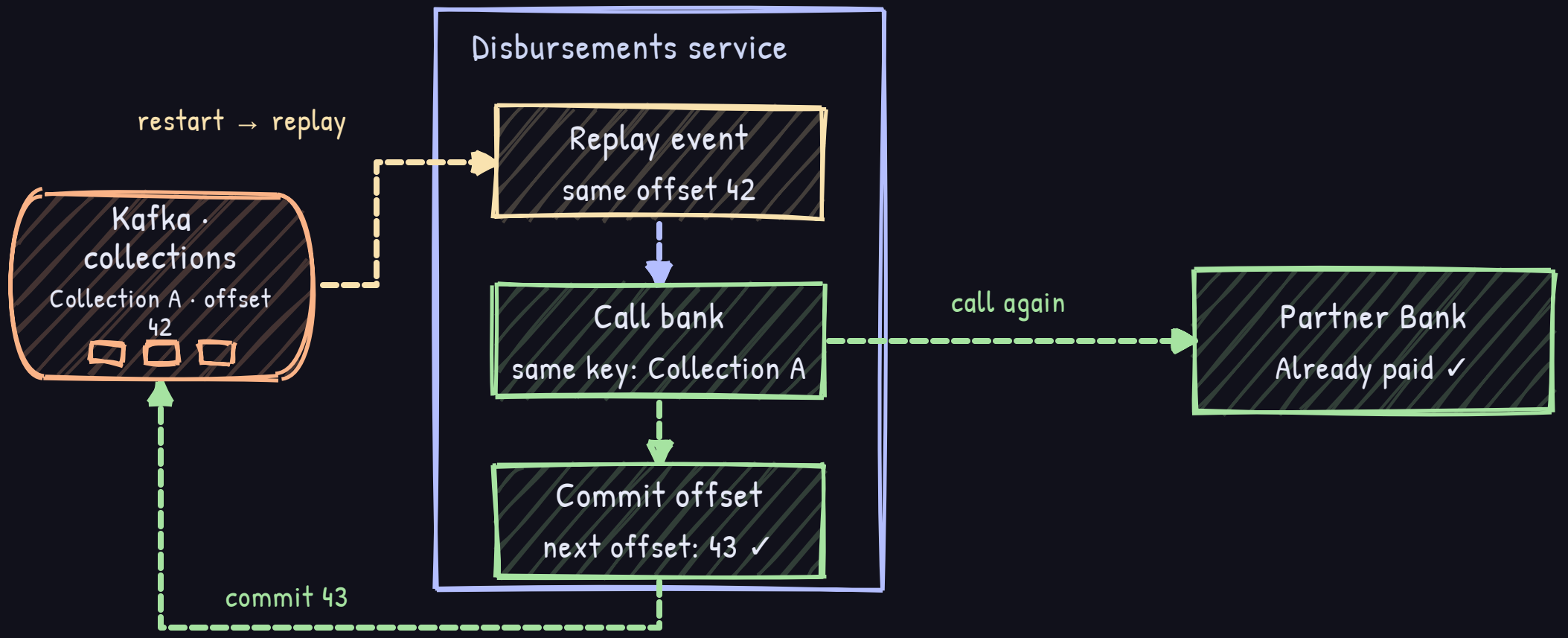
```
for _, r := range fetches.Records() {
    if err := handleCollectionCompleted(ctx, r); err != nil {
        return err
    }

    if err := client.CommitRecords(ctx, r); err != nil {
        return err
    }
}
```

What if a consumer dies?

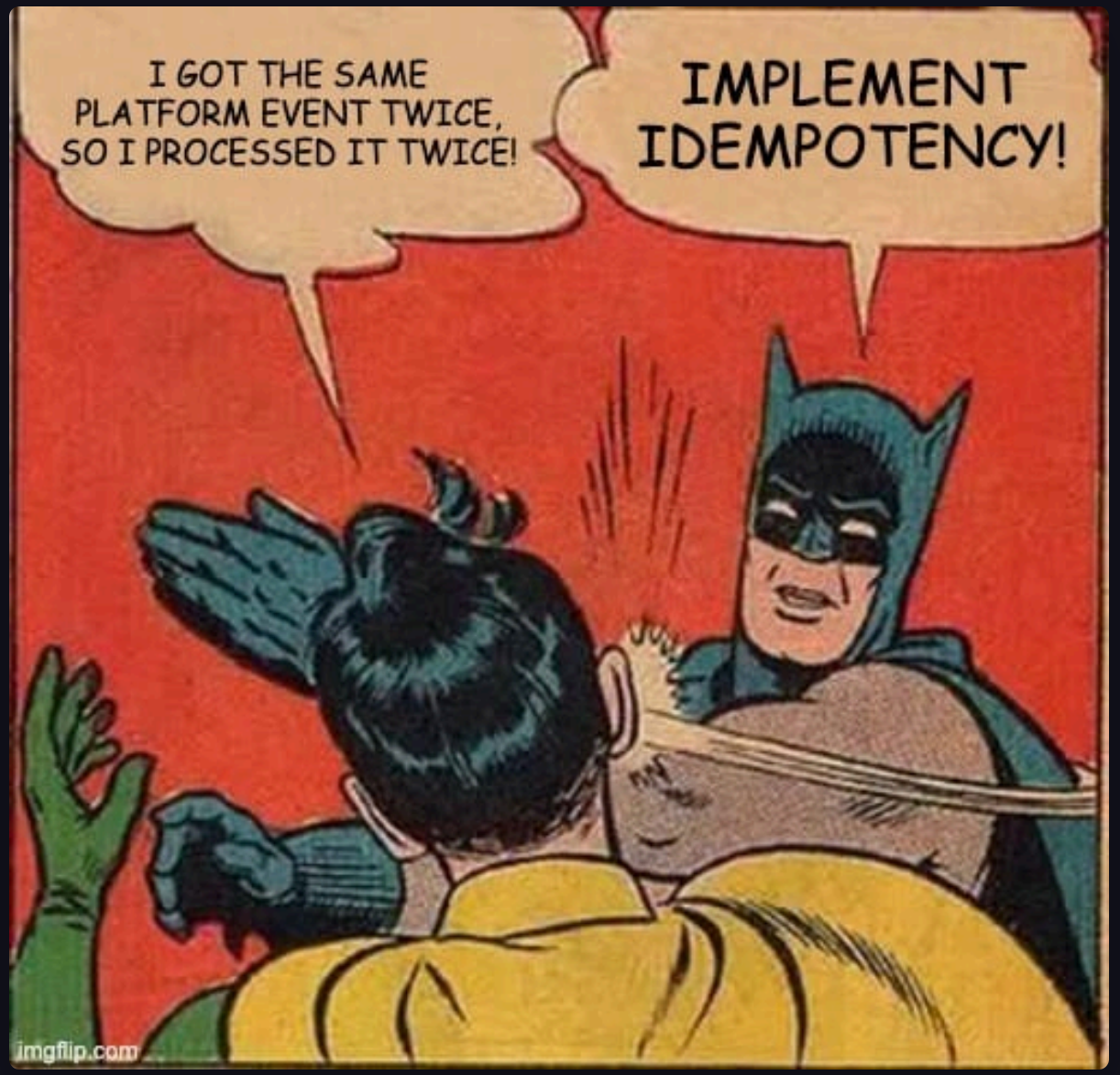


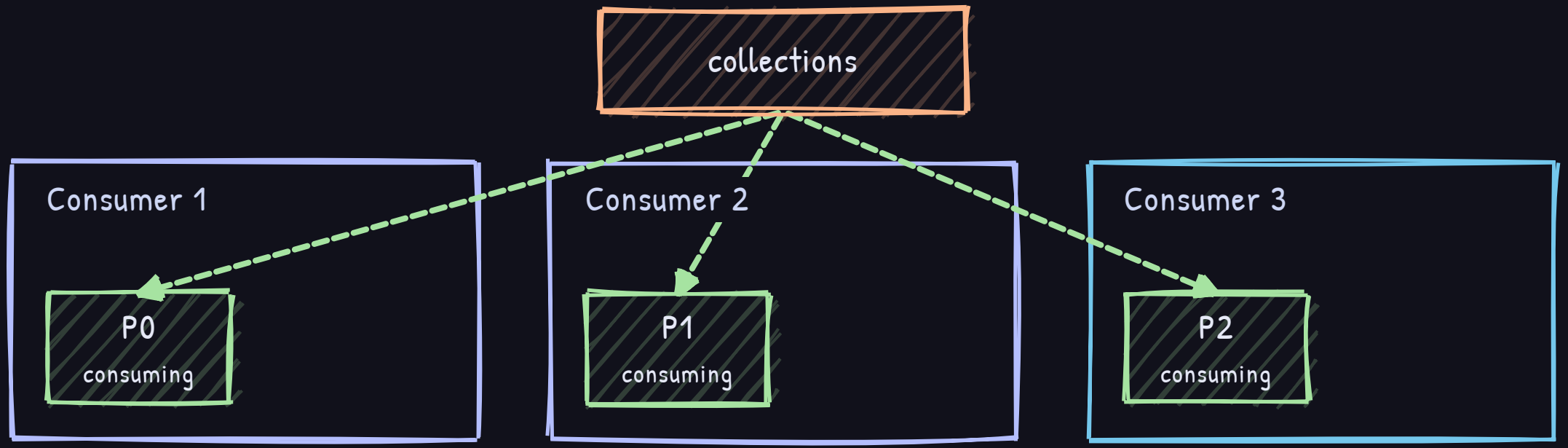

Collections
produce
collections topic
P2
P1 + P0 · replay 42
Consumer group: disbursements
Consumer 1
down
Consumer 2
P1 + P0
Consumer 3
P2

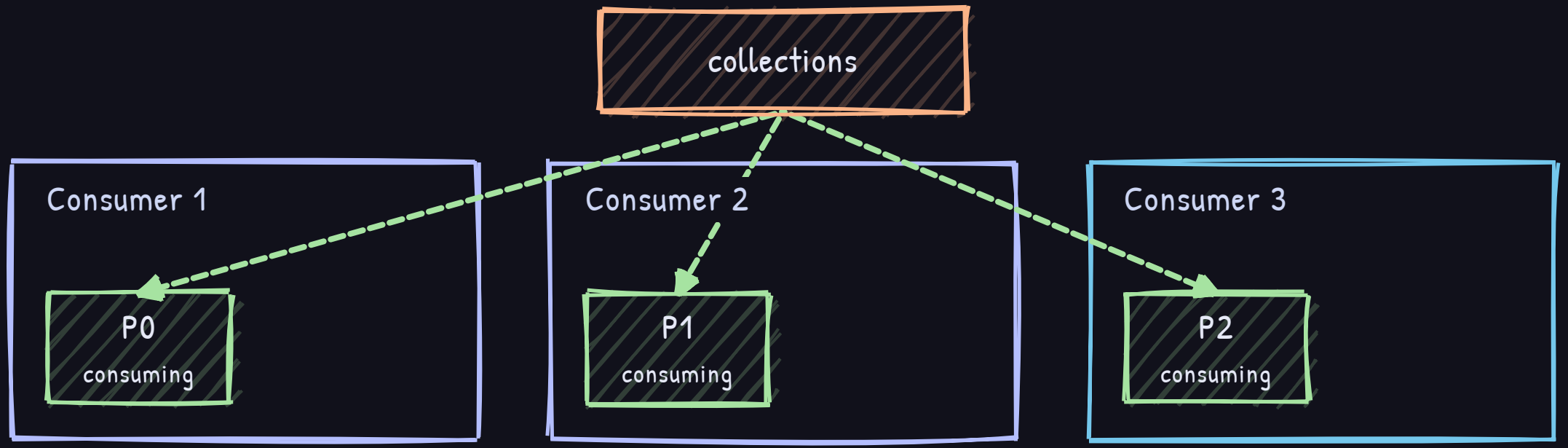


3. With a stable key

Use the same key on the original call and replay. The bank returns the original result; then commit.

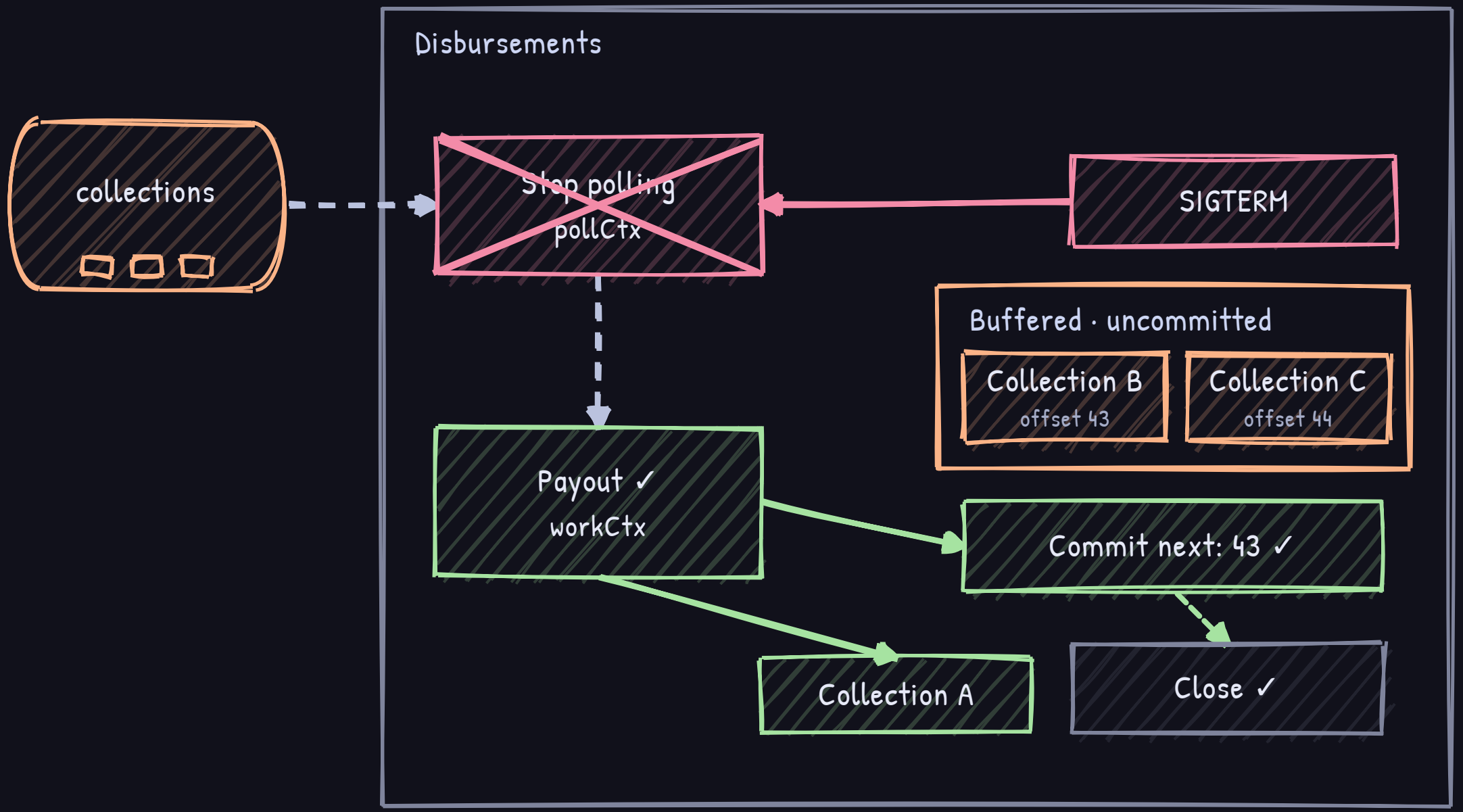

I GOT THE SAME PLATFORM EVENT TWICE, SO I PROCESSED IT TWICE!
IMPLEMENT IDEMPOTENCY!
imgflip.com


collections
Consumer 1
P0
consuming
Consumer 2
P1
consuming
Consumer 3
P2
consuming


collections
Consumer 1
P0
consuming
Consumer 2
P1
consuming
Consumer 3
P2
consuming

Cooperative · sticky

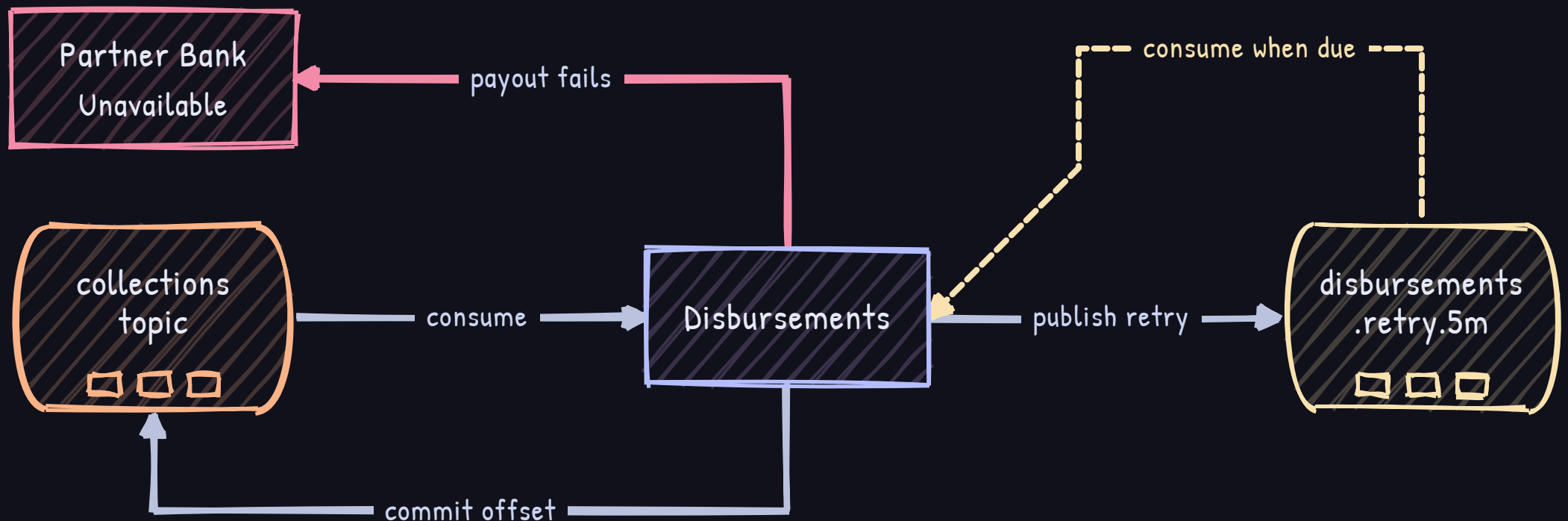
```
client, err := kgo.NewClient(
    kgo.SeedBrokers("localhost:9092"),
    kgo.ConsumerGroup("disbursements"),
    kgo.ConsumeTopics("collections"),
    kgo.DisableAutoCommit(),
    kgo.Balancers(kgo.CooperativeStickyBalancer()),
)
```


Disbursements
collections
Stop polling
pollCtx
SIGTERM
Buffered · uncommitted
Collection B
offset 43
Collection C
offset 44
Payout ✓
workCtx
Commit next: 43 ✓
Collection A
Close ✓

```go
client, err := kgo.NewClient(
    kgo.SeedBrokers("localhost:9092"),
    kgo.ConsumerGroup("disbursements"),
    kgo.ConsumeTopics("collections"),
    kgo.DisableAutoCommit(),
    kgo.BlockRebalanceOnPoll(),
    kgo.RebalanceTimeout(30*time.Second),
)
if err != nil { return err }
defer client.CloseAllowingRebalance()
```

What if the bank stays down?





1. Bank call fails

Payment unfinished.

2. Publish retry

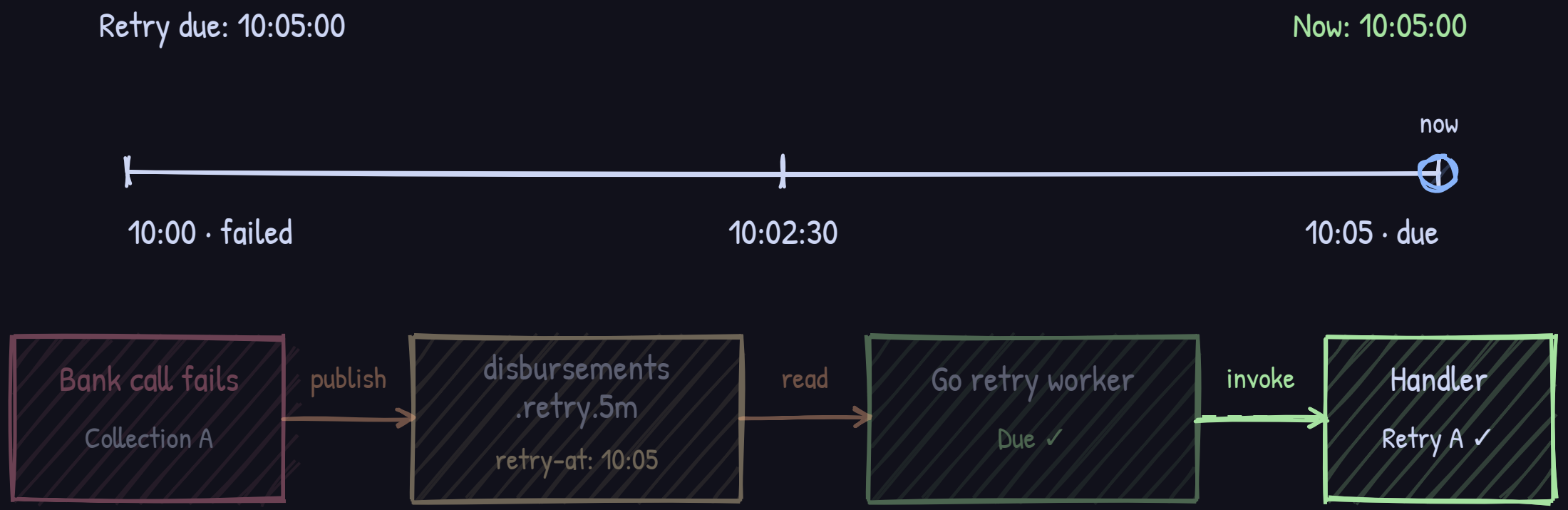
Wait for acknowledgement.

3. Commit offset

Advance the original checkpoint.

4. Retry when due

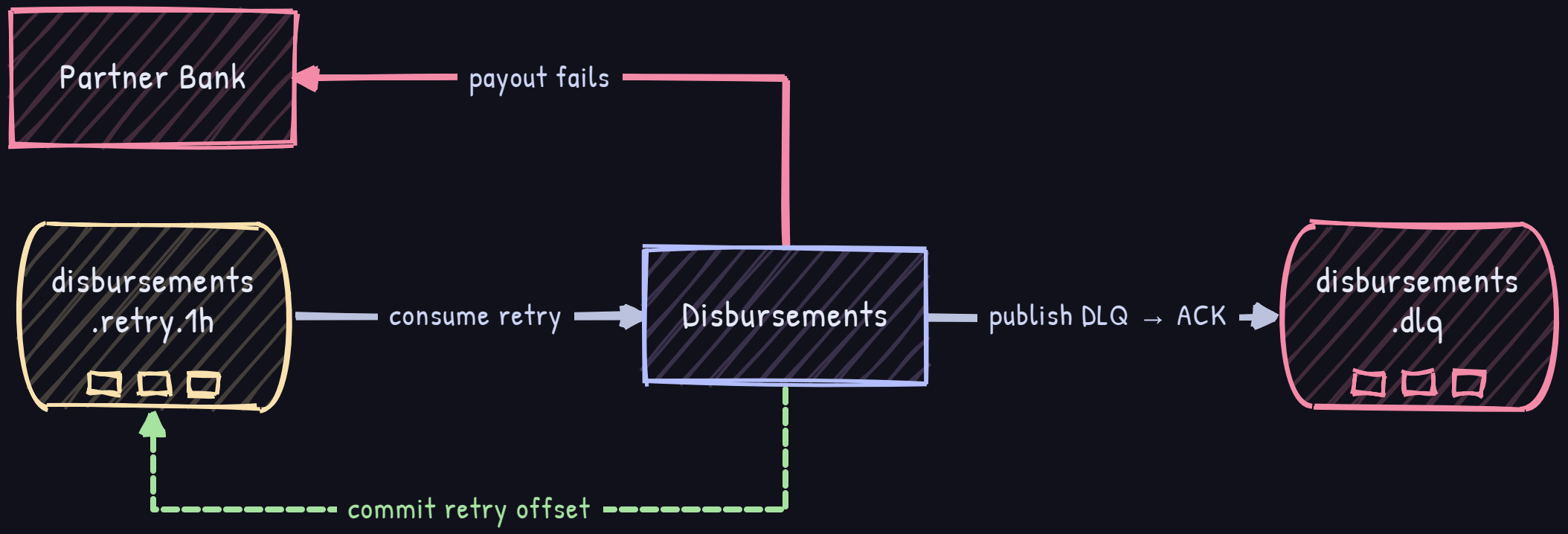
A scheduler releases due events.


Retry due: 10:05:00
Now: 10:05:00
now
10:00 · failed
10:02:30
10:05 · due
Bank call fails
Collection A
publish
disbursements
.retry.5m
retry-at: 10:05
read
Go retry worker
Due ✓
invoke
Handler
Retry A ✓


Retry · Collection A retried · all handled
Partition 1
Disbursements
Retry · 5m
Handled
Collection B
completed
offset 1
Collection C
completed
offset 2
Collection D
completed
offset 3
Collection A
completed
offset 0

What travels with a retry?

```
topic: disbursements.retry.5m
key:   collection-a
headers:
  retry-count:      "2"
  original-topic:   collections
  original-offset: "1842"
  error:            "bank timeout after 30s"
  retry-at:         "2026-12-18T10:05:00Z"
  traceparent:      "00-4bf92f3577b34da6a3ce929d0e0e4736-00f0
```


Partner Bank
payout fails
disbursements
.retry.1h
consume retry
Disbursements
publish DLQ → ACK
disbursements
.dlq
commit retry offset

DISBURSEMENTS

Dead-letter queue

disbursements.dlq

Events

3 records

Collection A — Replay queued
Partner bank timeout
7 attempts · today, 13:51

Collection B — Permanent
Invalid bank account
1 attempt · today, 13:42

Collection C — Exhausted
Partner bank timeout
7 attempts · todo

Showing 3 of 3

Collection A

evt_a17

Failure	Partner bank timeout
Event type	collection.completed
Record key	collection-a

PAYLOAD

```
{
  "event_id": "evt_a17",
  "collection_id": "collection-a",
  "status": "completed"
}
```

REPLAY DESTINATION

disbursements.retry.5m

✓ **Published to disbursements.retry.5m**
Audit #RP-1042 · alex.ops · INC-204 · awaiting processing

What about ordered retries?

Compaction & tombstones

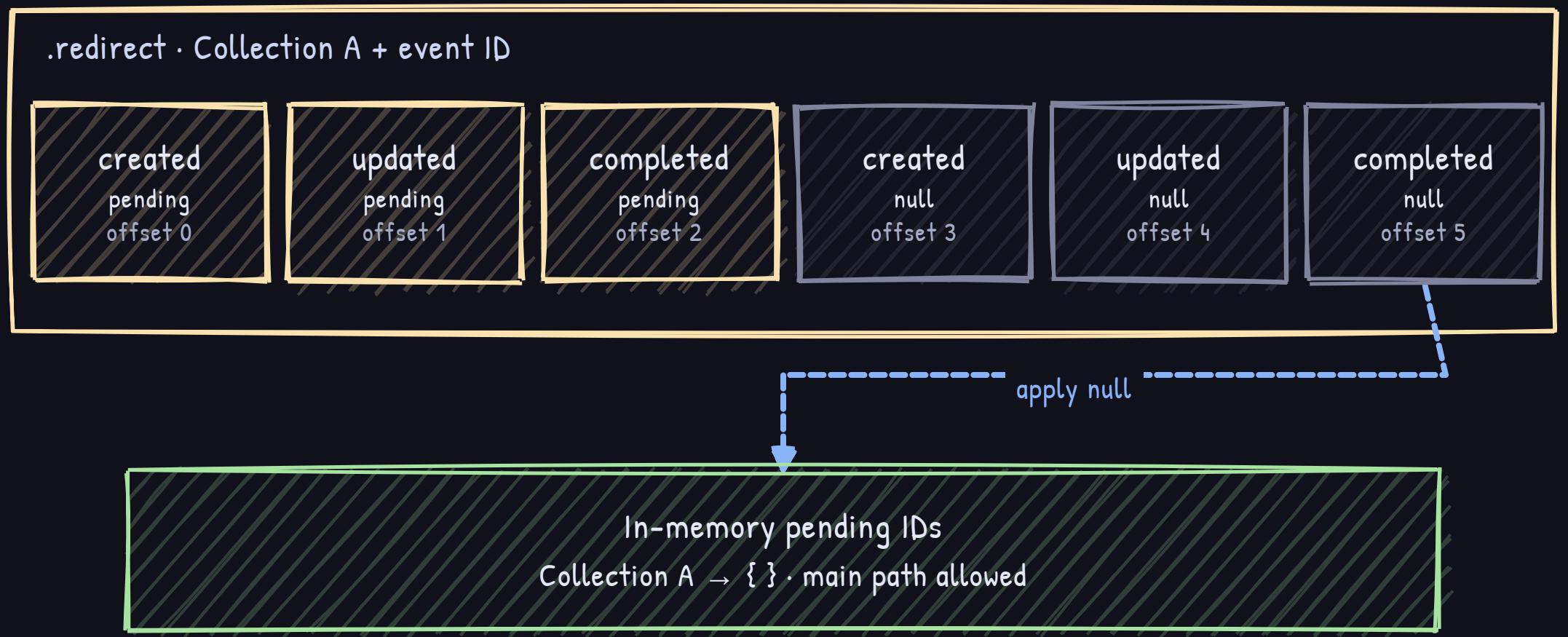





Redirect · Collection A keeps order: 0 → 1 → 3
Partition 1
Disbursements
Redirect
∅ completed
Collection A
Retry · 5m
Handled
Collection B
created
offset 2
Collection A
created
offset 0
Collection A
updated
offset 1
Collection A
completed
offset 3

Redirect state

Completed clears → release Collection A.



Need per-key order?

Redirect. Otherwise, plain retry.

What if we lose the event?



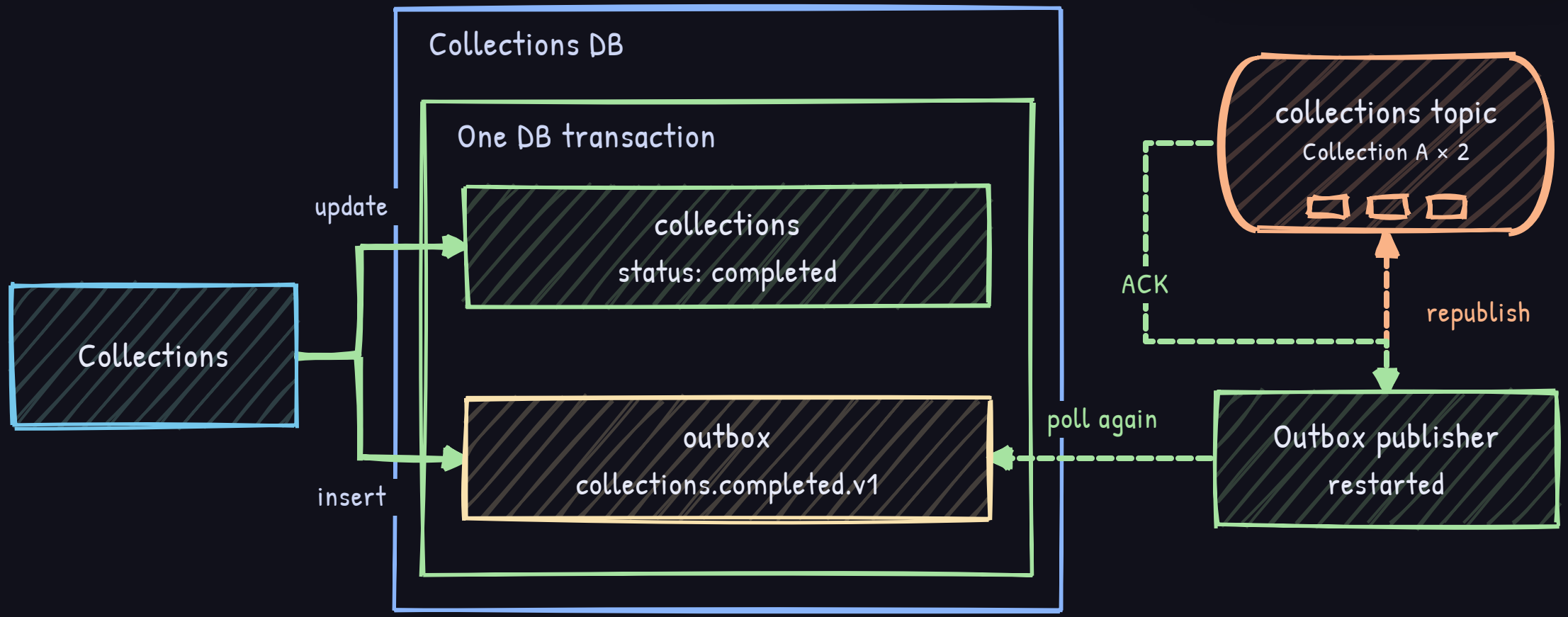
3. Publish first?

DB write fails → ghost event.

Either order breaks.

Transactional Outbox Pattern





Alternative: crash after Kafka ACK, before delete commits.

```
err := pgx.BeginFunc(ctx, db, func(tx pgx.Tx) error {
    tag, err := tx.Exec(ctx,
        `UPDATE collections SET status='completed'
         WHERE id=$1`,
        collection.ID,
    )
    _, err = tx.Exec(ctx,
        `INSERT INTO outbox (topic, key, value)
         VALUES ($1, $2, $3)`,
        "collections",
        collection.ID.String(),
        collectionCompletedJSON,
    )
}
```

```
for {
    if err := drainBatch(ctx); err != nil {
        logErrs(err)
    }

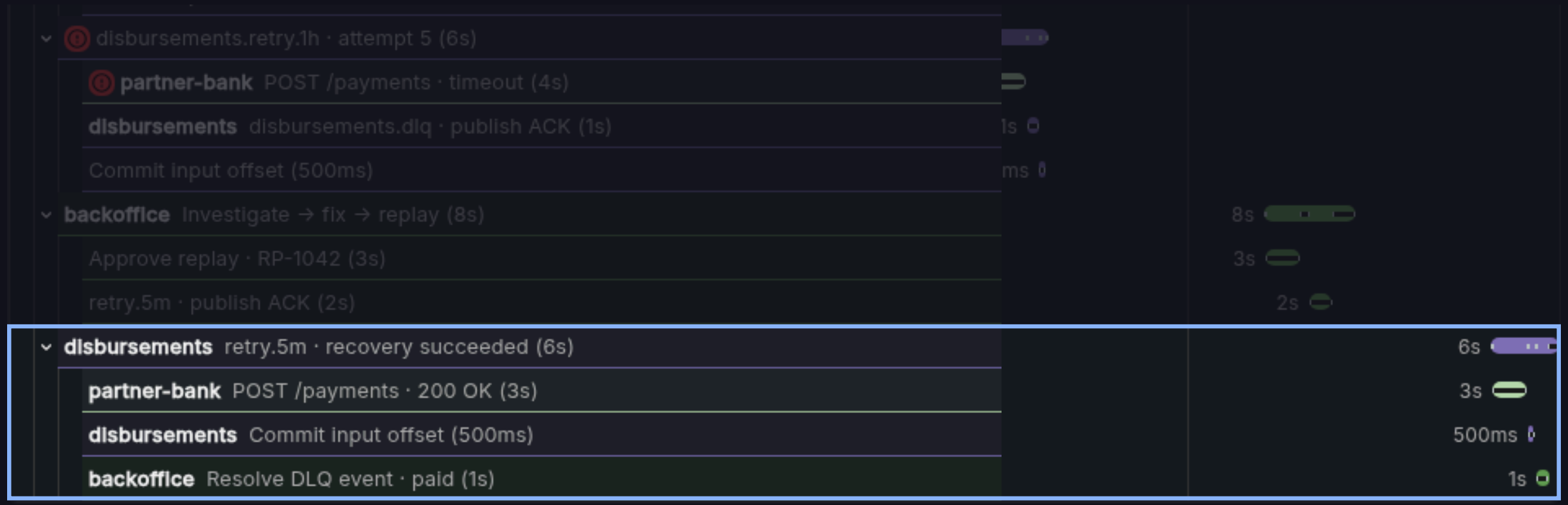
    select {
    case <-ctx.Done():
        return ctx.Err()
    case <-time.After(time.Second):
    }
}
```

Is recovery working?

Unresolved DLQ



12 unresolved events need investigation.


disbursements.retry.1h · attempt 5 (6s)
partner-bank POST /payments · timeout (4s)
disbursements disbursements.dlq · publish ACK (1s)
Commit input offset (500ms)
backoffice Investigate → fix → replay (8s)
Approve replay · RP-1042 (3s)
retry.5m · publish ACK (2s)
disbursements retry.5m · recovery succeeded (6s)
partner-bank POST /payments · 200 OK (3s)
disbursements Commit input offset (500ms)
backoffice Resolve DLQ event · paid (1s)
1s
ms
8s
3s
2s
6s
3s
500ms
1s

Inspect Kafka with `kaf`

```
$ kaf consume collections -p 0 --offset 42 -l 1
Key:         collection-a
Partition:   0
Offset:      42
Timestamp:   2026-12-18 10:05:00 +0000 UTC
{
  "collection_id": "collection-a"
}
```

Takeaways

- Idempotent consumers
- Retries and DLQ
- Order in partitions
- Monitor key metrics



Thanks!



- Slides: https://haseebmajid.dev/slides/go-labs-eda-go-kafka
- Repo: https://gitlab.com/hmajid2301/blog

Appendix

- Gently Down the Stream — Kafka, illustrated
- Error Handling Patterns in Kafka
- Several Event Types in One Kafka Topic
- Kafka 101 — Confluent
- franz-go — Go Kafka client
- kaf — Kafka CLI