Durable Execution

An Introduction

What if your functions could survive crashes, restarts, and weeks of waiting?

Samuel Thien

A Problem You've Seen

THE PROBLEM

What happens when the server crashes?

You're building an online store. A customer places an order.

1. **Reserve inventory** (database write)
2. **Create order record** (database write)
3. **Charge payment** (payment API call)
4. **Notify warehouse** (webhook call)
5. **Start delivery dispatch** (async process)

THE PROBLEM

The partial failure problem

The server crashes between step 3 and step 4.

- Payment is charged. The customer's money is gone.
- But the warehouse was never notified. No one picks the order.
- The customer waits. Nothing arrives. They call support.
- **This is the partial failure problem.**

The system is in an **inconsistent state**, and no single component knows it.

TRADITIONAL SOLUTIONS

Why the obvious solutions don’t work

Approach	Why it falls short
Try/catch + retry	Retries the whole function. Step 3 runs again, customer is **double-charged**.
Database transaction	Can’t wrap payment API calls inside a Postgres transaction. Only works within one DB.
Message queue	Split logic across producers/consumers. Now manage dead-letter queues, idempotency keys, and a state machine.
Polling state machine	Store state in a `workflow_state` column, poll every 30s, giant switch/case.

Every one of these pushes reliability into **your application code**. You end up writing infrastructure, not features.

Long-running workflows — a solved problem?

You've all built this before.

- A `workflow_state` column with a state enum
- API endpoints that advance the state on each event
- Background jobs for timeouts and escalation
- A reconciliation cronjob to catch data that fell out of sync

It works. But look at what it costs you:

- **Business logic gets scattered** across handlers, state transitions, and cron jobs
- **Every workflow** needs its own state machine from scratch
- **Recovery code** is rarely tested, and often wrong when it finally runs
- A **5-step approval** that's 10 lines of logic becomes 200+ lines of infrastructure

The workflow runs. But you're writing more plumbing than business logic, and it's the same plumbing every time.

What if one approach solved both — crash recovery for fast operations and long-running coordination — without the state machine boilerplate?

Building from First Principles

BUILDING BLOCK 1

Checkpointing

The fundamental problem: when the process crashes, we **lose track of what already happened**.

The idea: After each step completes, save its result somewhere durable.

```
Step 1: Reserve inventory → result saved
Step 2: Create order → result saved
Step 3: Charge payment → result saved
CRASH
Step 4: Notify warehouse → never executed
Step 5: Dispatch → never executed
```

BUILDING BLOCK 1

How checkpointing works

We know *what happened*. But how do we *resume*?

BUILDING BLOCK 2

Replay

We have checkpoints. Now we need a way to **use them**.

The idea: On restart, re-run the workflow function from the beginning. But instead of executing completed steps, **return their saved results**.

```
Step 1: Checkpoint found → return saved result — skip
Step 2: Checkpoint found → return saved result — skip
Step 3: Checkpoint found → return saved result — skip
Step 4: No checkpoint → execute for real → saved
Step 5: No checkpoint → execute for real → saved
```

BUILDING BLOCK 2

Replay in practice

BUILDING BLOCK 3

Determinism

For replay to work, the function must produce the **same sequence of steps** every time.

The idea: Wrap non-deterministic values (time, random, config) in a checkpointed step.

This breaks replay

```
func dispatchWorkflow(ctx WorkflowCtx, orderID int)
    // ... previous steps checkpointed ...
    if time.Now().Hour() < 17 {
        step(ctx, func() { notifySameDay(orderID) })
    } else {
        step(ctx, func() { queueNextDay(orderID) })
    }
}
```

At 4:58 PM the code takes one path; after the crash at 5:03 PM it takes the other. **Same checkpoint, different branch.**

BUILDING BLOCK 3

Determinism — the fix

On replay, the runtime returns the **saved result** instead of re-reading the clock.

Fixed — checkpoint the decision

```
func dispatchWorkflow(ctx WorkflowCtx, orderID int)
    // ... previous steps checkpointed ...
    cutoff := step(ctx, func() bool {
        return time.Now().Hour() < 17
    })
    if cutoff {
        step(ctx, func() { notifySameDay(orderID) })
    } else {
        step(ctx, func() { queueNextDay(orderID) })
    }
}
```

Same rule for random values, UUIDs, and config reads. If it could differ between runs, **checkpoint it**.

BUILDING BLOCK 4

Durable Messaging

If you know Go channels, you already know this.

The idea: Two primitives

1. `Recv` blocks a workflow until a message arrives.
2. `Send` delivers it.

Like `←ch` and `ch ←`, but the state lives in the database (survives restarts, waits for days with zero compute)

Go channel

```
ch := make(chan string)

// goroutine blocks until value arrives
decision := ←ch

// another goroutine sends
ch ← "approved"
```

Durable messaging

```
// workflow blocks until message arrives
decision, _ := dbos.Recv[string](
    ctx, "approval", 7*24*time.Hour)

// HTTP handler sends
dbos.Send(ctx, wfID, "approval", "approved")
```

BUILDING BLOCK 4

Durable messaging in action

What did we just build?

#	Building Block	What it does
1	**Checkpointing**	Saves each step's result to a database
2	**Replay**	On restart, re-runs the function, skips completed steps
3	**Determinism**	Ensures replay produces the same step sequence
4	**Durable messaging**	Lets workflows sleep and wake across crashes

You write a normal function. The runtime makes it crash-proof.

This pattern is called durable execution.

You write a normal function. The runtime makes it crash-proof. That's it.

What durable execution is *not*

- It's **not a database**. It uses one, but it's not replacing Postgres.
- It's **not a message queue**. It's not competing with Kafka or RabbitMQ.
- It's **not a job scheduler**. It's not cron with extra steps.

It's a **runtime pattern** that makes your functions survive failures. Think of it as a reliability layer you add to existing code.

The Landscape

THE LANDSCAPE

Tools in this space

Runtime	What you deploy	Best for
Temporal	Server cluster (4 services) + workers	Enterprise scale, multi-team orchestration
DBOS	Library → your existing Postgres	Adding durability without a separate server cluster
Restate	Single server binary + SDK	Low-latency stateful microservices
Inngest	Nothing — serverless, HTTP-invoked	Event-driven flows, no workers to manage
Azure Durable Functions	Cloud-managed replay engine (Azure)	Teams already on Azure
AWS Lambda Durable Functions	Cloud-managed replay engine (AWS)	Serverless durable execution on AWS

THE LANDSCAPE

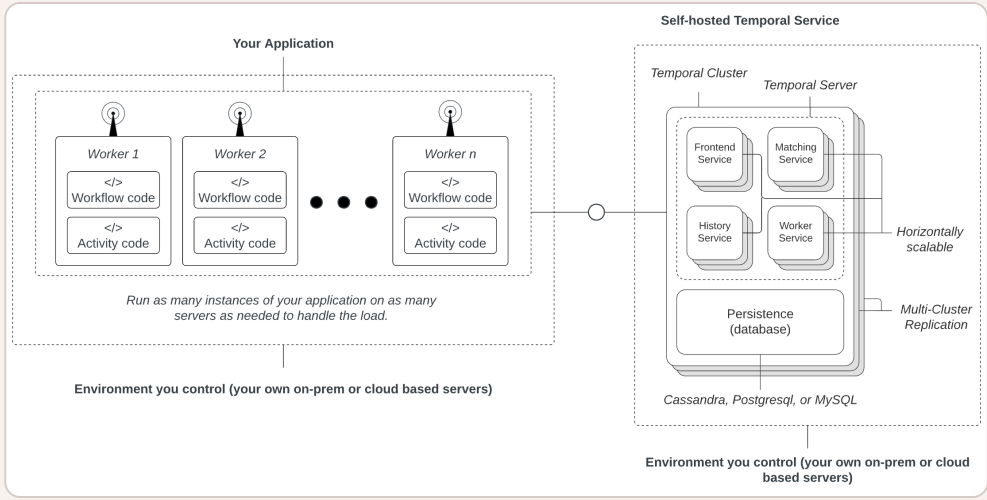
Architecture spectrum

Three models, same building blocks, different trade-offs.

Embedded tools are easier to try. Dedicated runtimes usually buy you more operational features.

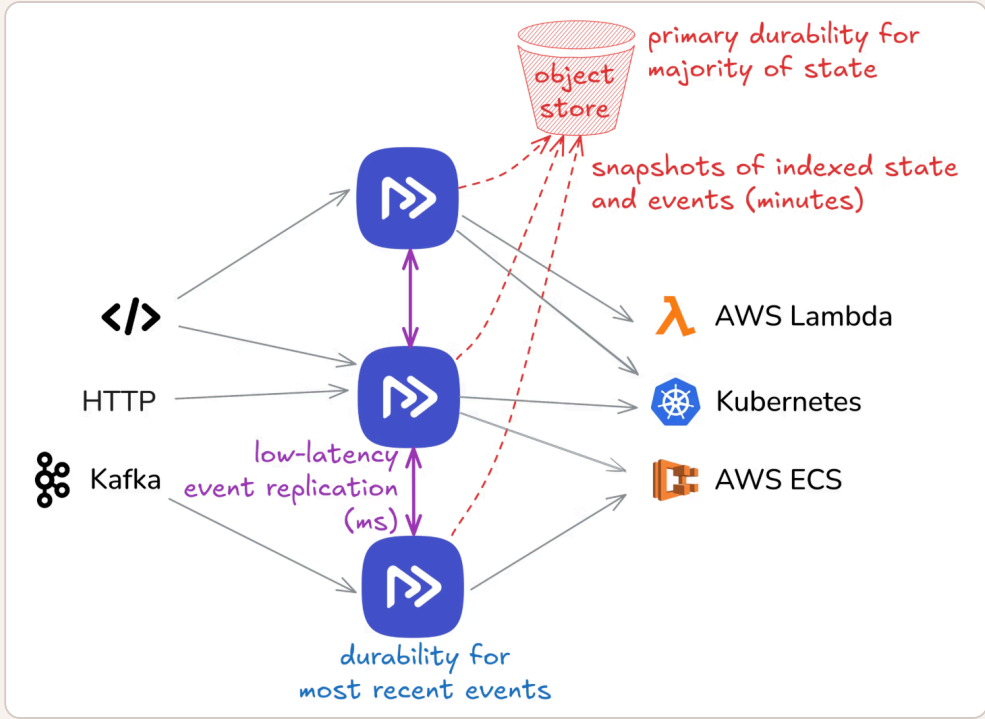
Temporal — Dedicated cluster

- **4 internal services** — Frontend, History, Matching, Worker
- **Separate worker processes** — your code runs in workers that poll the server
- **Cassandra, Postgres, or MySQL** for persistence
- **Horizontally scalable** — battle-tested at Netflix, DoorDash scale



Restate — Lightweight server

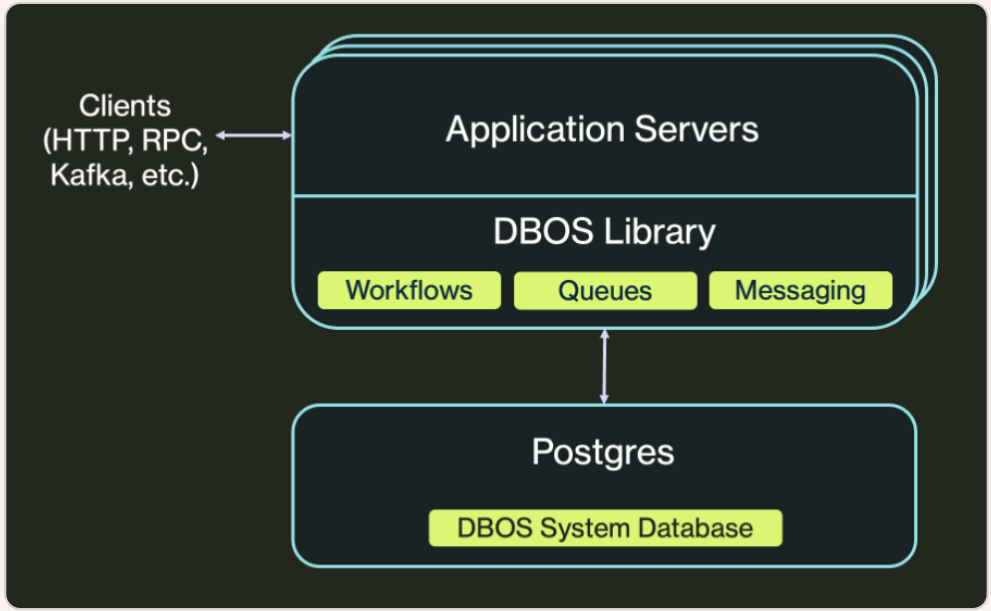
- **Single binary** with built-in event replication
- **Journal-based execution** — replays from an append-only log
- **Integrates with Lambda, K8s, ECS** — your code runs wherever you want
- **Object store durability** — snapshots state to S3-compatible storage



THE LANDSCAPE — EMBEDDED

DBOS — Embedded library

- **Library inside your app** — no separate server or cluster
- **Checkpoints to Postgres** — the database you already run
- **Workflows, queues, messaging** built into the library
- **Zero extra infrastructure** — simplest path to durable execution



THE LANDSCAPE

Why DBOS for today's demos

For this talk, I want the simplest setup that still shows the mechanics clearly.

- **Embeds into existing code** — enough to show the pattern without introducing a separate control plane
- **Uses the Postgres you already run** — keeps the demo setup simple
- **Available in Go, Python, TypeScript** — useful for showing the same ideas across languages
- **Open source** — easy to inspect how the runtime works

The mechanics matter more than the product choice. Checkpointing, replay, determinism, and durable messaging show up in different forms across the ecosystem.

Code and Demos

DEMO 1

Marketplace Checkout

The checkout scenario from earlier, implemented with durable execution.

- Linear function: reserve inventory → create order → wait for payment → dispatch
- `RunAsStep` wraps each database operation (checkpointed)
- `Recv` waits for payment confirmation from an external webhook
- `RunWorkflow` starts a child workflow for dispatch

This is still a **regular function**. The recovery logic lives in the runtime instead of being spread across the application.

DEMO 1 — MARKETPLACE CHECKOUT

Checkout workflow (Go)

```
    orderID, _ := dbos.RunAsStep(ctx, func(stepCtx context.Context) (int, error) {
        return createOrder(stepCtx)
    })
    // Step 2: Reserve inventory (checkpointed)
    success, _ := dbos.RunAsStep(ctx, func(stepCtx context.Context) (bool, error) {
        return reserveInventory(stepCtx)
    })
    // Step 3: Wait for payment — durably (survives crashes)
    payment, _ := dbos.Recv[string](ctx, PAYMENT_STATUS, 60*time.Second)
    if payment == "paid" {
        // Step 4: Update status (checkpointed)
        dbos.RunAsStep(ctx, func(stepCtx context.Context) (string, error) {
            return updateOrderStatus(stepCtx, orderID, PAID)
        })
        // Step 5: Start dispatch as child workflow
        dbos.RunWorkflow(ctx, dispatchOrderWorkflow, orderID)
    }
}
```

DEMO 2

AI Agent with Human Approval

Same `Recv` / `Send` pattern, now applied to AI agents.

- Agent does work → reaches a decision point
- Calls `DBOS.recv()` to wait for human approval
- `set_event` updates agent status (frontend polls this)
- HTTP handler calls `DBOS.send()` when the human responds

Same building block, different use case. `Recv` waits for payment in the store; `Recv` waits for human approval here.

DEMO 2

Durable agent workflow (Python)

```python
def durable_agent(request: AgentStartRequest):
    agent_status = AgentStatus(name=request.name, task=request.task, status="working")
    DBOS.set_event(AGENT_STATUS, agent_status)

    # Do some work...

    # Durable wait for human approval (survives crashes)
    agent_status.status = "pending_approval"
    DBOS.set_event(AGENT_STATUS, agent_status)
    approval = DBOS.recv(timeout_seconds=3600)

    if approval is None or approval.response == "deny":
        raise Exception("Agent denied or timed out")

    agent_status.status = "working"
    DBOS.set_event(AGENT_STATUS, agent_status)
    return "Agent successful"
```

DEMO 3

Access Management — Long-Running Approval Chain

Multi-approver workflow that spans days, with saga-pattern rollback.

- `Recv` per approver with 7-day timeout
- Denial at any step stops the chain
- On full approval → child `ProvisioningWorkflow`
- Saga pattern: if external provisioning fails, compensate by revoking local assignment

DEMO 3

Multi-step approval (Go)

```go
func ApprovalWorkflow(ctx dbos.DBOSContext, input ApprovalWorkflowInput) (Result, error) {
    // Load approval chain (checkpointed)
    steps, _ := dbos.RunAsStep(ctx, func(c context.Context) ([]ApprovalStep, error) {
        return loadApprovalSteps(c, input.RequestID)
    })
    // Wait for each approver (7-day timeout, survives restarts)
    for _, step := range steps {
        decision, err := dbos.Recv[Decision](ctx, "approval", 7*24*time.Hour)
        if err ≠ nil { /* timeout → auto-escalate */ }
        if decision == "DENIED" { return denied() }
    }
    // All approved → start child provisioning workflow
    handle, _ := dbos.RunWorkflow(ctx, ProvisioningWorkflow,
        ProvisioningInput{RequestID: input.RequestID})
    result, _ := handle.GetResult()
    return result, nil
}
```

DEMO 3 — SAGA PATTERN

Provisioning with rollback (Go)

```
    // Saga Step 1: Assign role in local database
    dbos.RunAsStep(ctx, func(c context.Context) (string, error) {
        return "ok", assignUserRole(c, input.UserID, input.RoleID)
    })
    // Saga Step 2: Provision in external system (3 retries)
    _, err := dbos.RunAsStep(ctx, func(c context.Context) (string, error) {
        return provisionExternal(input.UserID, input.RoleName)
    }, dbos.WithStepMaxRetries(3))
    if err ≠ nil {
        // Compensate: roll back Step 1
        dbos.RunAsStep(ctx, func(c context.Context) (string, error) {
            return "ok", revokeUserRole(c, input.UserID, input.RoleID)
        })
        return Result{Status: "FAILED"}, err
    }
    return Result{Status: "PROVISIONED"}, nil
}
```

Practical Guidance

GUIDANCE

When to use it — and when to skip it

Reach for DE when...

Scenario	Why
Payment + fulfillment	Crash = money taken, no delivery
Multi-step approvals	Must survive restarts across human-speed waits
AI agent orchestration	LLM calls cost money; retry wastes $ and context
Saga with rollback	Forward and compensating steps must both complete

Skip DE when...

Scenario	Use instead
Single-step CRUD	Direct DB call
Millisecond-fast ops	Inline code (checkpoint adds 1-5ms)
Database-local work	Single transaction or savepoints
Cheap-to-retry ops	Job queue (BullMQ, Celery)

Gut check: describe the work as a single verb -- *send, process, sync*? It's a job, use a queue. Need "and then" or "wait until"? It's a workflow.

GUIDANCE

Gotchas

1. Determinism discipline

`time.Now()`, `uuid.New()`, `rand.Intn()` must be inside checkpointed steps. Non-deterministic code outside steps causes replay mismatches — and these bugs only surface during crash recovery, exactly when you need the system to work.

2. Workflow versioning

Changing step sequences while instances are in-flight causes replay mismatches. This is the hardest operational problem in DE adoption — plan for it before your first long-running workflow ships.

3. Idempotency

Steps have at-least-once semantics. If a step succeeds but the checkpoint write fails, the step re-executes. External calls need idempotency keys.

4. Payload size

Every step's I/O is serialized to the database. Pass references (S3 URL, record ID), not large objects.

GUIDANCE

Complexity migrates — it never disappears

Durable execution doesn't reduce your system's total complexity. It **relocates** it.

You stop dealing with…	You start dealing with…
Hand-rolled retry loops at every call site	**Determinism discipline** — non-deterministic code must live inside steps
Custom state machines + polling tables	**Workflow versioning** — changing step sequences breaks in-flight instances
Dead-letter queues + manual reprocessing	**Idempotency boundaries** — external calls still need idempotency keys
Ad-hoc crash recovery scattered across services	**Payload awareness** — every step's I/O is serialized to the database

The trade: recovery logic stops being scattered across services and becomes explicit rules the team can review and enforce. You're not removing complexity. You're moving it.

The Coordination Landscape

COORDINATION

The distributed transaction problem

When a single operation spans multiple services, a local DB transaction can't help you.

- Each service has its **own database**
- A local DB transaction can’t span services
- External APIs don’t participate in DB transactions

Three families of solutions:
- distributed transactions (lock everything),
- sagas (compensate on failure),
- durable execution (orchestrate as a function).

COORDINATION

Two-Phase Commit (2PC)

A coordinator locks all participants, then commits atomically.

- **Blocking:** participants hold locks while waiting for the coordinator's decision
- **Fragile:** if the coordinator crashes between prepare and commit, all participants are stuck
- **Closed-world:** every participant must support the 2PC protocol (XA, PostgreSQL `PREPARE TRANSACTION`), but external APIs don't
- **Where it lives today:** inside distributed DBs (Google Spanner, CockroachDB), rarely across microservices

COORDINATION

Saga pattern

The practical alternative: break the transaction into local transactions, each with a compensating action.

- Each step is a **local transaction** – no distributed locks
- If step N fails, run **compensating transactions** for steps N-1 down to 1
- No atomicity guarantee – the system is **eventually consistent**
- Two coordination strategies: **choreography** and **orchestration**
- **Framework support:** Axon Framework (Java), MassTransit (.NET), Temporal, eventuate.io

COORDINATION — SAGA

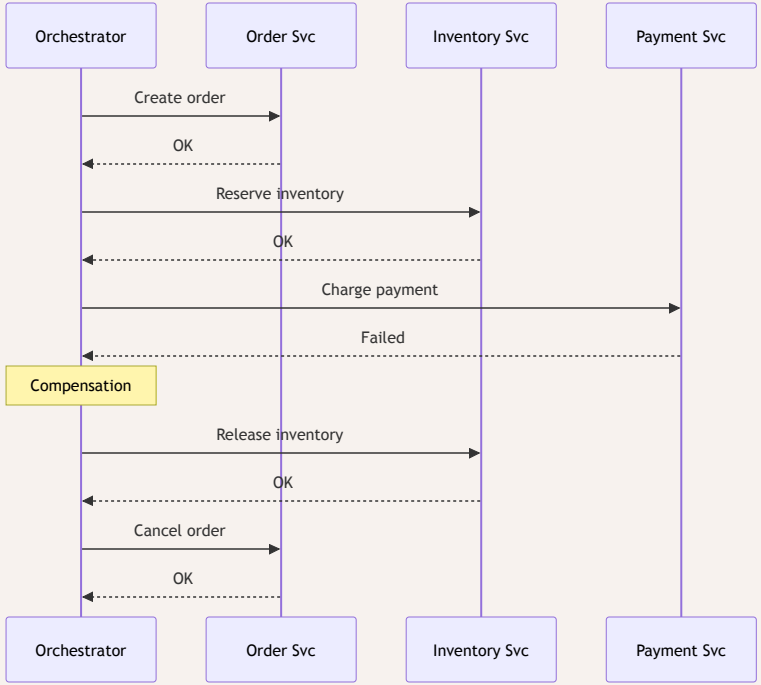
Choreography

Each service emits events. The next service listens and reacts. No central coordinator.

- **Decoupled:** services only know about events, not each other
- **Hard to trace:** the full flow is implicit, scattered across event handlers
- **Compensation is event-driven too:** failure events trigger reverse actions, but no one tracks overall status
- **Common brokers:** Apache Kafka, RabbitMQ, NATS, Redis Streams, AWS EventBridge

COORDINATION — SAGA

Orchestration



A central orchestrator drives the saga. It tells each service what to do and handles failures.

- **Visible:** the full flow lives in one place – the orchestrator
- **Debuggable:** one log stream, one state to inspect
- **But:** the orchestrator itself must be reliable – if it crashes mid-saga, the whole thing stalls
- **Tools:** Temporal, Camunda (BPMN), AWS Step Functions, Netflix Conductor (now Orkes)

COORDINATION

Where durable execution fits

DE is a **saga orchestrator where the orchestrator itself is crash-proof**.

- The orchestrator problem – *"what if it crashes?"* – is solved by checkpoint/replay
- Compensation is ordinary error handling inside a linear function
- Unlike traditional sagas, durable messaging enables **long waits** (human approvals, webhooks)
- **Implementations:** Temporal (server cluster), DBOS (library + Postgres), Restate (single binary), Inngest (serverless)

COORDINATION

Orchestration vs choreography

Not a competition – they compose.

	When to reach for it	Sounds like
Orchestration (DE)	Steps depend on each other	*"Do A, then B, then C. If C fails, undo B and A."*
Choreography (Kafka)	Steps are independent	*"Something happened; whoever cares can react."*

Same event, different coordination needs. Kafka distributes. DE orchestrates.

COORDINATION

How these patterns compose in production

A food delivery order touches multiple coordination strategies:

Step	Pattern	Why
Event published atomically	**Outbox + CDC**	DB write and event must be atomic
Reserve, charge, dispatch	**DE workflow** (saga)	Survives crashes; no double charges
↳ Payment gateway call	**Circuit breaker + idemp. key**	DE retries alone hammer a degraded service
Send email, update analytics	**Choreography** (Kafka)	Subscribe to order event – no new publish
Generate receipt PDF	**Job queue** (BullMQ)	Loss is tolerable – a reconciliation job catches misses

No single pattern covers the full flow. The skill is knowing which to reach for at each layer.

COORDINATION

What DE handles vs. what you still own

Pattern	Status	Detail
Retry with backoff	Absorbed	Built-in step config – `WithStepMaxRetries(3)` with exponential backoff.
Idempotent consumer	Mostly absorbed	Steps replay from checkpoint. External calls still need idempotency keys.
Circuit breaker	**Not absorbed**	Needed at the HTTP client level – e.g., `gobreaker` (Go), `opossum` (Node).
Timeout / bulkhead	**Not absorbed**	Context deadlines, connection pools – below the workflow layer.

A real failure: Grab Food order. A workflow step charges RM 45.00 via a payment gateway. Gateway processes the charge, but the response times out. DE retries. Gateway charges again -- **RM 90.00 debited**. Without an idempotency key, the retry creates a duplicate. Without a circuit breaker, retries pile up against a struggling service. **The patterns must compose.**

COORDINATION

Choosing the right tool

You need to...	Reach for	Not
Atomic commit across DBs	**2PC** (within a DB cluster)	2PC across services (fragile)
DB write + event publish atomically	**Outbox + CDC**	Dual writes (DB then Kafka)
Multi-step process with rollback	**Durable execution** (saga built in)	Custom state machine + polling
Independent reactions to events	**Choreography** (Kafka)	Orchestration (unnecessary coupling)
Single-step, loss-tolerable work	**Job queue** (BullMQ, Celery)	DE (overkill for reconciliation jobs)
Full audit trail of state changes	**Event sourcing**	Mutable rows with `updated_at`

Common mistake: reaching for DE when a job queue would do, or hand-rolling a state machine when DE would save weeks.

Write workflows as ordinary functions. The runtime makes them survive crashes, restarts, and days-long waits.

RESOURCES

Learn more

Resource	Link
DBOS	dbos.dev — open source, docs, quickstart
Temporal	temporal.io — the category leader
Restate	restate.dev — lightweight alternative
Inngest	inngest.com — serverless-native

The core mechanics to look for:

Checkpointing. Replay. Determinism. Durable messaging.

The tools change. These four ideas don’t.

Questions?

Thank you