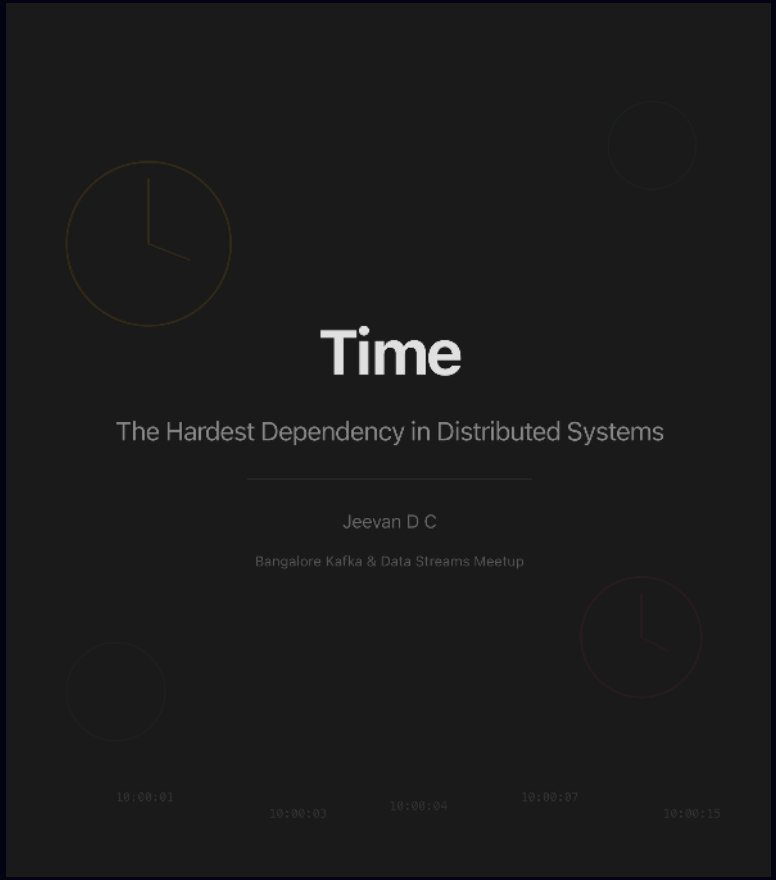
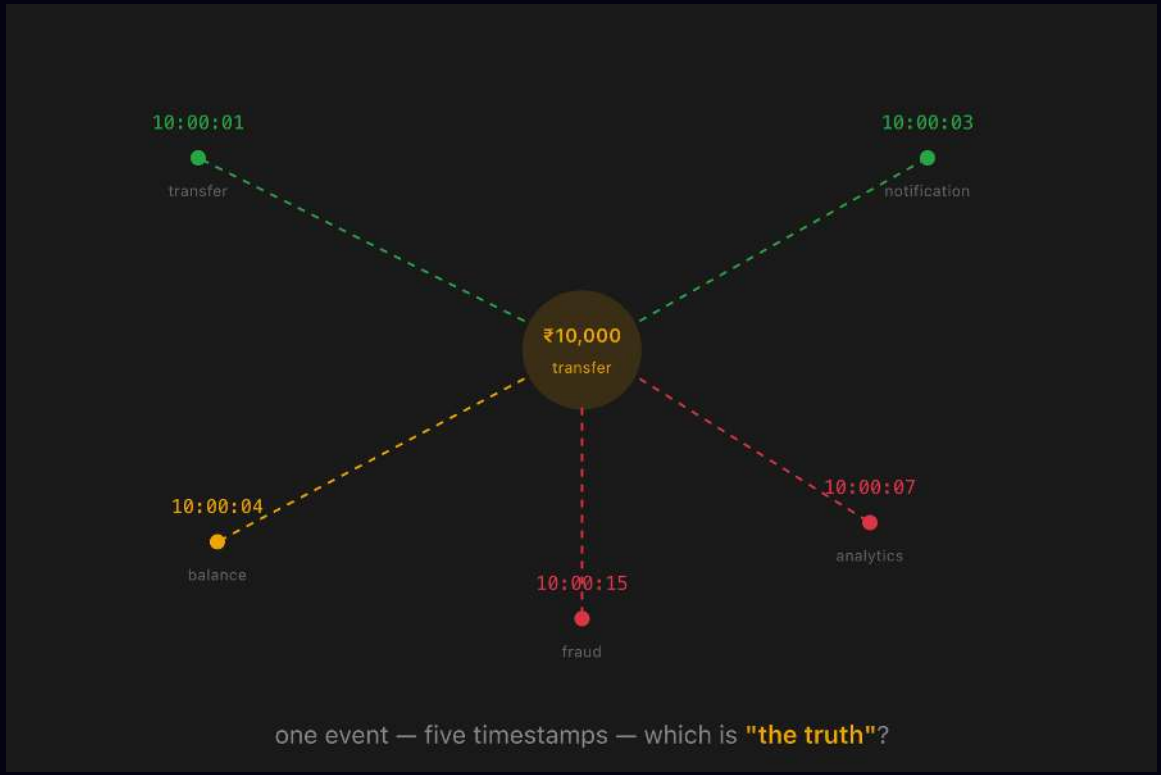

Time
The Hardest Dependency in Distributed Systems
Jeevan D C
Bangalore Kafka & Data Streams Meetup
10:00:01
10:00:03
10:00:04
10:00:07
10:00:15

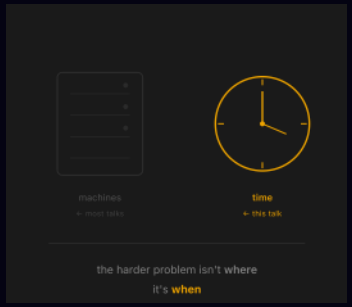

10:00:01
transfer
10:00:03
notification
₹10,000
transfer
10:00:04
balance
10:00:07
analytics
10:00:15
fraud
one event — five timestamps — which is "the truth"?

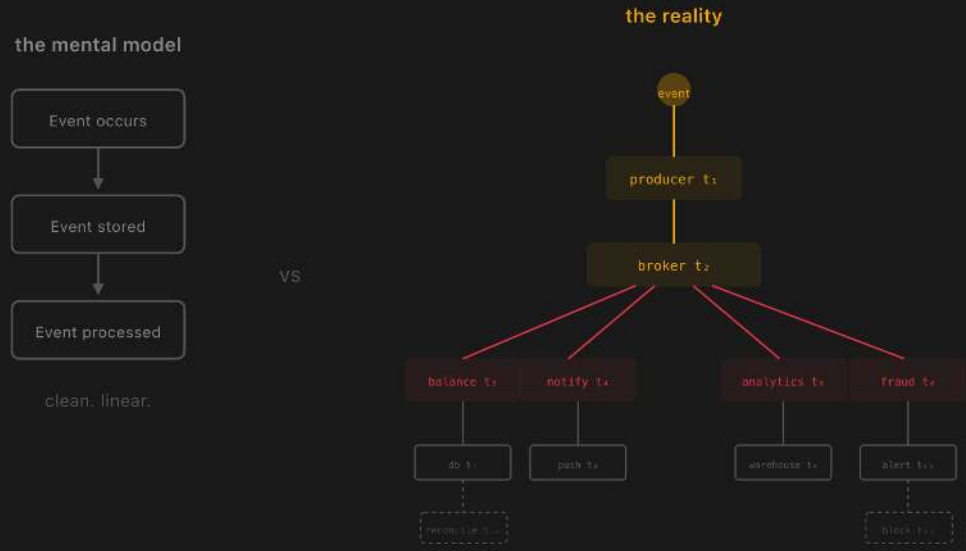
The Thesis

Distributed systems are difficult not because they are distributed across machines,

but because they are distributed across time.

A lens — not the only lens — but one that explains a surprising number of production bugs.





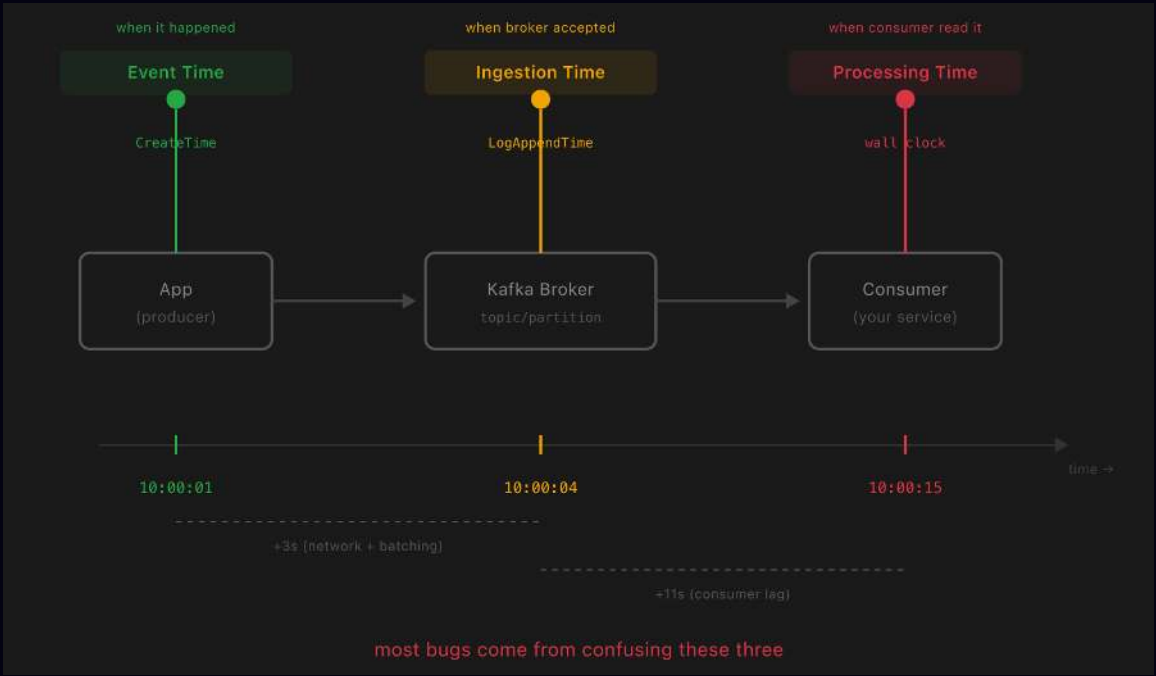
one event → many systems → many timestamps → many "truths"

This Is A Kafka Config Decision

```
# server.properties
message.timestamp.type=CreateTime    # producer's clock
message.timestamp.type=LogAppendTime # broker's clock
```

- The choice determines which "time" downstream consumers see
- Most teams never consciously choose — they inherit a default

Three Kinds of Time




PART 1
Time Is Scary
There is no "now."

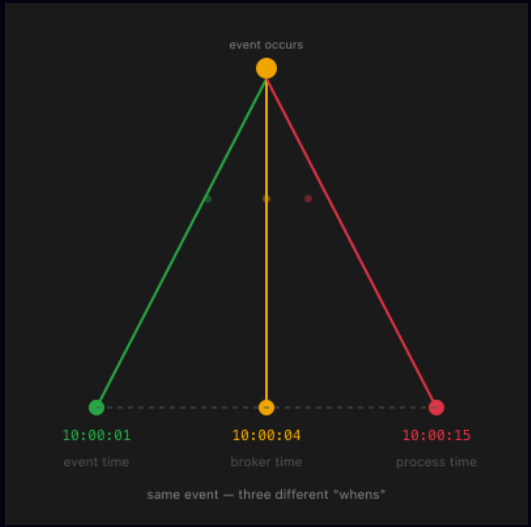
There Is No "Now"

- Time is not a fact
- Time is an agreement
- In distributed systems, there's no one to agree with

Our brains experience a ~3 second window and call it "now."

We built systems that assume the same thing.

There's no window in a distributed system.



Clock Drift

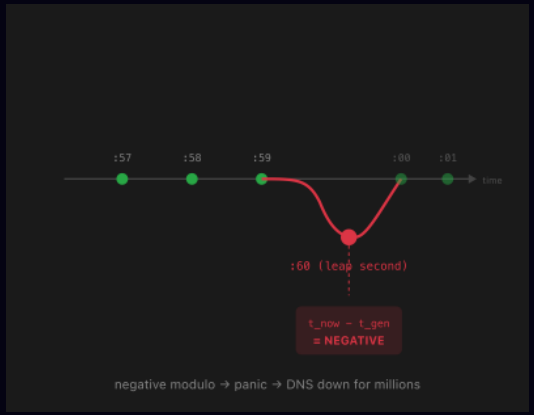
Why clocks lie: NTP drift, leap seconds, VM pauses, container migration

> Time is the only dependency that every service shares but nobody owns.

Cloudflare 2017: A Pure Time Bug

- RRDNS calculates: `t_now - t_generated`
- Leap second makes `t_now` go backward
- Result goes negative → modulo → panic
- DNS errors spike across Cloudflare's edge

The code assumed time moves forward. It doesn't always.



Google & CockroachDB Took This Seriously

Google Spanner:

- GPS + atomic clocks in every data center
- Exposes time as an interval, not a point:

```
TrueTime.now() → [earliest, latest]
```

CockroachDB (Hybrid Logical Clocks):

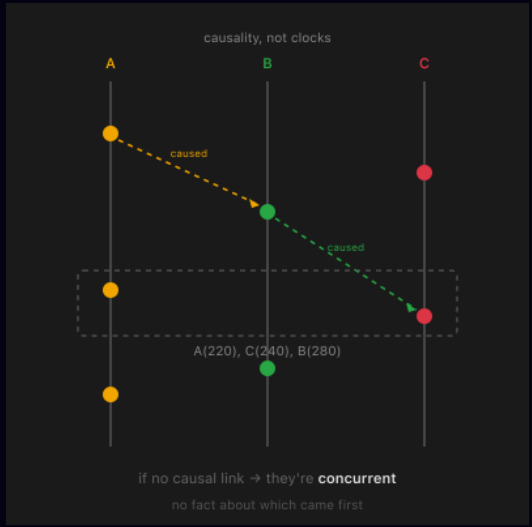
- No custom hardware
- Wall clock + logical counter
- Different approach: assumes a clock skew bound, restarts transactions that violate it

Lamport (1978): We Don't Need Real Time

We don't need clocks to establish ordering. We need causality.

- If A could have caused B → A "happened before" B
- If neither could have caused the other → they are concurrent
- No clock needed

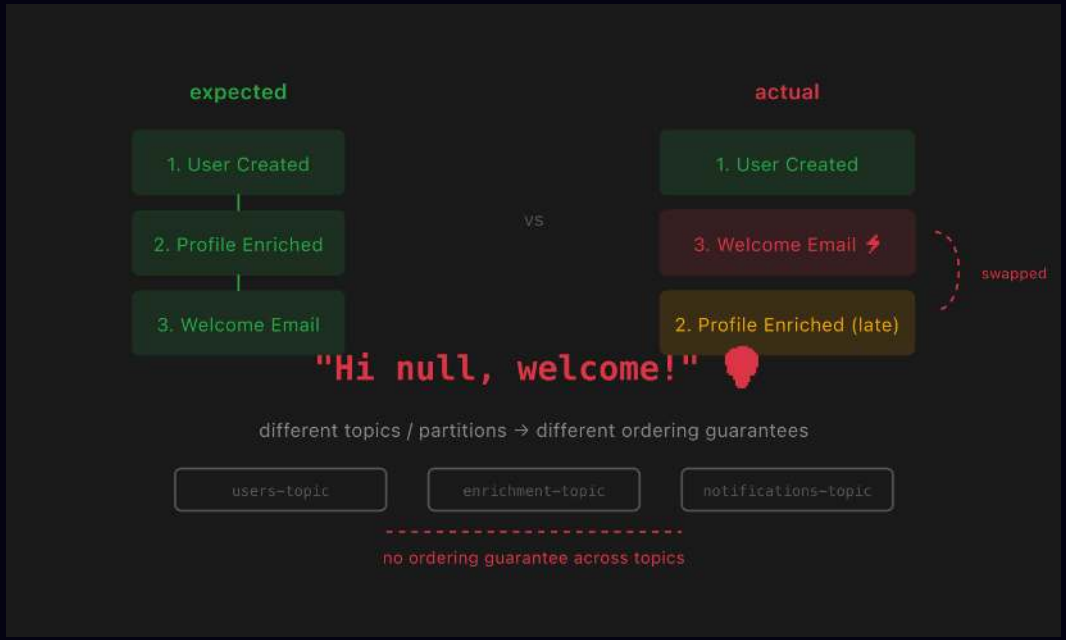
Most events in a system are concurrent. We force ordering where none naturally exists — and pay for it.



Ordering Is An Illusion

Kafka gives ordering within a partition.

Business workflows span multiple topics.



The notification consumer saw `User Created` and fired immediately — enrichment hadn't arrived yet.

> Ordering is not a property of the world. Ordering is a property of an observer.


PART 2
Time Is Expensive
Every guarantee has a price.

The Coordination Chain

- How do we guarantee ordering? →

Coordination

- How do we guarantee coordination? →

Waiting

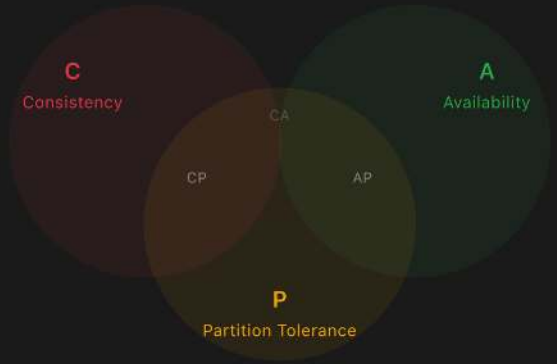
- How do we guarantee waiting? →

Latency

Every ordering guarantee has a price. Are you sure the business needs it?

CAP

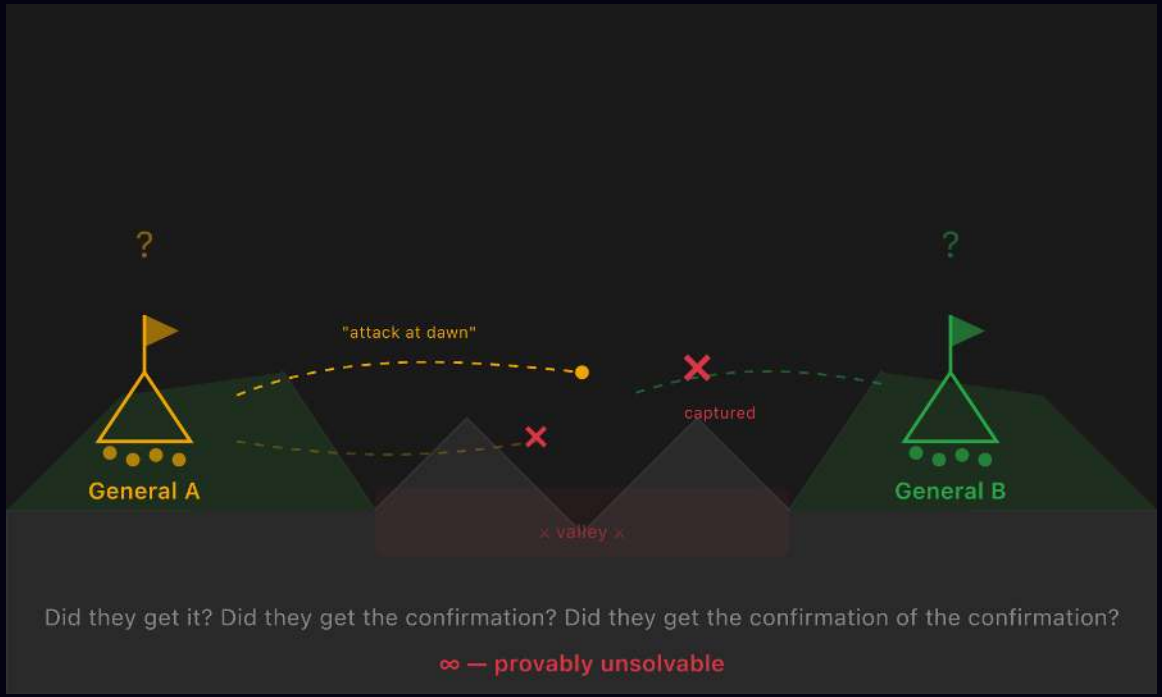
pick two (during a partition)



PACELC

even with NO failure, you still trade:

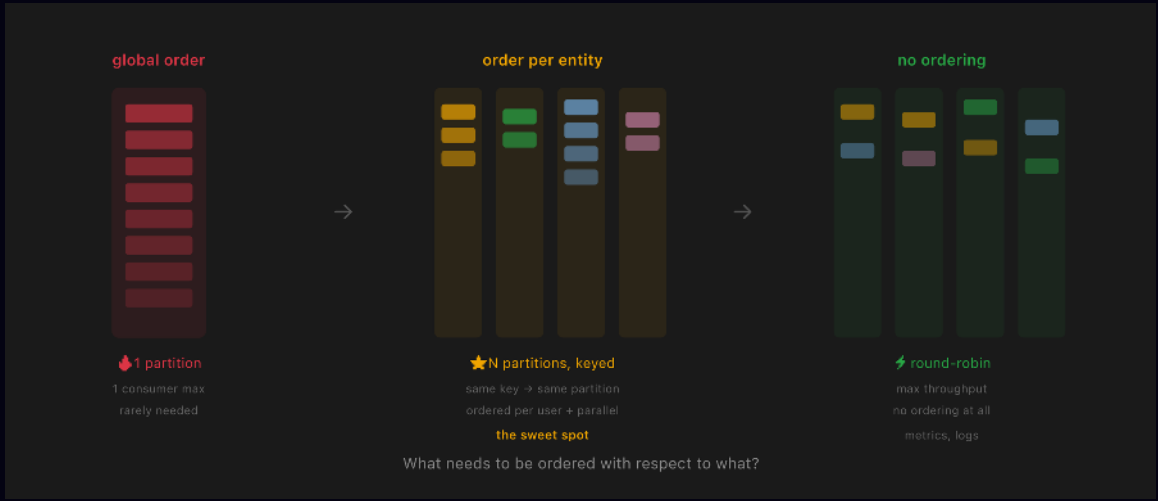
your coordination chain is PACELC — **always trading**



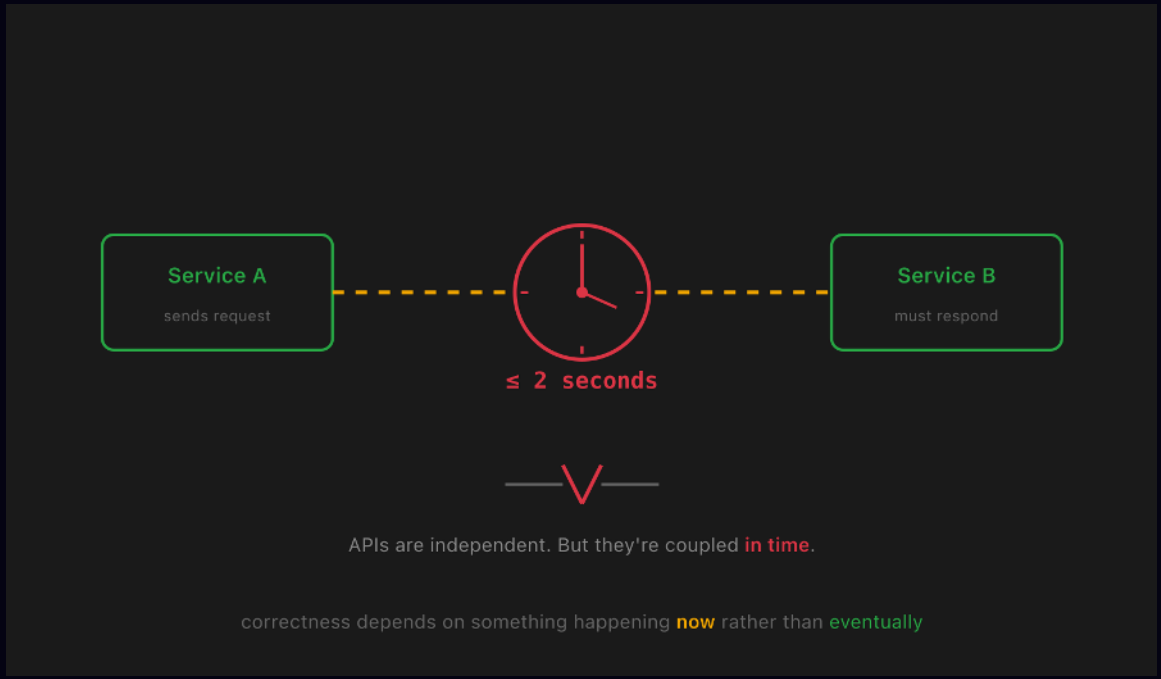
Provably unsolvable.

Coordination over an unreliable channel cannot be guaranteed.

The Partition Key Question

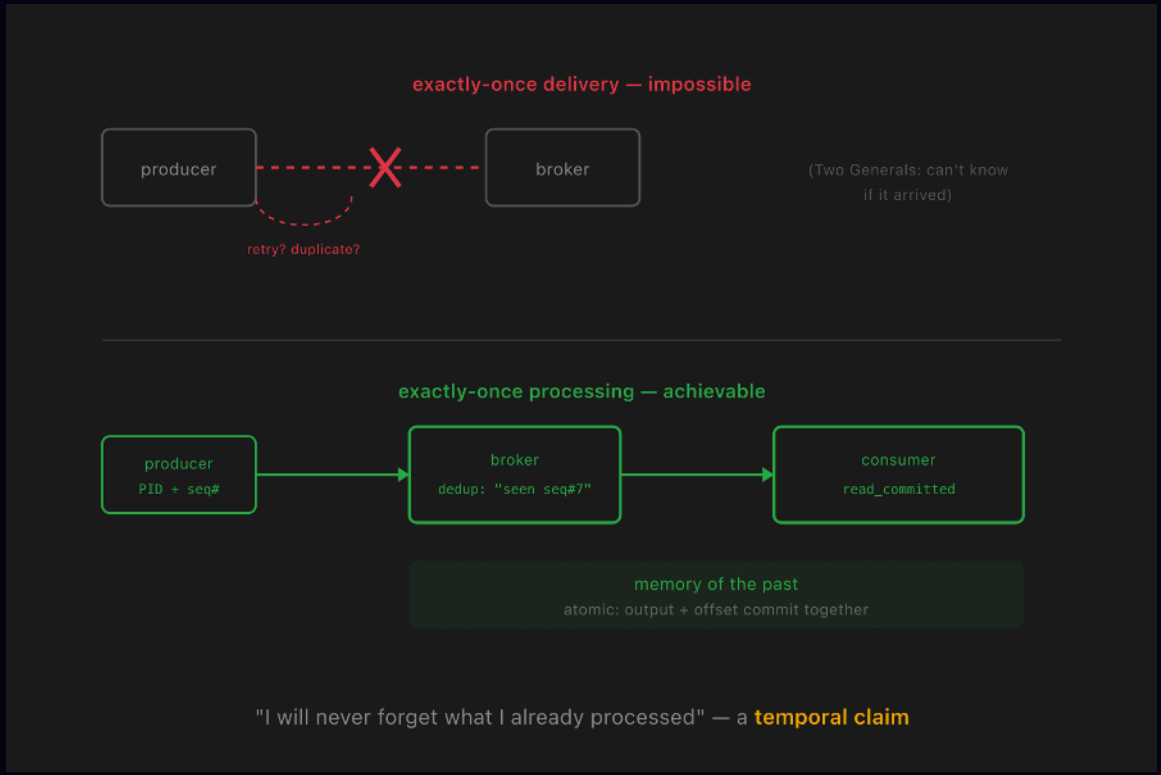


Ask: What actually needs to be ordered with respect to what?

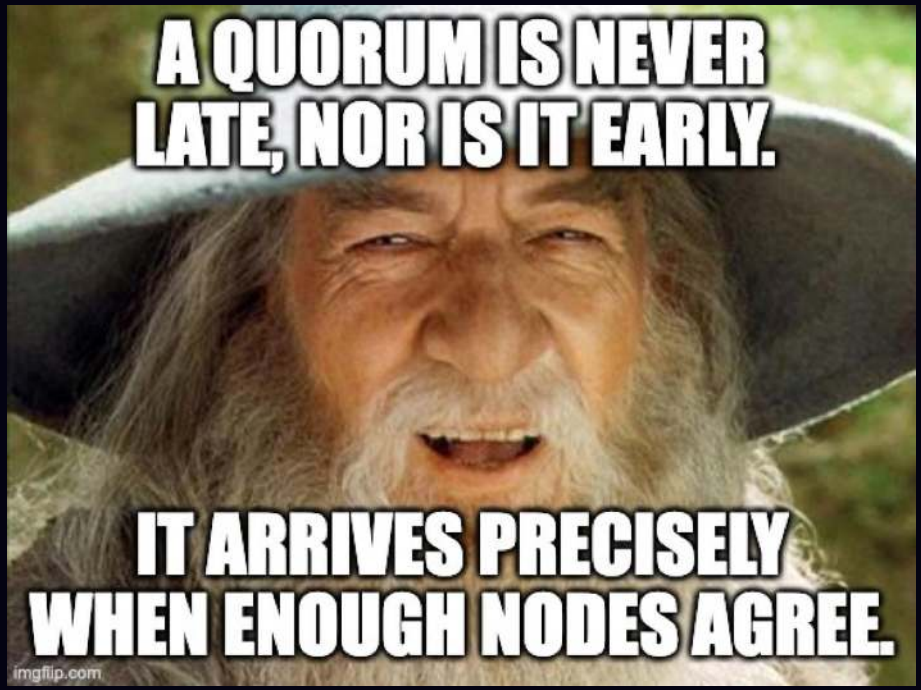


Why do services become fragile? Common answers: network, code quality, dependencies.

Often the real answer: time.



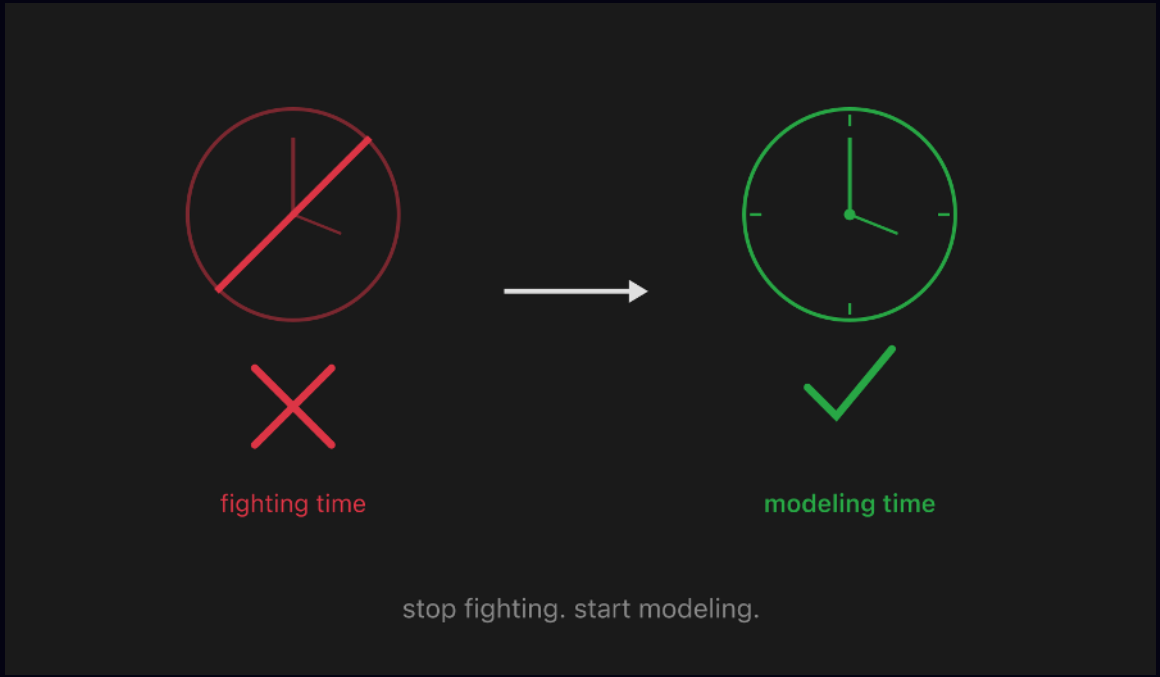
⚠️ Boundary: EOS holds inside Kafka (consume → process → produce). Any external side effect (DB write, API call, email) means at-least-once + idempotency.


A QUORUM IS NEVER
LATE, NOR IS IT EARLY.
IT ARRIVES PRECISELY
WHEN ENOUGH NODES AGREE.
imgflip.com

The Pivot

We stop fighting time.

We start modeling it.




PART 3
Time Is A Superpower
Model it. Query it. Replay it.

How To Model Time (Lightest → Heaviest)

1. Embed `event_time` in every message — choose `CreateTime` consciously
2. Design for late arrivals — `grace()` periods, out-of-order windows
3. Decouple time expectations — async over sync, idempotency keys
4. Make time observable — alert on temporal lag (seconds behind, not messages behind)
5. Make time queryable — replay, event sourcing, temporal state

```
// Strict: late events are dropped
TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(5))

// Tolerant: accept events up to 1 min late
TimeWindows.ofSizeAndGrace(
    Duration.ofMinutes(5),
    Duration.ofMinutes(1)
)
```

- This is modeling time
- This is an explicit temporal contract
- Not "watermarks" — that's Flink/Beam. Kafka Streams uses stream time + grace()

Traditional vs Event-Driven

Traditional:

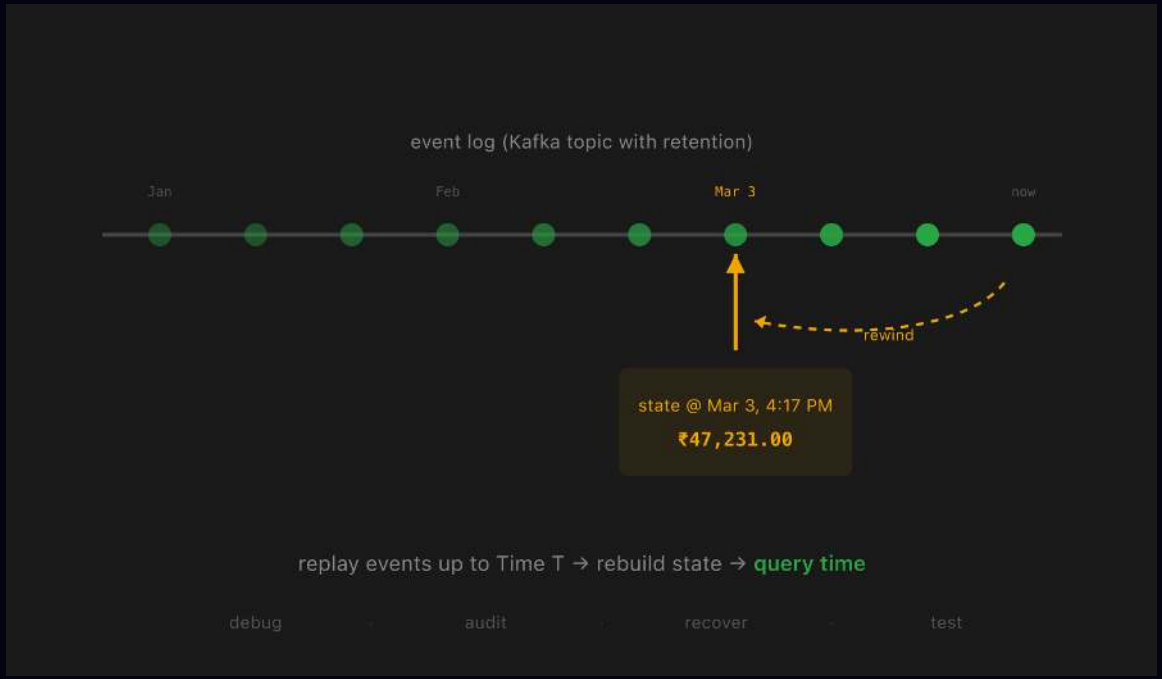
```
Current State
(that's all there is)
```

Event-driven:

```
State = f(all events up to Time T)
```

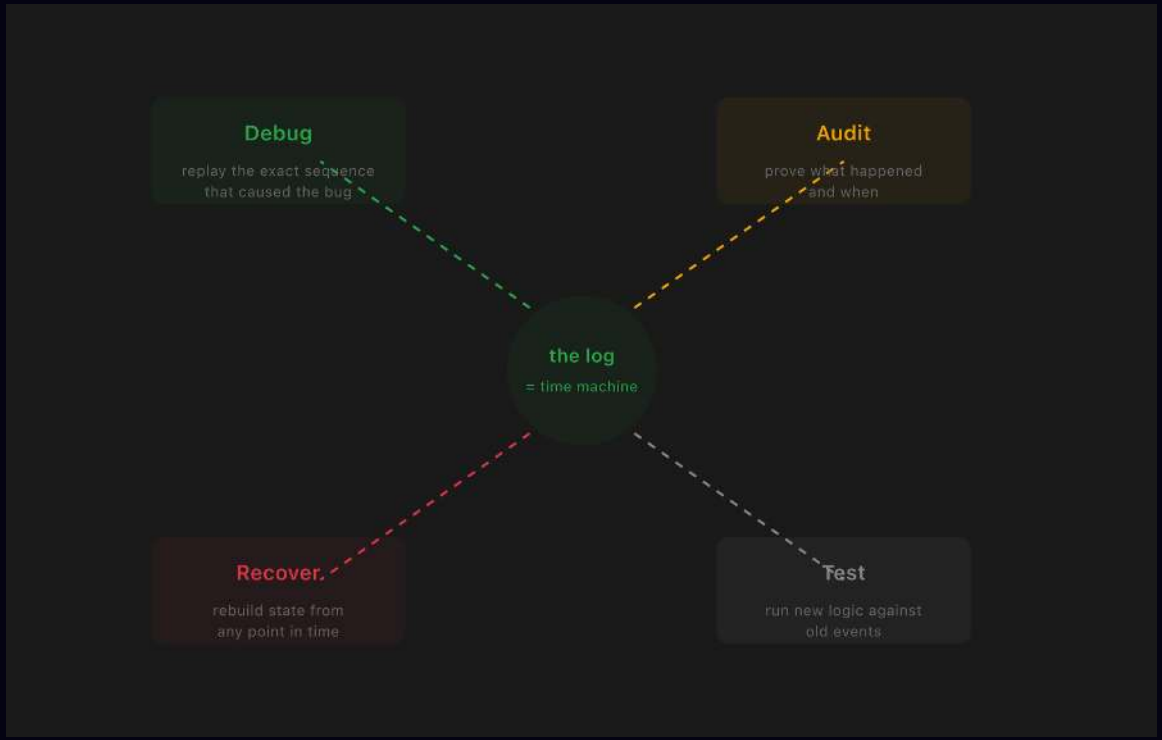
Time becomes queryable.

Replayability Changes Everything



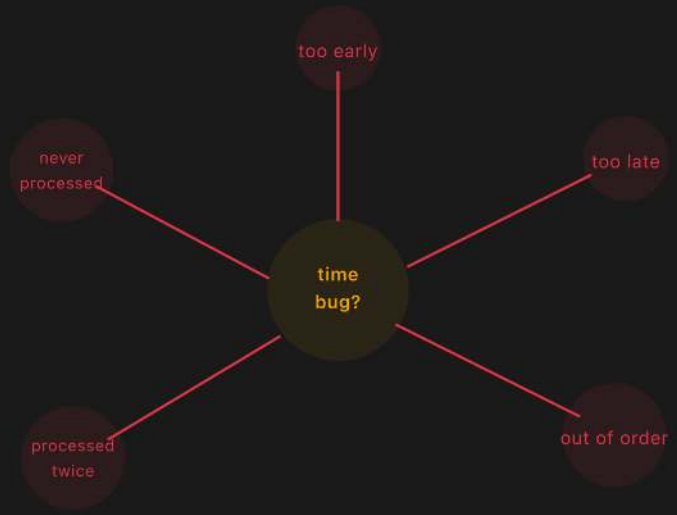
That's not reading a database — that's querying time.

What Replay Enables



next incident → ask:

is this a time bug?



surprisingly often, it is.

Three Things To Remember

1. Time is not a fact — it's an agreement. And in distributed systems, there's no one to agree with.

2. Every ordering guarantee is a latency bill. Ask what actually needs to be ordered.

3. For every event contract, ask: "What happens if this arrives 30 minutes late?"

By The Way — AI Agents Have The Same Problem

- Agent context windows are stale reads — there is no "now" for an LLM either
- Multi-agent coordination is temporal coupling — same Two Generals, new costume
- Tool calls are external side effects — back to at-least-once + idempotency

> If you're building agents that coordinate over Kafka... congratulations, you now have all of these problems plus non-determinism.

Thank You

Questions?

jeevan.dc24@alumni.iimb.ac.in

https://noobj.me