Architecture: DDD, CQRS, and Event Sourcing

You model the workshop shop with DDD, separate writes and reads with CQRS, and store orders as events, only where it pays off.

Mike Bild

Exercises: github.com/MikeBild/introduction-nodejs

Understand the domain

First the experts' language, then the code · Where a word changes meaning, there is a boundary · Does it have an identity or only a value?

Why is everything called update in the code?

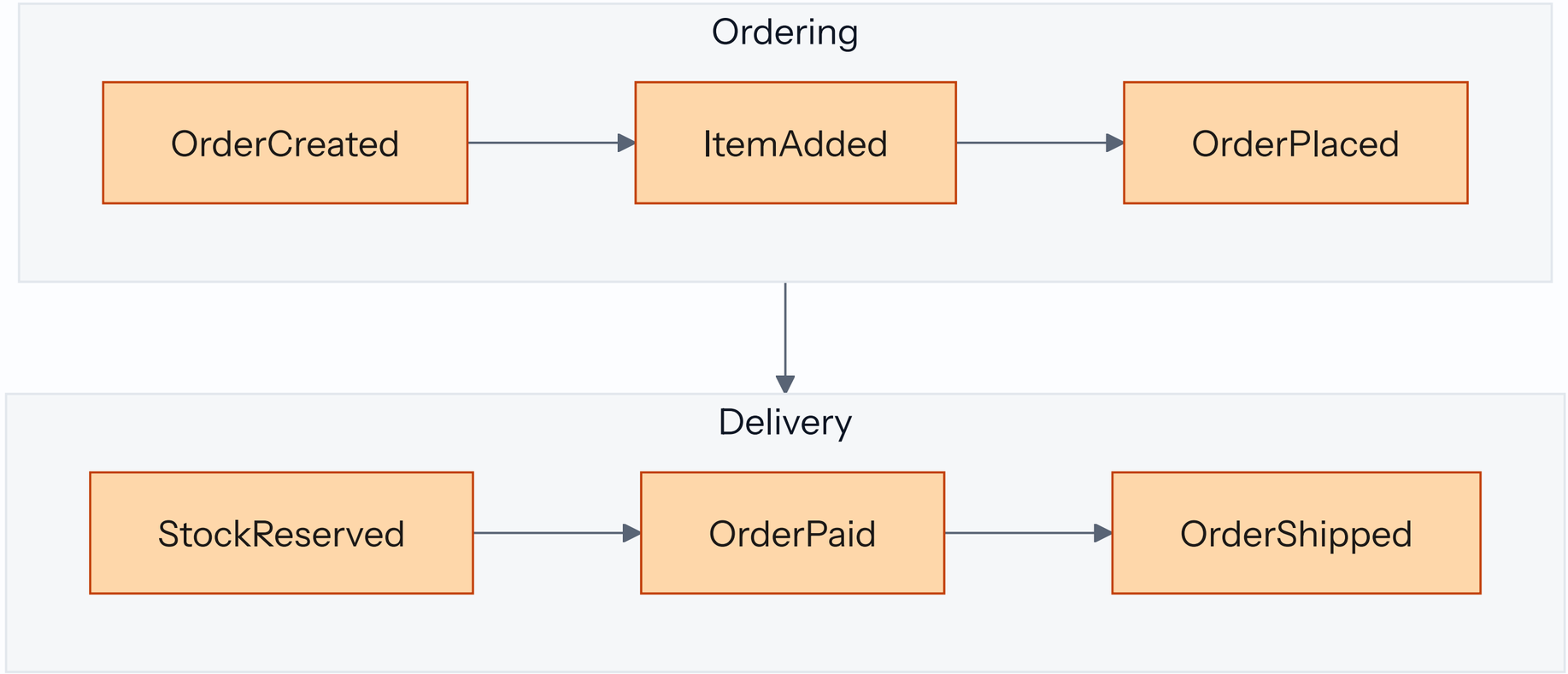
- The code speaks in tables: save, update, status
- The experts speak in events: ordered, paid, shipped
- Every translation between them costs time and creates errors

One word, one meaning

```
export const glossary = {
  Order: 'Bestellung einer Kundin, ab „placed“ verbindlich',
  Item: 'Position einer Bestellung: Produkt, Menge und Preis in Cent',
  Stock: 'Bestand eines Produkts im Lager, abzüglich Reservierungen',
}
```

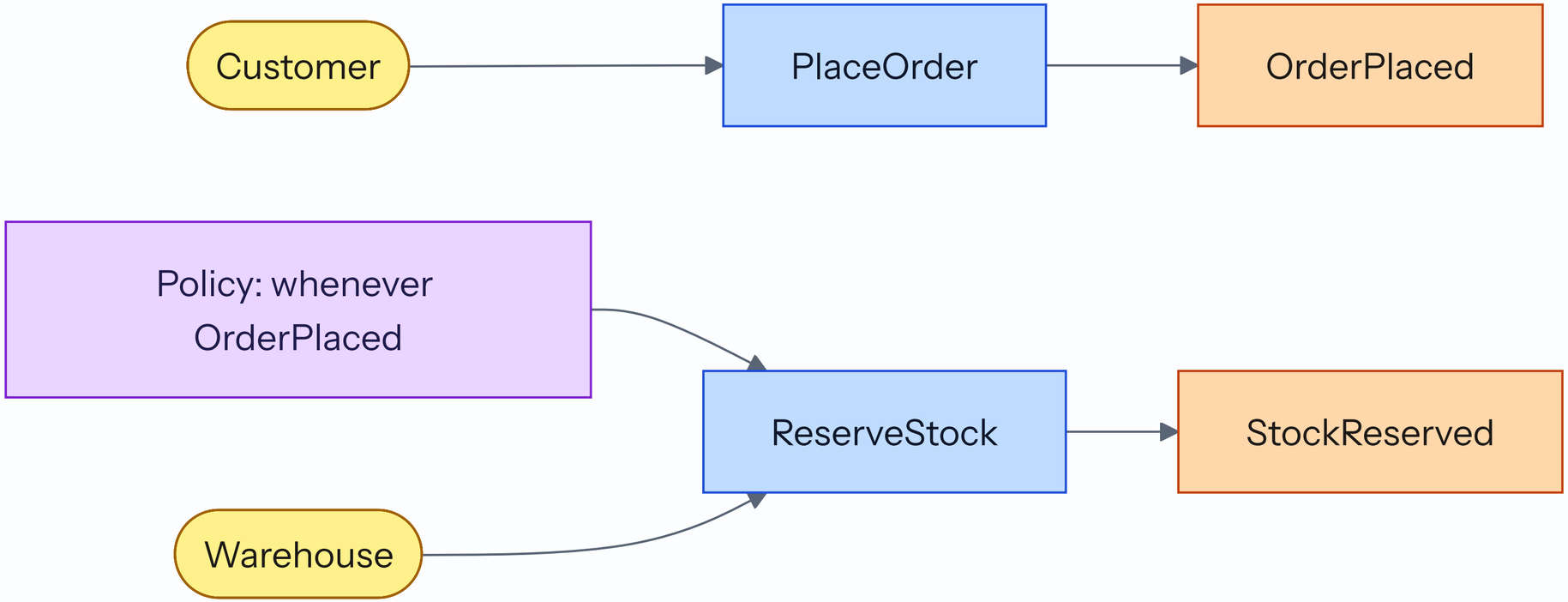
Event Storming starts with what happened

- Orange sticky notes: Domain Events in the past tense



Commands and Policies explain why

- Orange: Event, blue: Command, yellow: Actor, purple: Policy



Hotspots are the most valuable sticky notes

- Red sticky note for every open question
- Disagreement between experts is a finding, not a mistake
- You get answers later instead of guessing them in code

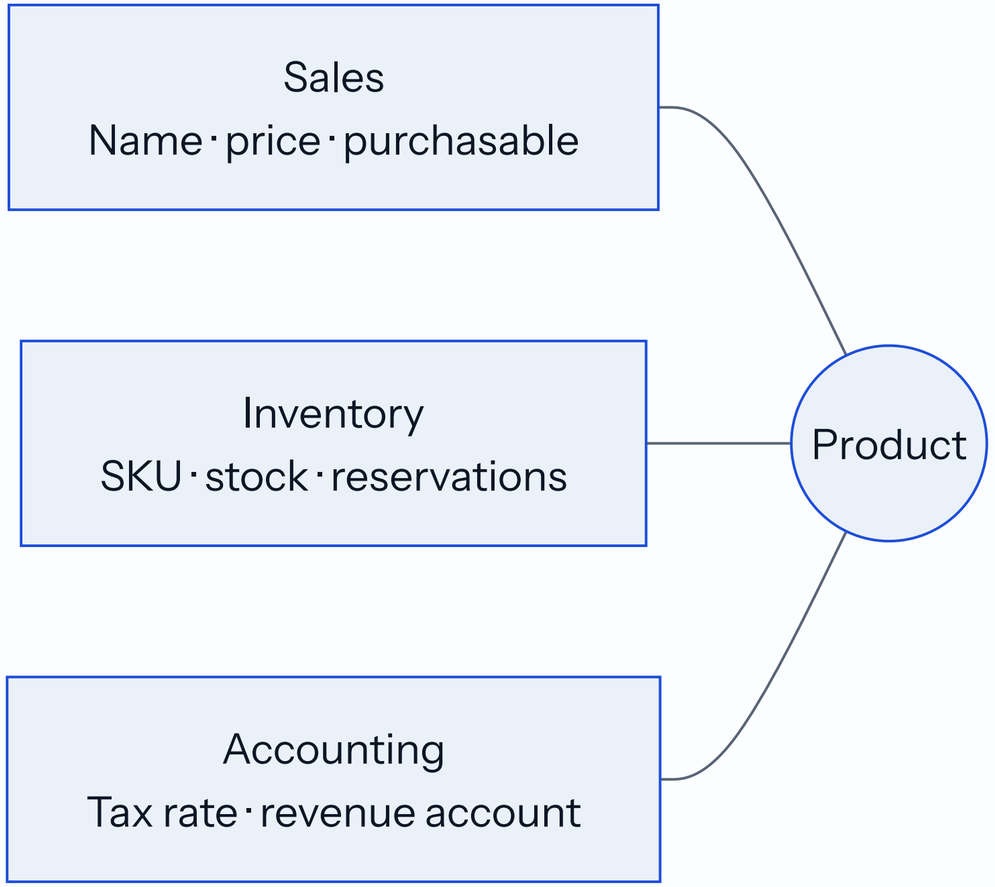
The board as code checks the language

```
export type Sticky =
  | { kind: 'event'; name: string }
  | { kind: 'command'; name: string; actor: string; emits: string[] }
  | { kind: 'policy'; whenever: string; then: string }
  | { kind: 'hotspot'; question: string }

const technicalVerbs = ['Save', 'Update', 'Set', 'Change', 'Process']

// An event is in the past tense: Placed, Paid, Shipped …
const isPastTense = (name: string) => /(ed|Paid|Sent|Sold)$/.test(name)
```

One product, three meanings



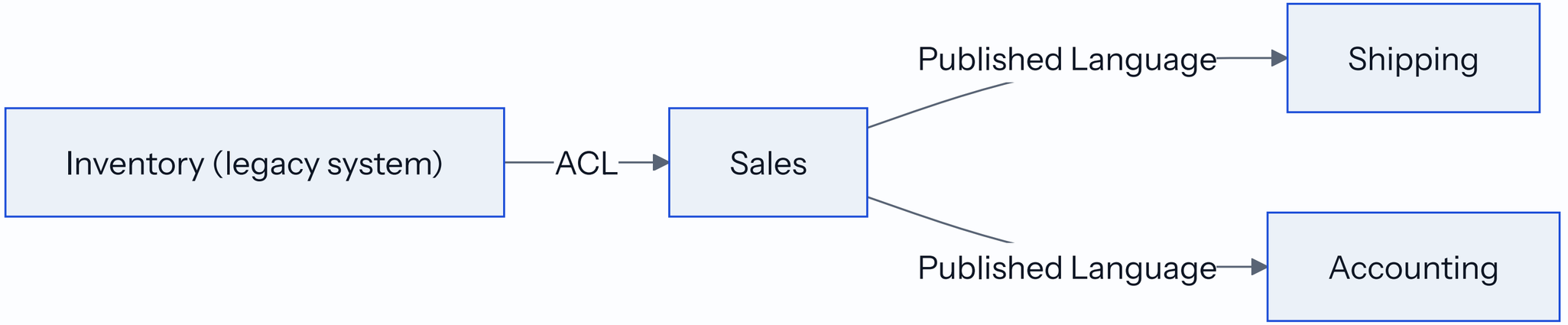
A Bounded Context is a language boundary

- Within it, exactly one model with exactly one language applies
- Across the boundary, the same word may mean something else
- Rules that apply only here mark the boundary

Core, Supporting, or Generic?

- Core Domain: what sets your company apart from competitors
- Supporting Subdomain: necessary, but not a differentiator
- Generic Subdomain: already solved, buy instead of build

The edges are the architecture



An Anti-Corruption Layer translates in one place

```
// “19,99” → 1999, without a floating-point number
export const parseEuroToCents = (text: string) => {
  const match = /^(\d+),(\d{2})$/.exec(text)
  if (!match) throw new Error(`Invalid price ${text}`)
  return Number(match[1]) * 100 + Number(match[2])
}

export const toProduct = (item: InventoryItemDto): Product => ({
  id: ProductId(item.sku),
  name: item.title,
  priceCents: parseEuroToCents(item.price_eur),
  available: item.qty_on_hand - item.qty_reserved > 0,
})
```

First the boundary, then the service

- Bounded Context: boundary of the model and the language
- Service: unit for deployment and operations
- One context may have multiple services, not the other way around

What slips through when everything is string?

```
const reorder = (customerId: string, productId: string) =>
  `${customerId} bestellt ${productId} nach`

reorder('p-1', 'c-1') // vertauscht, kompiliert trotzdem

createOrderLine('p-1', 0, 12.9) // Menge null, Preis in Euro
lineTotal(createOrderLine('p-4', 3, 0.1)) // 0.30000000000000004
```

An Entity has an identity

- Equal means having the same ID
- Name and email change; the customer stays the same
- Customer, order, and product are Entities

A Value Object is only its value

- Equal means having the same values
- Immutable: changing means creating a new one
- Price, quantity, email, and line item are values

Smart Constructors check once

```ts
type Brand<T, B extends string> = T & { readonly __brand: B }

export type Cents = Brand<number, 'Cents'>
export const cents = (value: number): Cents => {
  if (!Number.isInteger(value) || value < 0) throw new Error(`Invalid amount ${value}`)
  return value as Cents
}

export type Quantity = Brand<number, 'Quantity'>
export const quantity = (value: number): Quantity => {
  if (!Number.isInteger(value) || value < 1) throw new Error(`Invalid quantity ${value}`)
  return value as Quantity
}
```

Line items are frozen values

```ts
export type OrderItem = Readonly<{ productId: ProductId; quantity: Quantity; priceCents: Cents }>

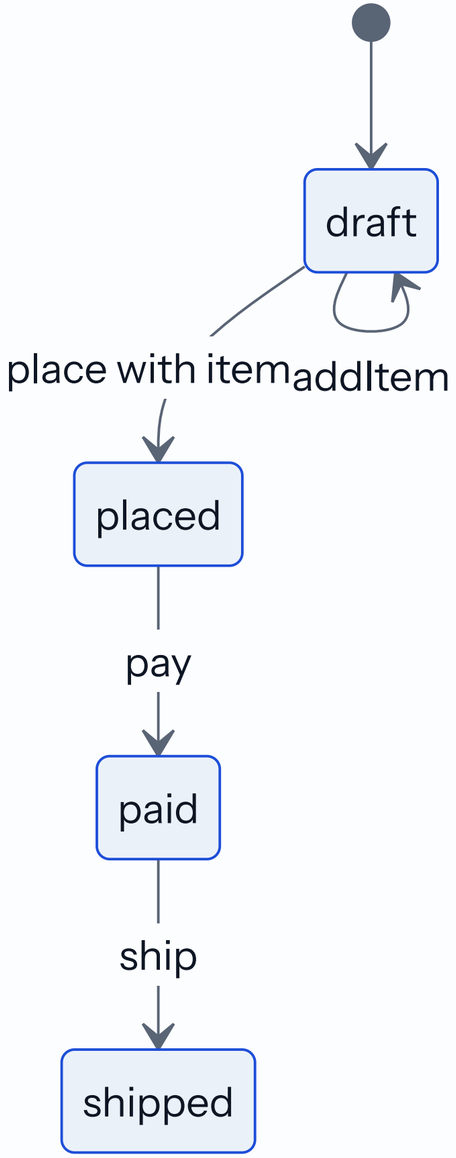
export const orderItem = (input: { productId: string; quantity: number; priceCents: number }) =>
  Object.freeze({
    productId: ProductId(input.productId),
    quantity: quantity(input.quantity),
    priceCents: cents(input.priceCents),
  }) satisfies OrderItem

export const sameItem = (a: OrderItem, b: OrderItem) =>
  a.productId === b.productId && a.quantity === b.quantity && a.priceCents === b.priceCents
```

Rules, messages, and separate models

An aggregate protects its rules as a unit · Is this a wish, a fact, or a question? · CQRS is a domain boundary, not a framework

Who protects the order’s rules?



The root is the only entry point

```
export const addItem = (order: Order, item: OrderItem): Order => {
  if (order.status !== 'draft') throw new Error('Only draft orders accept items')
  if (!Number.isInteger(item.quantity) || item.quantity < 1)
    throw new Error('Quantity must be at least 1')
  return { ...order, items: [...order.items, item] }
}

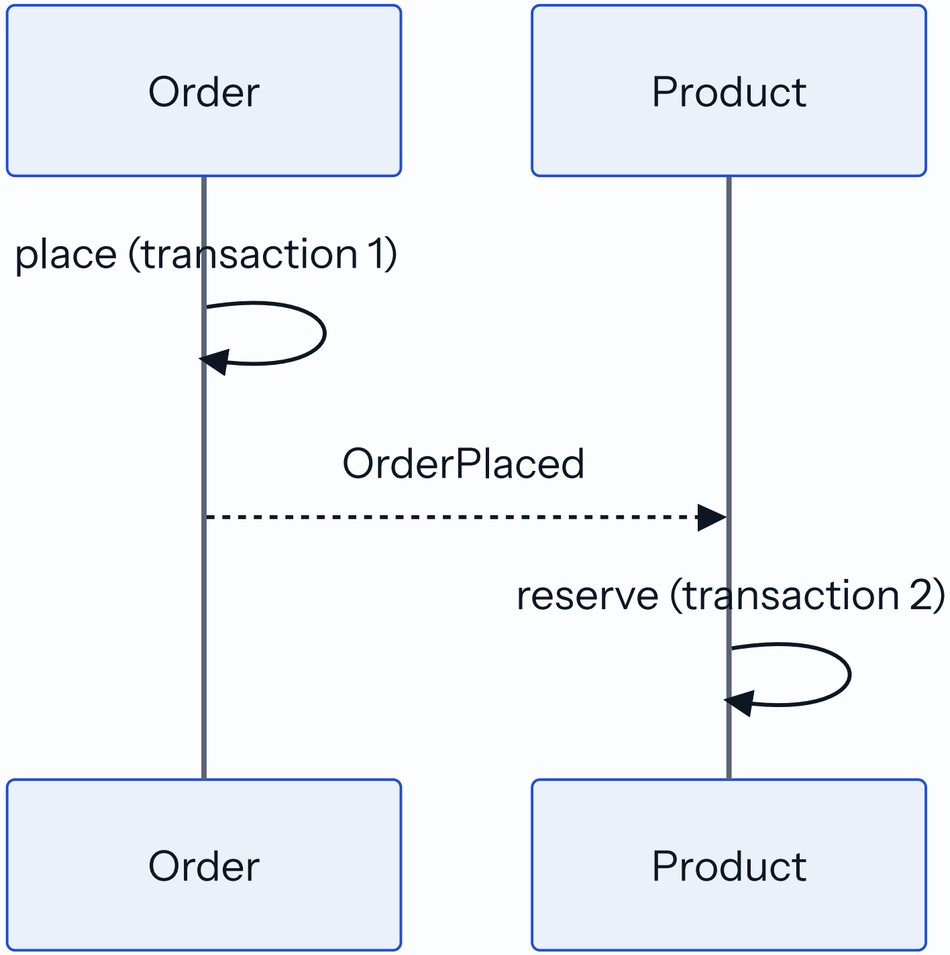
export const place = (order: Order): Order => {
  if (order.status !== 'draft') throw new Error('Only draft orders can be placed')
  if (order.items.length === 0) throw new Error('An order needs at least one item')
  return { ...order, status: 'placed' }
}
```

Other aggregates only by ID

```
export type Order = Readonly<{
  id: OrderId
  customerId: CustomerId // not customer: Customer
  status: OrderStatus
  items: readonly OrderItem[] // productId instead of Product
}>

export type Product = Readonly<{
  id: ProductId
  stock: number
  reservations: Readonly<Record<OrderId, number>>
}>
```

One transaction, one aggregate



Save with the expected version

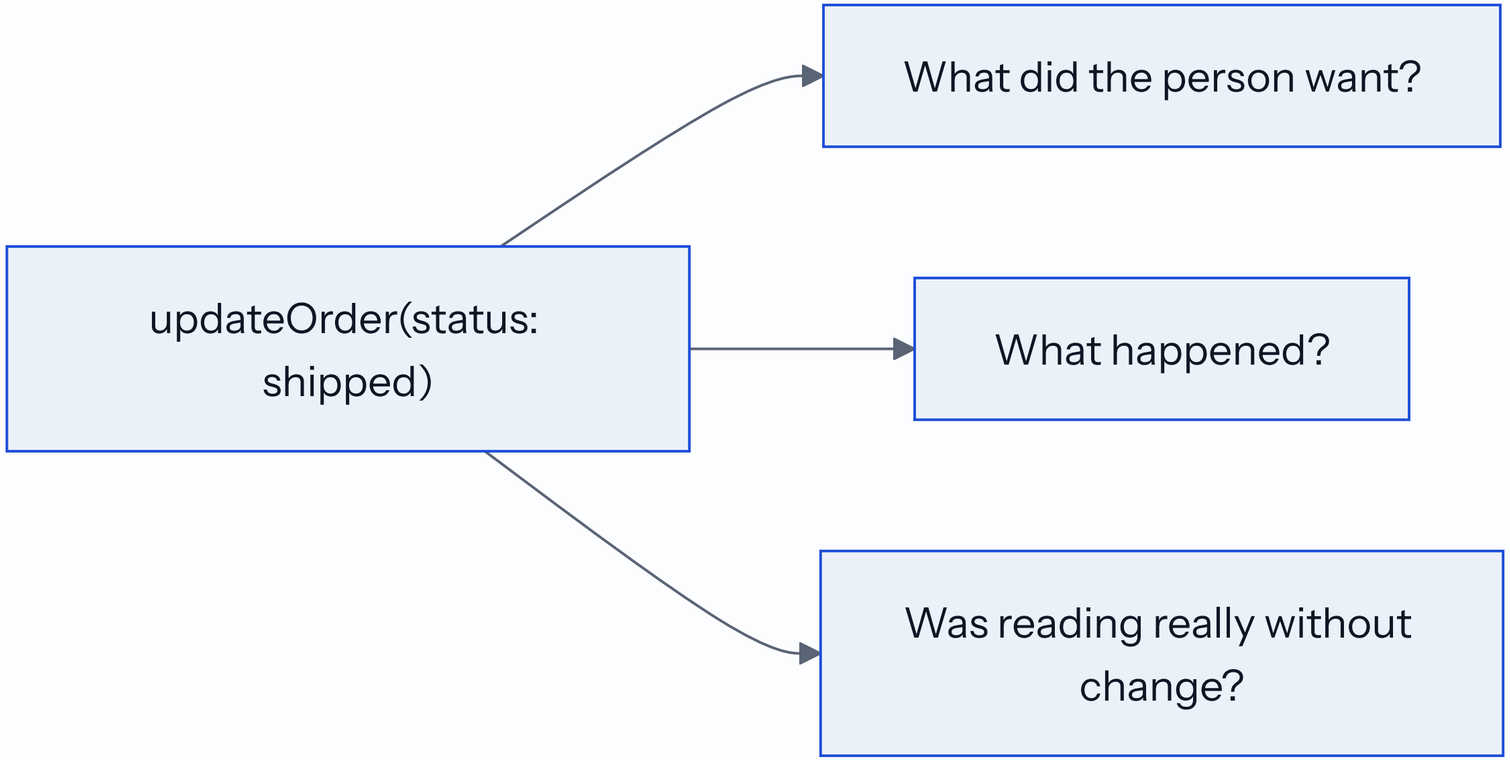
```
export const change = async <T extends { id: string }>(
  repository: Repository<T>,
  id: string,
  decide: (value: T) => T,
) => {
  const { value, version } = await repository.load(id)
  await repository.save(decide(value), version) // wirft ConcurrencyError
}

await change(orders, orderId, place)
await change(products, productId, (p) => reserve(p, orderId, 2))
```

Three pitfalls

- Aggregate equals table or Entity from the ORM
- Root as a god object that contains everything
- Direct references to other aggregates

What does updateOrder mean?



Three messages, three meanings

```
export type OrderCommand =
  | { type: 'AddItem'; orderId: string; item: OrderItem }
  | { type: 'PlaceOrder'; orderId: string }
  | { type: 'ShipOrder'; orderId: string }

export type OrderEvent =
  | { type: 'ItemAdded'; orderId: string; item: OrderItem }
  | { type: 'OrderPlaced'; orderId: string; placedAt: string }
  | { type: 'OrderShipped'; orderId: string; shippedAt: string }

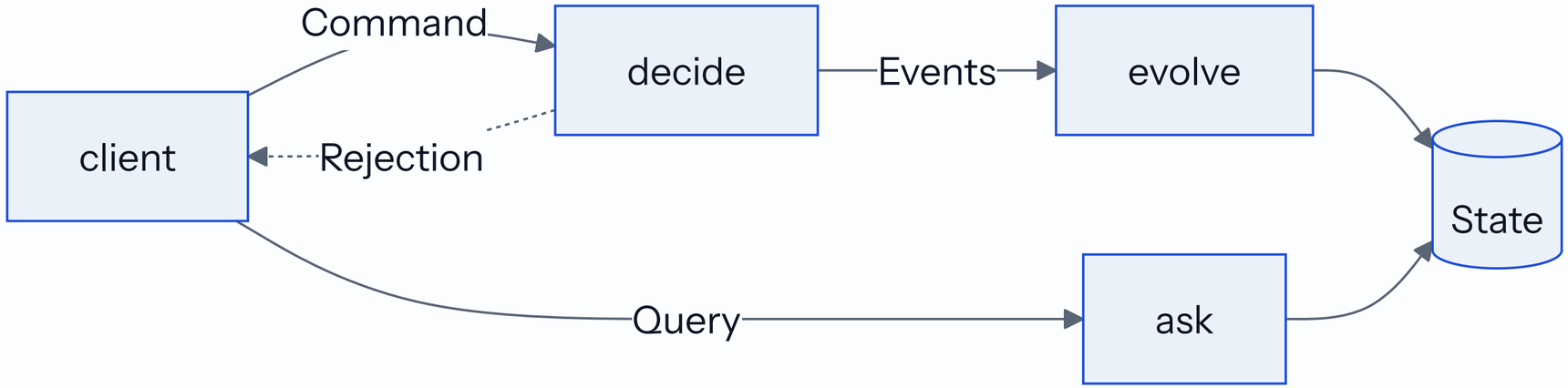
export type OrderQuery = { type: 'GetOrder'; orderId: string } | { type: 'ListOpenOrders' }
```

decide checks, evolve applies

```ts
export const decide = (order: Order, command: OrderCommand, now = new Date()): OrderEvent[] => {
  switch (command.type) {
    case 'PlaceOrder':
      expectStatus(order, ['draft'], 'Only draft orders can be placed')
      if (order.items.length === 0) throw new Error('An order needs at least one item')
      return [{ type: 'OrderPlaced', orderId: command.orderId, placedAt: now.toISOString() }]
    // …
  }
}

export const evolve = (order: Order, event: OrderEvent): Order => {
  switch (event.type) {
    case 'OrderPlaced':
      return { ...order, status: 'placed' }
    // …
  }
}
```

Wish, fact, answer



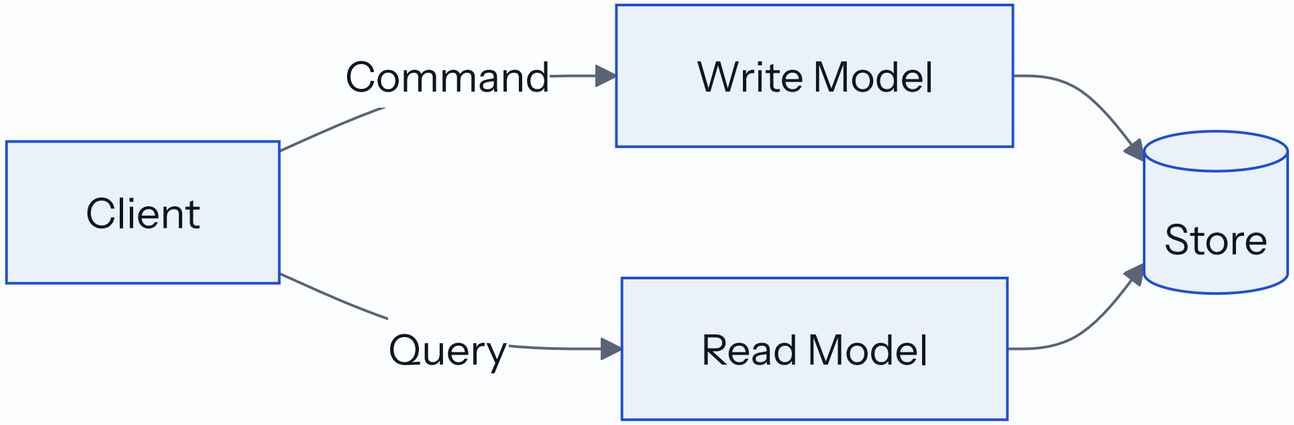
Where naming misleads

- Events in the imperative, Commands in the past tense
- Queries with Side Effect, such as a timestamp when reading
- Commands as setters: SetStatus instead of ShipOrder

One model for everything gets too large

- The overview loads each aggregate and each customer separately
- Display fields drift into the Write Model
- Rules and presentation always change together

CQRS separates models, not databases



The write side validates and stays silent

```
const change = (store: Store, id: string, decide: (order: Order) => Order): void =>
  store.transaction((tx) => {
    const order = tx.orders.get(id)
    if (!order) throw new Error(`Order ${id} not found`)
    const next = decide(order)
    tx.orders.set(id, next)
    updateOpenOrders(tx, next) // View in derselben Transaktion
  })

export const placeOrder = (store: Store, id: string) =>
  change(store, id, (order) => {
    if (order.items.length === 0) throw new Error('An order needs at least one item')
    return { ...order, status: 'placed' }
  })
```

The read side returns exactly the row

```
export type OpenOrderRow = {
  orderId: string
  customerName: string
  itemCount: number
  totalCents: number
  status: 'placed' | 'paid'
}

export const listOpenOrders = (store: Store) =>
  store.read((tables) =>
    [...tables.openOrders.values()].sort((a, b) => a.orderId.localeCompare(b.orderId)),
  )
```

One store is enough to start

- View in the same transaction: no delay
- Second store only when load or shape requires it
- It brings Eventual Consistency into play

Event Sourcing

Event Sourcing stores what happened · You can rebuild a read model anytime · Events stay readable for years

An update forgets how it got there

- CRUD overwrites the old state
- Event Sourcing only appends and never changes
- The Event Store becomes the domain logbook

The state is a Left Fold

```
type OrderEvent =
  | { type: 'ItemAdded'; quantity: number; priceCents: number }
  | { type: 'OrderPlaced'; placedAt: string }
  | { type: 'OrderPaid'; paidAt: string }
  | { type: 'OrderShipped'; shippedAt: string }
type Order = { status: 'draft' | 'placed' | 'paid' | 'shipped'; totalCents: number }

const next = { OrderPlaced: 'placed', OrderPaid: 'paid', OrderShipped: 'shipped' } as const

const evolve = (o: Order, e: OrderEvent): Order =>
  e.type === 'ItemAdded'
    ? { ...o, totalCents: o.totalCents + e.quantity * e.priceCents }
    : { ...o, status: next[e.type] }
```

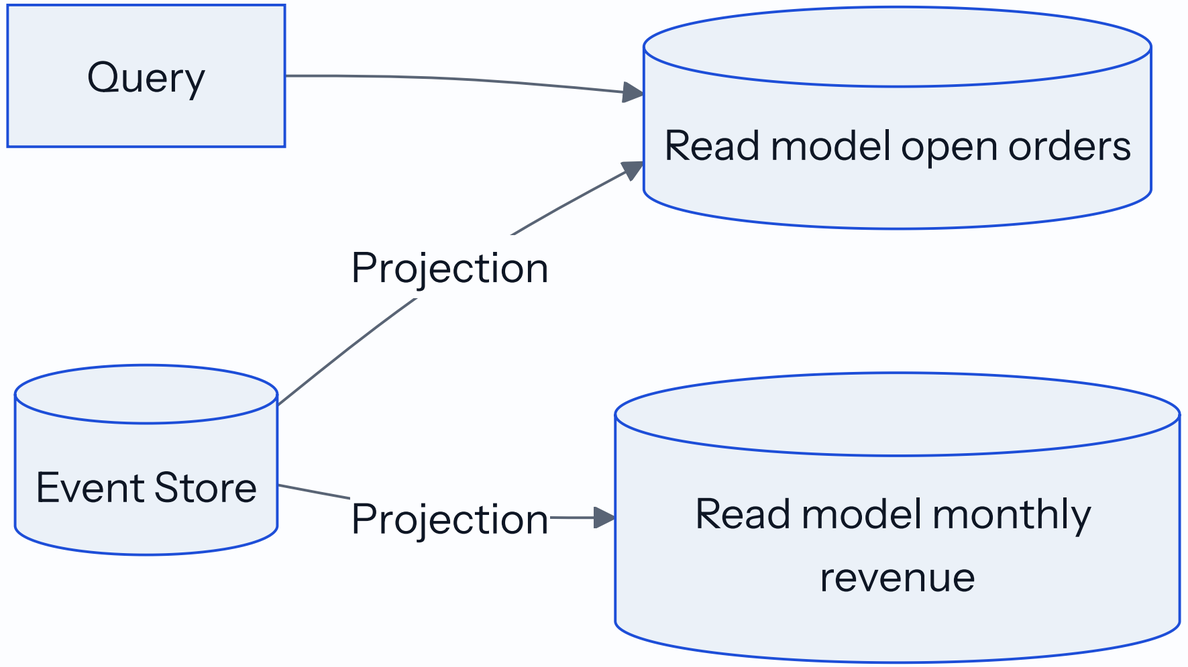
The aggregate decides, the store appends

```
const initial: Order = { status: 'draft', totalCents: 0 }

const ship = (order: Order): OrderEvent[] => {
  if (order.status !== 'paid') throw new Error('Only paid orders can ship')
  return [{ type: 'OrderShipped', shippedAt: new Date().toISOString() }]
}

export const shipOrder = async (store: EventStore, orderId: string) => {
  const { events, version } = await store.load(orderId)
  const order = events.reduce(evolve, initial)
  await store.append(orderId, version, ship(order))
}
```

Projections are disposable



What Event Sourcing costs

- Events stay readable for years and need versions
- Read models lag slightly behind the store
- A replay must not send email

Projection or cache?

- Projection: a view from events, rebuildable anytime
- Cache: a copy that silently goes stale at some point
- Rebuild is the test that tells them apart

A fold with a checkpoint

```ts
export type Projection<S> = {
  name: string
  initial: S
  apply: (state: S, stored: StoredEvent) => S
}
export type Checkpointed<S> = { state: S; position: number }

// Idempotent: What has already been processed changes nothing.
export const step = <S>(projection: Projection<S>, current: Checkpointed<S>, stored: StoredEvent) =>
  stored.position <= current.position
    ? current
    : { state: projection.apply(current.state, stored), position: stored.position }
```

Rebuild is the same function from zero

```
// Continue reading from the checkpoint, for example after a restart
export const catchUp = async <S>(
  store: EventStore,
  projection: Projection<S>,
  current: Checkpointed<S>,
) =>
  (await store.readAll(current.position)).reduce(
    (acc, stored) => step(projection, acc, stored),
    current,
  )

// Rebuild: the same function, just from position zero
export const rebuild = <S>(store: EventStore, projection: Projection<S>) =>
  catchUp(store, projection, { state: projection.initial, position: 0 })
```

Side effects belong in a reactor

```
export const paymentMails = (mailer: Mailer) => {
  let position = 0
  return {
    handle: (stored: StoredEvent) => {
      if (stored.position <= position) return
      if (stored.event.type === 'OrderPaid') mailer.send(stored.streamId, 'Danke für deine Zahlung')
      position = stored.position
    },
  }
}
```

Snapshots are stored folds

```ts
export type Snapshot = { version: number; state: Order }

// Load from snapshot: fold only the events after it
export const loadFrom = async (store: EventStore, id: string, snapshot?: Snapshot) => {
  const { events, version } = await store.load(id)
  const from = snapshot?.version ?? 0
  return { state: events.slice(from).reduce(evolve, snapshot?.state ?? initial), version }
}
```

The schema changes, the event log does not

- Stored events are never changed or deleted
- evolve knows only the current schema
- Between them, you need a translation

Every event is inside an envelope

```ts
export type Envelope = { type: string; version: number; data: Record<string, unknown> }

export const log: readonly Envelope[] = [
  { type: 'ItemAdded', version: 1, data: { productId: 'p-1', quantity: 2, price: 12.9 } },
  { type: 'OrderPlaced', version: 1, data: { at: '2023-05-02T08:00:00.000Z' } },
  { type: 'ItemAdded', version: 2, data: { productId: 'p-2', quantity: 1, priceCents: 1999 } },
  { type: 'OrderPlaced', version: 2, data: { placedAt: '2026-10-12T10:00:00.000Z' } },
]
```

One upcaster per version

```ts
type Upcaster = (data: Record<string, unknown>) => Record<string, unknown>

export const currentVersion: Record<OrderEvent['type'], number> = { ItemAdded: 2, OrderPlaced: 2 }

// One step to the next version for each type and version
const upcasters: Record<string, Upcaster> = {
  'ItemAdded@1': ({ price, ...rest }) => ({ ...rest, priceCents: Math.round(Number(price) * 100) }),
  'OrderPlaced@1': ({ at, ...rest }) => ({ ...rest, placedAt: at }),
}
```

Upcast to the current version

```
export const upcast = (envelope: Envelope): OrderEvent => {
  const target = currentVersion[envelope.type as OrderEvent['type']]
  if (target === undefined) throw new Error(`Unknown event type ${envelope.type}`)
  if (envelope.version > target)
    throw new Error(`${envelope.type} v${envelope.version} is newer than this code`)
  let { version, data } = envelope
  while (version < target) {
    data = upcasters[`${envelope.type}@${version}`]!(data)
    version++
  }
  return { type: envelope.type, ...data } as OrderEvent
}
```

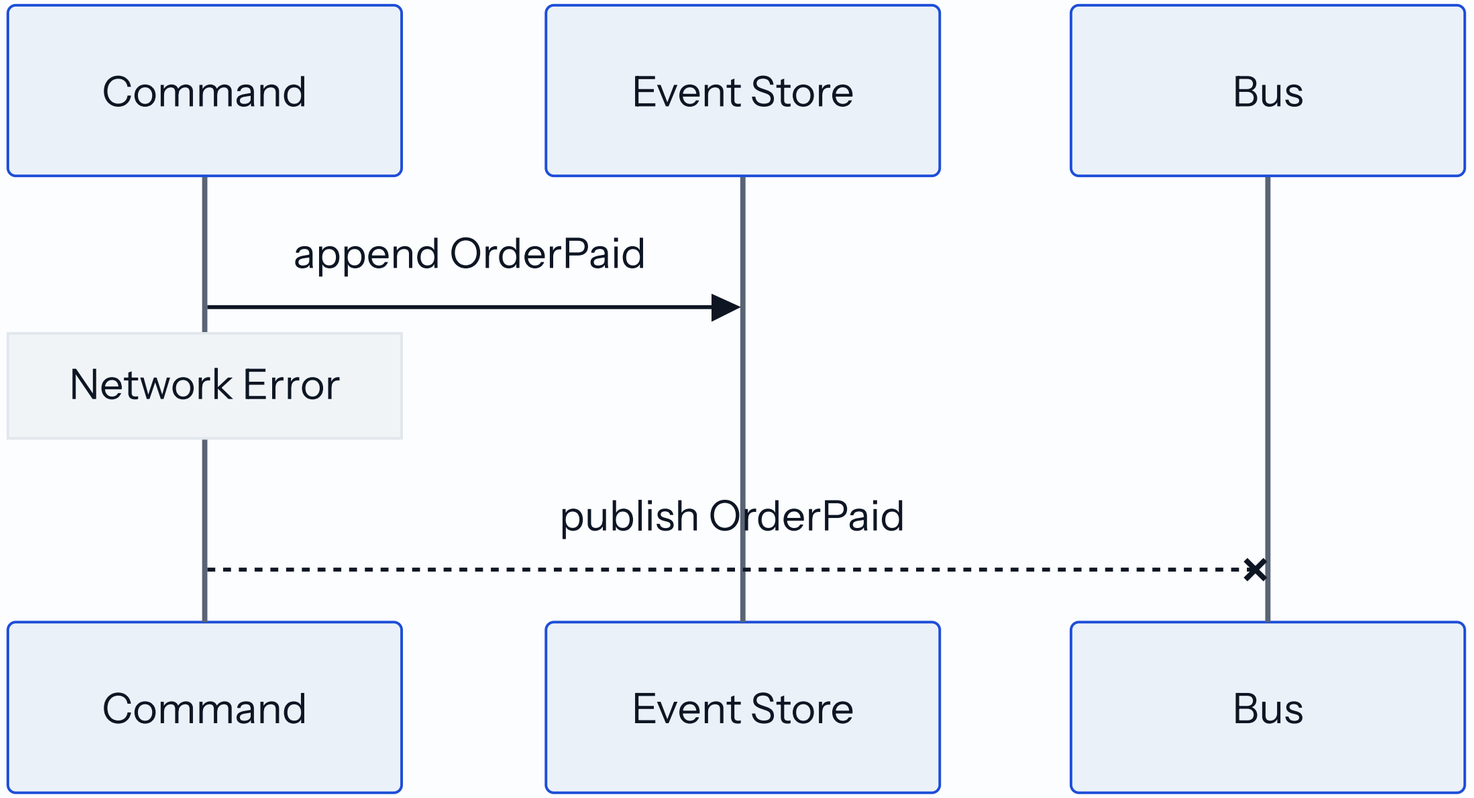
Not every change needs a version

- New optional field: old readers ignore it
- Rename or convert: new version with upcaster
- New meaning: new event type instead of new version

Consistency and decisions

Your event store is already your outbox · ADRs record the why next to the code · When are CQRS and Event Sourcing too much?

Two systems, no shared commit



The relay reads from its checkpoint

```
export const createRelay = (store: EventStore, bus: Bus) => {
  let checkpoint = 0
  return {
    pump: async () => {
      for (const stored of await store.readAll(checkpoint)) {
        await bus.publish(stored) // throws? Then the checkpoint stays put.
        checkpoint = stored.position
      }
    },
  }
}
```

Consumers detect duplicates

```
// Stream and version are the key for each event.
export const idempotent = (handle: (stored: StoredEvent) => void) => {
  const seen = new Set<string>()
  return (stored: StoredEvent) => {
    const key = `${stored.streamId}@${stored.version}`
    if (seen.has(key)) return
    handle(stored)
    seen.add(key)
  }
}
```

Eventual Consistency is a deliberate trade-off

- The command returns the position it wrote
- The query waits until the read model knows it
- Or the UI honestly shows the older state

Read-your-writes by position

```ts
export const readAfter = async (
  view: StatusView,
  position: number,
  id: string,
  attempts = 3,
) => {
  for (let attempt = 0; attempt < attempts && view.position() < position; attempt++)
    await view.catchUp()
  if (view.position() < position) throw new Error('Read model is behind, try again')
  return view.status(id)
}

const position = await payOrder(store, orderId)
await readAfter(view, position, orderId) // 'paid'
```

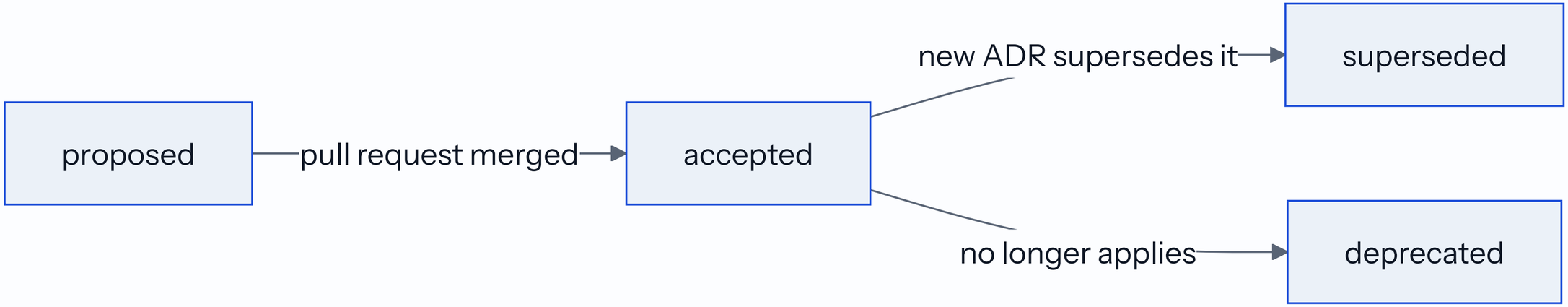
Why is this written this way?

- Code shows what was built
- The why is in chats, meetings, and people’s heads
- New team members dig instead of read

Five sections, one decision



An ADR is never overwritten



When is an ADR worth it?

- The decision is hard to reverse
- It affects multiple teams
- It rejects an obvious alternative

The baseline is CRUD

- One model, one table, immediately consistent
- The right answer for most contexts
- Not a compromise, but the starting point

What the patterns really cost

- More types, handlers, and moving parts
- Delay between writes and reads
- Events that must stay readable for years

One decision flow per context

```
1. Colliding rules?            no  → CRUD
2. Asymmetry or history?       no  → CQRS-lite, one store
3. Team carries the lag?       no  → CRUD with audit table
4. History part of domain?     yes → CQRS with Event Sourcing
                               no  → CQRS
```

The flow as code

```
export const recommend = (context: ContextProfile): Recommendation => {
  if (!context.invariantsCollide) return 'crud'
  if (!context.readWriteAsymmetry && !context.historyRequired) return 'cqrs-lite'
  if (!context.teamReady) return 'compromise'
  return context.historyRequired ? 'cqrs-event-sourcing' : 'cqrs'
}

// Werkstatt-Shop: Katalog → crud, Kunden → crud, Lager → cqrs-lite,
// Bestellungen → cqrs-event-sourcing, Retouren → compromise
```

The pattern as an end in itself

- Event Sourcing everywhere because it sounds modern
- CRUD in disguise: events like OrderUpdated
- Two stores before there is a reason

Questions?

Mike Bild · mike.bild@gmail.com · mikebild.dev

Workshops für Teams, die Architekturentscheidungen selbst treffen