Challenges of Distributed Systems

Grzegorz Piwowarek

@pivovarit

{ 4comprehension.com }

WarsawJUG | Vavr | Oracle ACE

Independent Consultant/Trainer

distributed systems | microservices | async | reactive | java

@pivovarit

What makes systems...distributed?

CAP Theorem

- **C**onsistency
- **A**vailability
- **P**artition tolerance

Consistency

every read receives the most recent write

(a single-copy illusion)

Availability

every request receives a non-error response

(even if some nodes are down)

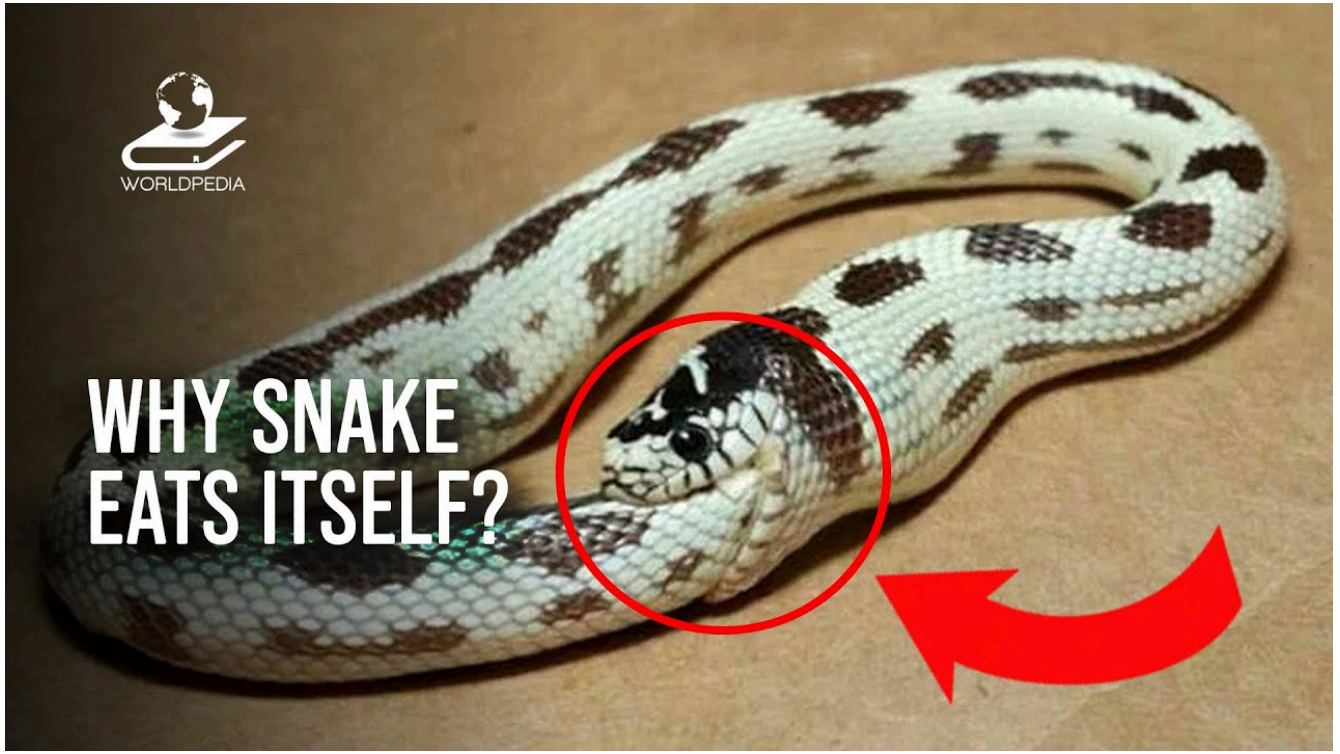
Partition tolerance

system continues to operate even if network links are lost

distributed systems are systems that communicate over unreliable channels

Rule: In the presence of a network partition, a distributed system must choose either *Consistency* or *Availability*.

...


WORLDPEDIA
WHY SNAKE
EATS ITSELF?

Monolith

Microservices

https://twitter.com/ddprrt/status/1425418538257428488

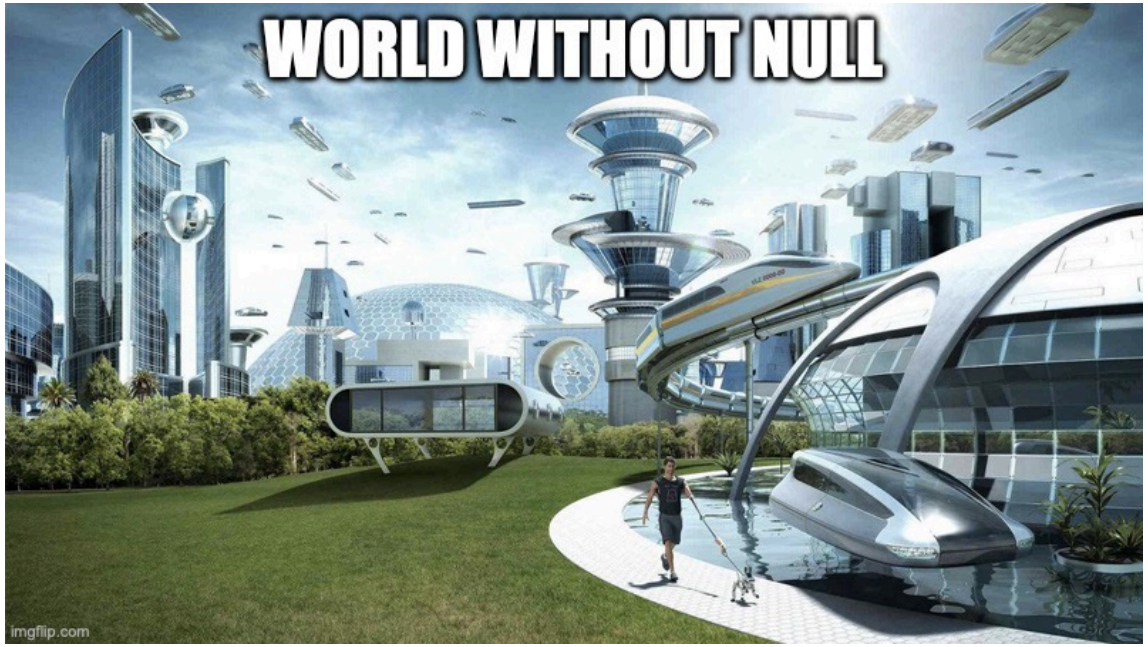
Monolith

Microservices

https://twitter.com/ddprrt/status/1425418538257428488

"It was the invention of the null reference in 1965...I call it my billion-dollar mistake."

Tony Hoare


WORLD WITHOUT NULL
imgflip.com

Another *"billion-dollar mistake":*

Microservices

Another *"billion-dollar mistake":*

"Micro"

Microservice vs service?

How small is ”micro”?

N lines of code?

N endpoints?

N classes?

N MBs?

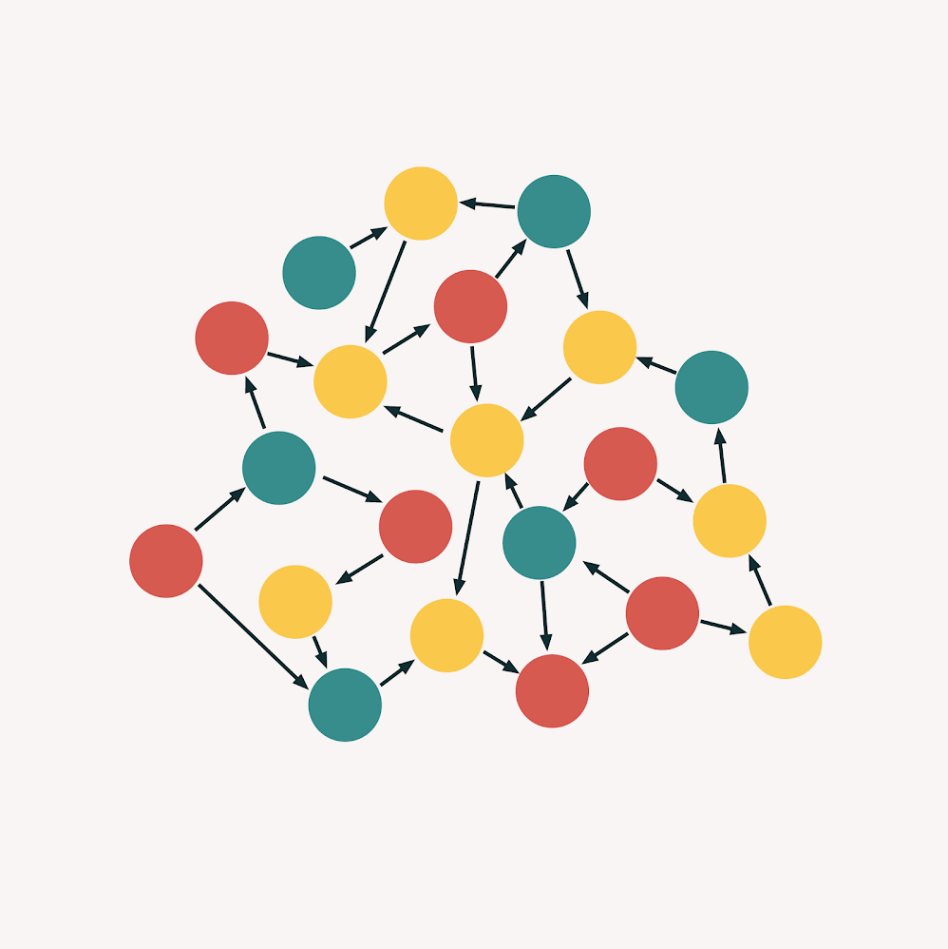
N responsibilities?

Rewritable in X time?

Microservices: the Main Idea

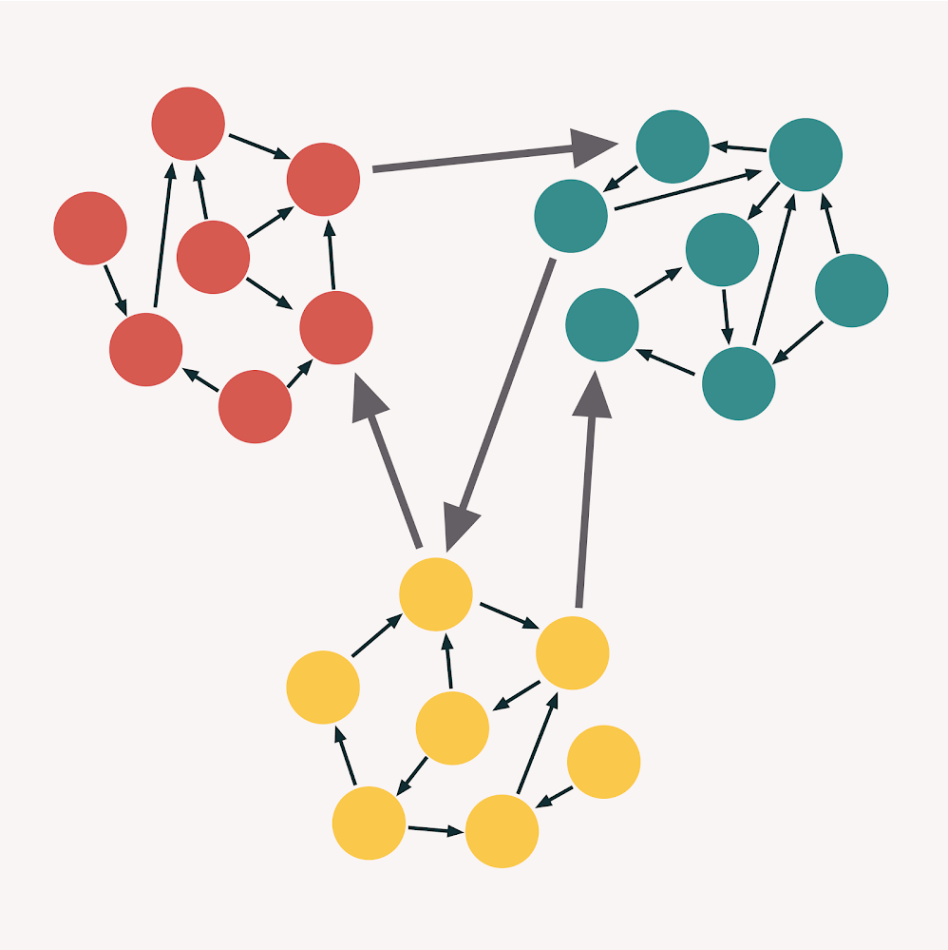
Enable scalability through independence and modularity

Tight coupling - Low cohesion



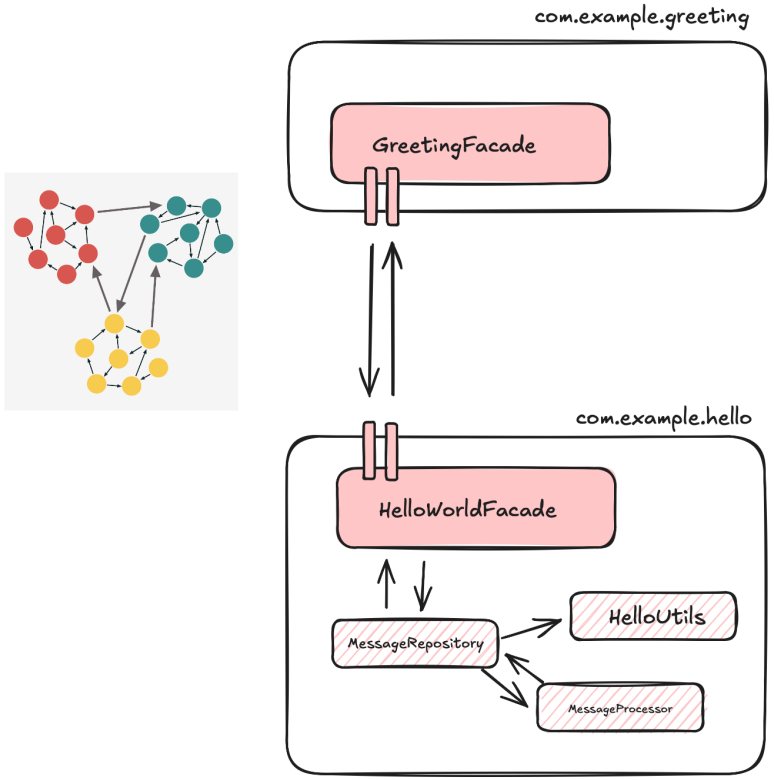
source: https://enterprisecraftsmanship.com/posts/cohesion-coupling-difference/

Low coupling - High cohesion

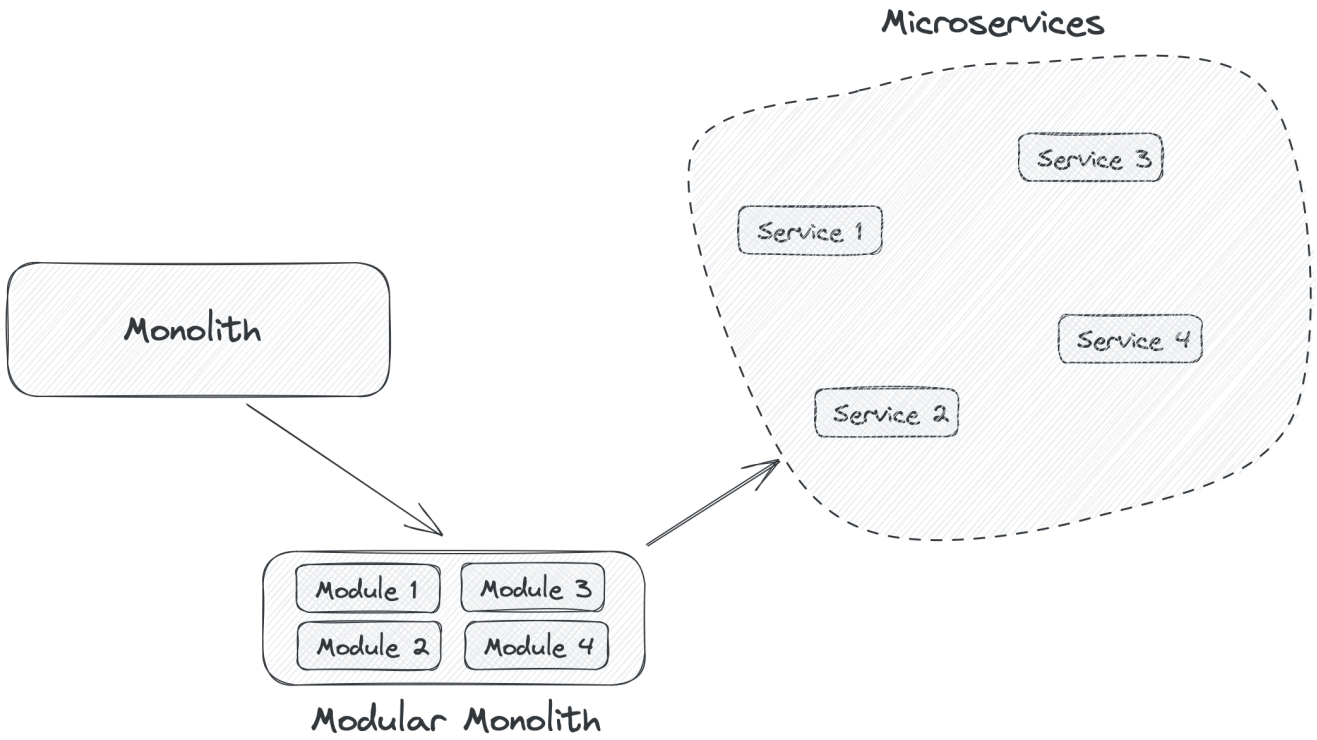


source: https://enterprisecraftsmanship.com/posts/cohesion-coupling-difference/

You can have modularity without microservices



Naturally, you don't get all the benefits, but it's not a demanding investment


Microservices
Service 3
Service 1
Service 4
Service 2
Monolith
Module 1
Module 3
Module 2
Module 4
Modular Monolith

"The smaller the service, the more you maximize the benefits and downsides of microservice architecture."

Sam Newman

Netflix & Micro-Services

author: Bruce Wong

Netflix: 2000 engineers

Applying the high scalability tips from Google and Netflix software architectures to your trivial project.

However, don't be fooled by the size of those microservices, because a lot of those so-called microservices at Netflix are a lot larger, just looking at the code base, than the big monoliths that I've worked at, at many other companies.

Paul Bakker

source: https://www.infoq.com/presentations/netflix-java/

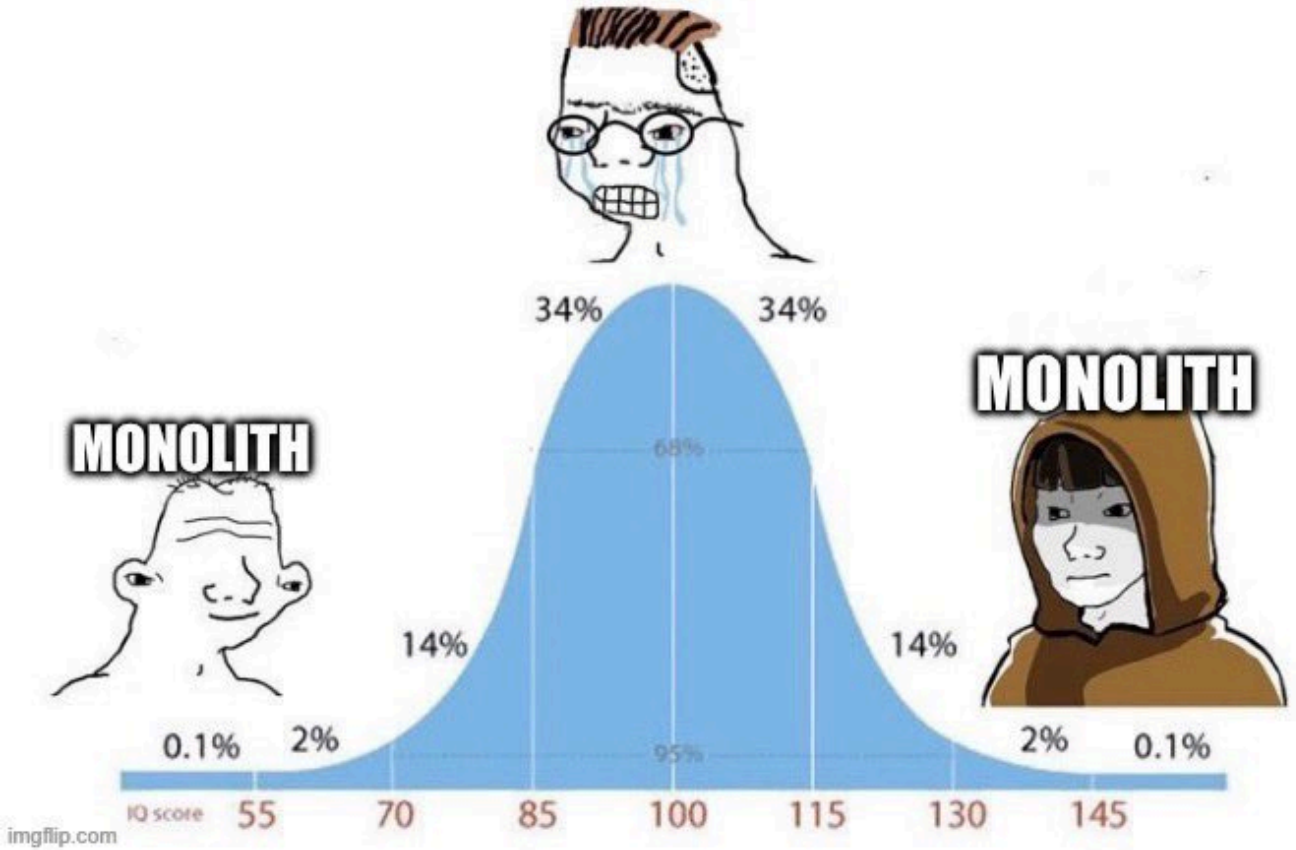
Do not ask about the max size, but when to split

So... when do we split?

When it hurts too much

The #1 rule of distributed systems: don't do it until you have to

EVENT-DRIVEN MICROSERVICES



The Pragmatic Default

https://twitter.com/dhh/status/1247522358908215296

Our server app is a monolith, one big codebase of several million lines and a few thousand Django endpoints [1], all loaded up and served together. A few services have been split out of the monolith, but we don't have any plans to aggressively break it up.

https://instagram-engineering.com/static-analysis-at-scale-an-instagram-story-8f498ab71a0c

Reliability in an unreliable world

When sending a message over unreliable channels, how to guarantee delivery?

(...thought experiment...)

Definition

An operation is **idempotent** if performing it multiple times produces the *same effect* as performing it once.

```
f(f(x)) = f(x)
```

Examples

- Setting a user's status to "active" → ✅
- Incrementing a counter → ❌

In a distributed system

Failures and retries are inevitable.

- Network retries may cause duplicate requests
- Clients or load balancers might resend operations
- Idempotency prevents unwanted side effects

Exactly-once delivery?

A fairy tale.

Reality: You only ever get:

- **at-most-once** → messages may be lost
- **at-least-once** → messages may be duplicated

Exactly-once doesn't exist

- Networks can fail between *send* and *acknowledge*
- Clients can retry
- Servers can crash mid-processing

Exactly-once *effect*

We can’t guarantee *exactly-once delivery*,

but we can design for **exactly-once effect** using idempotency.

```
at-least-once delivery + idempotency = exactly-once
```

PACELC Theorem

Daniel Abadi extended CAP to include trade-offs *when no partition occurs*:

***If P** (partition) → A or C*
***Else** (ok) → Latency or Consistency*

Even without partitions, there's another trade-off

Waiting for consensus takes time

This is the "ELC" part of PACELC.

Imagine two coffee shops sharing an order system

- network failure → choice: *keep taking orders (A)* or *pause until synced (C)*
- no failure → each order can be: *fast but maybe outdated (L)* or *slow but always accurate (C)*

Real-world examples

- **Amazon Dynamo / Cassandra** → prioritize *Availability + Low Latency*
- **Google Spanner** → prioritize *Consistency*, accept more latency

temporal coupling

when all your services need to responsive at the same time

False Dichotomy:

Consistent vs. Inconsistent

Different consistency levels

- **Strong consistency** → Data converges immediately
- **Eventual consistency** → Data converges... eventually
- **Accidental consistency** → Data converges... maybe

All are “consistent” — just in different ways.

Eventual consistency

*If no new updates occur, all replicas will **eventually** converge to the same state.*

It's a trade-off: we get **availability and speed** at the cost of temporary disagreement.

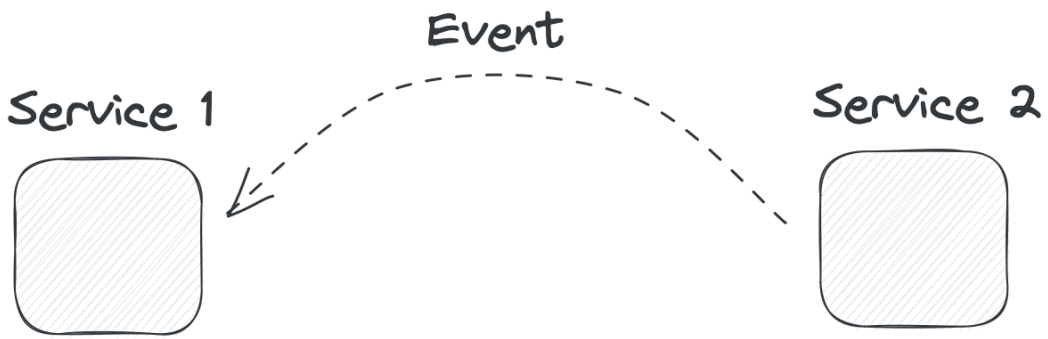
Why this makes sense

- Networks are slow and unreliable
- Waiting for everyone to agree slows everything down
- So we let nodes respond now and sync later

→ **Fast now, consistent later**

synchronous communication

asynchronous communication



From HTTP calls...

```
OrderService -> PaymentService -> NotificationService
```

Each service calls another directly...

...tight coupling, dependencies, and failure chains.

...to Events

```
OrderService --> publishes OrderCreatedEvent
PaymentService --> consumes OrderCreatedEvent
NotificationService --> consumes PaymentConfirmedEvent
```

Services react to **events** instead of making direct calls.

Local Read Models

Each service maintains its own **local view** of data it needs.

```
@EventListener
void on(PaymentConfirmedEvent event) {
    orderReadModel.updateStatus(event.orderId(), "PAID");
}
```

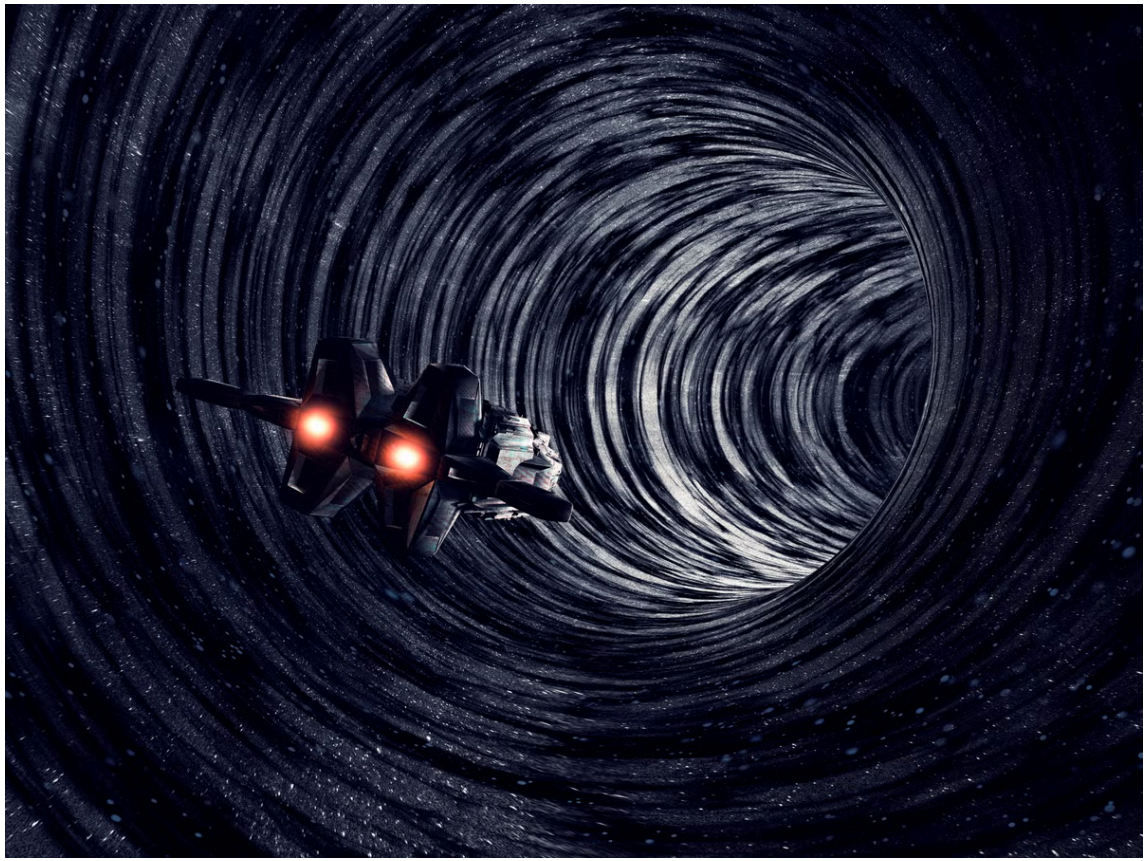
This enables fast, local reads - no cross-service queries needed

...but it eventually consistent

Why Local Read Models?

- No need to call other services for data
- Improves reliability - service can operate even if others are down

Time



```
var first = Instant.now();
var second = Instant.now();
```

first <= second?

A **monotonic clock** always moves forward - never backward.

But most `system clocks` are **not monotonic**.

Why clocks go backwards

- **NTP** (Network Time Protocol) adjustment
- Virtual machines paused and resumed
- Leap seconds
- Manual time correction by an admin

Monotonic clocks to the rescue

- In Java: `System.nanoTime()`
- In Linux: `CLOCK_MONOTONIC`

They never go backward, but don't represent "real" time.

Measuring time: from Java to kernel and back

In distributed systems, time is… **an illusion**.

The problem with clocks

- Each machine has its own clock
- Clocks drift - even if synced
- Network delays make “now” ambiguous

So: there is no single, global “current time.”

Example

Two servers record an event:

- Server A: event at 12:00:00.100
- Server B: event at 12:00:00.090

Which happened first? 🤔

Why this matters

- Event ordering affects state changes
- Conflicts appear when we can’t tell “what came first”
- Replication, logs, and causality all depend on time

Happened-before relationship

Instead of wall-clock time, we use **causal order**:

$A \rightarrow B$ *if A happened before B (causally)*

We care about *ordering* of events, not their timestamps.

Logical clocks

- **Lamport clocks** → simple counters to track causal order
- **Vector clocks** → richer structure to detect concurrent events

They don’t measure real time - they measure **cause and effect**.

Real time vs. logical time

- **Real time** → what your watch shows
- **Logical time** → what the system can *prove* happened first

Distributed systems live in **logical time**.

Google’s Spanner uses special hardware clocks (TrueTime API) ⏱️

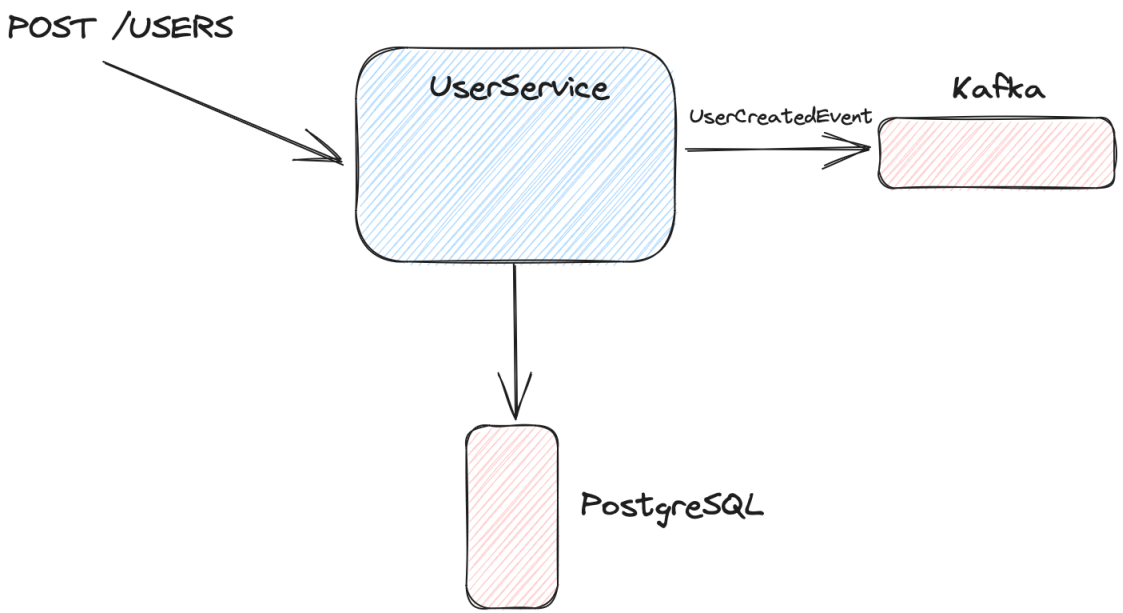
But even then, they include an **uncertainty window** - because perfect time doesn’t exist.

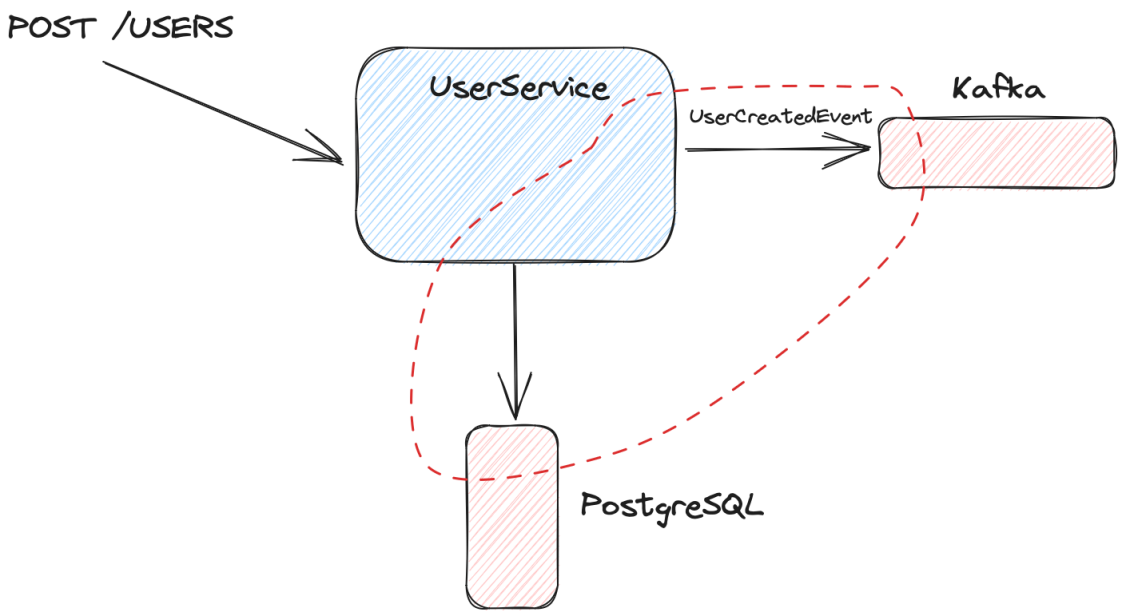
Summary

- Each node has its own imperfect sense of time
- We can’t rely on timestamps for ordering
- We use **logical or causal time** to reason safely

Time is relative - especially in distributed systems 

Eventual Consistency != Accidental Consistency

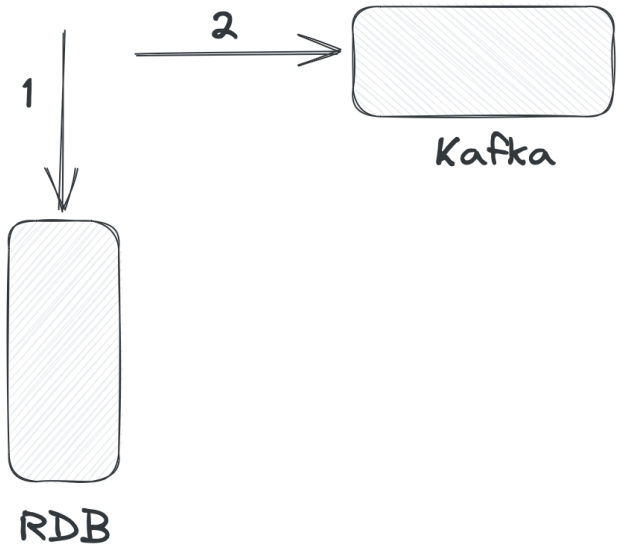

POST /USERS
UserService
UserCreatedEvent
Kafka
PostgreSQL


POST /USERS
UserService
UserCreatedEvent
Kafka
PostgreSQL

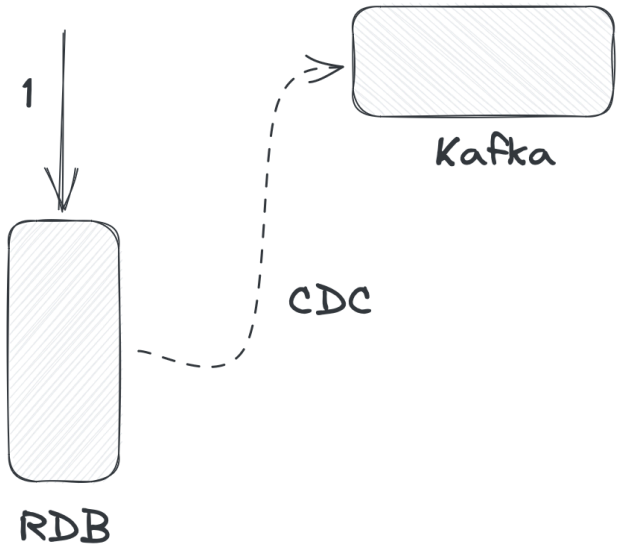
```
@Transactional
public void createUser(CreateUserCommand command) {
    var user = from(command);

    persist(user);                          // 1
    send(UserCreatedEvent.from(user)); // 2
}
```

`@Transactional` won't save you in the distributed world

Dual-Write (distributed transaction)



Singe-Write with async propagation



Transactional Outbox Pattern

Change Data Capture

Transactional Outbox Pattern

Write the event to a local **outbox table** as part of the same transaction.

```
@Transactional
public void createOrder(Order order) {
    orderRepository.save(order);
    outboxRepository.save(new OutboxEvent("OrderCreated", order.i
}
```

Then a separate process reads the outbox and publishes the events

Trivial Publisher

```
while (true) {
    for (OutboxEvent event : outboxRepository.findUnpublished())
        eventBus.publish(event.toDomainEvent());
        event.markAsPublished();
    }
}
```

Reliable, asynchronous, and decoupled event publishing.

Change Data Capture (CDC)

Instead of polling the outbox table manually...

We can let the database **stream its own changes**.

CDC in Action

- Tools like **Debezium** rely on db replication protocols
- Each committed change becomes an event
- No need to modify application logic

```
-- Database change -->
INSERT INTO outbox (...) VALUES (...)

-- Debezium -->
Produces Kafka event: OrderCreated { orderId: 42 }
```

Outbox + CDC Combo

Most robust solution:

- Use the **outbox table** for atomic writes
- Use **CDC** to stream events out automatically
- No dual writes, no lost events, no external transactions

Trade-offs

- Increased complexity (extra tables/processes)
- Possible duplication → requires idempotency

Distributed Transactions

When one business operation spans multiple services or databases.

- Each service has its own local data and transaction boundaries.
- We still want **atomicity** across them.

But... there's no global transaction manager in distributed systems.

Example

Booking a trip:

- Reserve a flight ✈️
- Book a hotel 🏨
- Charge a credit card 💳

If one step fails, the others must be reverted.

Two-Phase Commit (2PC)

Classic protocol to coordinate distributed commits.

1. **Prepare phase** - coordinator asks all participants if they can commit.
2. **Commit phase** - if all say “yes”, everyone commits; otherwise, all roll back.

2PC Pros and Cons

- ✅ Guarantees atomicity across systems.
- ❌ Introduces a single point of failure – the coordinator.
- ❌ Participants must lock data until the commit is decided.
- ❌ Doesn’t scale well under high latency or partial failures.

When 2PC Fails

If the coordinator crashes between phases...

- Some participants may have committed.
- Others may still be waiting.
- System enters an **uncertain state**.

Recovery requires manual intervention or a timeout heuristic.

The Saga Pattern

An alternative to distributed transactions.

- Each step is a **local transaction**.
- On failure, execute **compensating actions**.

Two Saga Coordination Styles

- **Choreography** - services react to each other's events.
- **Orchestration** - a central orchestrator tells each participant what to do.

Both achieve eventual consistency, but with different trade-offs.

Saga Example

```
BookTripSaga:
  1. FlightService.reserve(flight)
  2. HotelService.book(hotel)
  3. PaymentService.charge(user)

If step 3 fails:
  -> HotelService.cancelBooking()
  -> FlightService.releaseSeat()
```

Transaction Isolation

Even inside a single database, transactions can interfere with each other.

Isolation defines **how visible** one transaction's changes are to others.

Remember PACELC?

ACID Refresher

- **A**tomicity - all or nothing
- **C**onsistency - valid state transitions
- **I**solation - no interference
- **D**urability - once committed, it stays

Why Isolation Matters

Concurrent transactions may cause anomalies:

- **Dirty Read** - see uncommitted data
- **Non-repeatable Read** - unexpected data changes
- **Phantom Read** - new rows appear unexpectedly
- **Lost Update** - two transactions overwrite each other

Dirty Read Example

```
T1: UPDATE accounts SET balance = balance - 100 WHERE id=1;
T2: SELECT balance FROM accounts WHERE id=1; -- sees uncommitted
T1: ROLLBACK;
```

T2 read something that never really existed.

Non-Repeatable Read Example

```
T1: SELECT * FROM orders WHERE id = 1; -- sees "status = NEW"
T2: UPDATE orders SET status = 'PAID' WHERE id = 1; COMMIT;
T1: SELECT * FROM orders WHERE id = 1; -- sees "status = PAID"
```

Same query, different result - during one transaction.

Phantom Read Example

```
T1: SELECT * FROM orders WHERE status = 'NEW'; -- returns 3 rows
T2: INSERT INTO orders (status) VALUES ('NEW'); COMMIT;
T1: SELECT * FROM orders WHERE status = 'NEW'; -- now 4 rows
```

New “phantom” data appears mid-transaction.

Write Skew Anomaly

Occurs when two concurrent transactions read overlapping data and make **non-conflicting writes** based on those reads.

Even though each transaction is consistent on its own, the *combined result* breaks an invariant.

Classic Example

Hospital rule: **at least one doctor must be on call.**

```
CREATE TABLE doctors (
    id INT,
    on_call BOOLEAN
);
-- Initially:
-- Dr. Alice: on_call = true
-- Dr. Bob:   on_call = true
```

Step 1 - Two Transactions Start

```
T1: SELECT * FROM doctors WHERE on_call = true;
-- sees Alice + Bob

T2: SELECT * FROM doctors WHERE on_call = true;
-- sees Alice + Bob
```

Both see that someone else is on call.

Step 2 - Both Decide to Go Off Call

```
T1: UPDATE doctors SET on_call = false WHERE name = 'Alice';
T2: UPDATE doctors SET on_call = false WHERE name = 'Bob';
```

Each assumes the other doctor stays on call.

Step 3 - Both Commit

```
T1: COMMIT;
T2: COMMIT;
```

Invariant broken: nobody is on call anymore 🚨

Why It Happens

- Both transactions read the same initial state
- Each makes a decision that was valid at that moment
- No direct conflict → no locking → no blocking
- Combined result violates a business rule

Isolation Levels and Write Skew

Isolation Level	Prevents Write Skew?
Read Committed	❌
Repeatable Read	❌ (in most DBs)
Serializable	✅

Only Serializable prevents it, e.g. via predicate locks or SSI (Serializable Snapshot Isolation).

Serializable ≠ Simple

Serializable isolation **simulates sequential execution**.

But it comes at a cost:

- Higher contention and locking
- Deadlocks
- Lower throughput

Using common sense is the ultimate Best Practice™.

Thank You!

Need help? Reach out! It's free.

@pivovarit

4comprehension.com