Temporalizing Transaction Processing: A Developer’s Resilience Journey

How we transformed a fragile, monolithic system into into a durable, observable, and scalable workflow.

The Mission: Processing High-Stakes Financial Transactions

We had a critical financial transaction processing system with a demanding set of requirements:

AI-Powered Analysis: Process transactions through multiple AI analysis steps.

Fraud Detection: Utilize vector similarity search for sophisticated fraud detection.

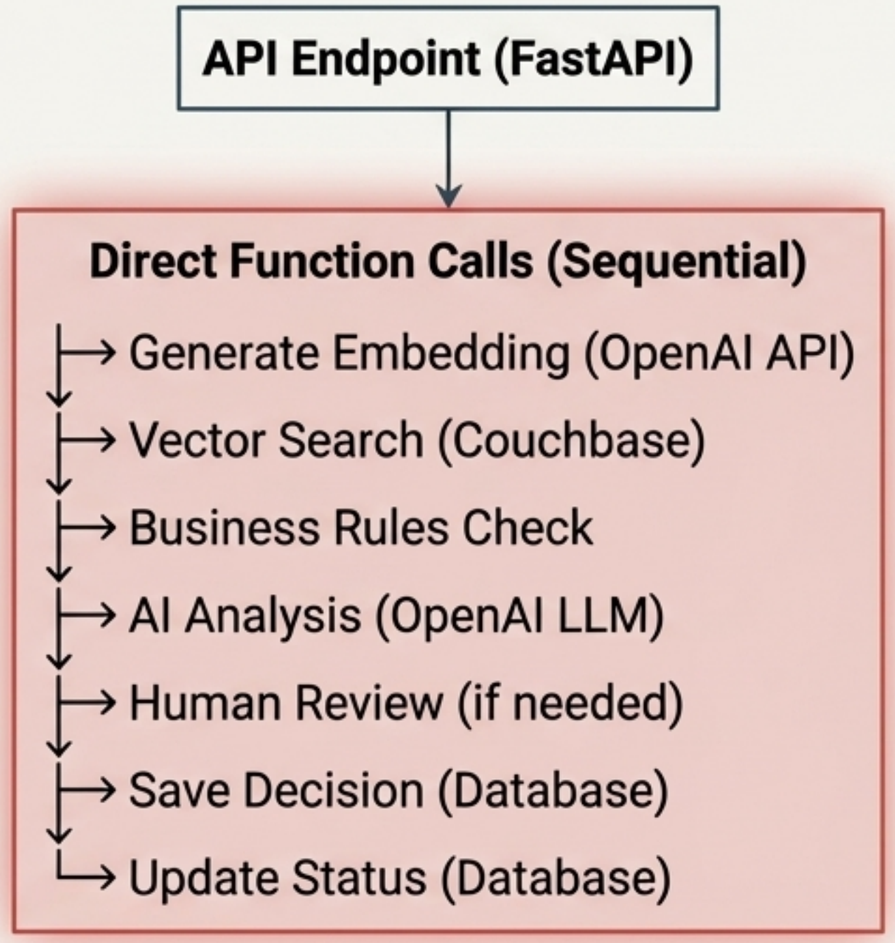
Rule Enforcement: Apply a complex set of compliance and business rules.

Human Oversight: Support human-in-the-loop review workflows for edge cases.

Data Integrity: Ensure absolute data consistency and reliability for every transaction.

Our Original Architecture Was a Synchronous Monolith

Before: The Monolithic Approach



Key Characteristics

- **Synchronous Execution:** All steps ran sequentially within a single request.
- **No State Persistence:** If the process failed mid-way, all progress was lost.
- **Manual Retries:** Custom, brittle retry mechanisms had to be built and maintained.
- **No Visibility:** Difficult to track where a transaction was in the pipeline.
- **Tight Coupling:** All logic was in one place, making it hard to test and maintain.

This Architecture Created Six Points of Critical Failure

The design led to a cascade of recurring, painful problems:

- ❌ **Long-running requests:** Transactions took 30-60 seconds, causing constant API timeouts.
- ❌ **Manual state tracking:** We had to manually track transaction state in the database, risking inconsistencies.
- ❌ **No recovery:** If a server crashed mid-process, the transaction was permanently lost.
- ❌ **Retry complexity:** Implementing our own retries required complex, error-prone state machines.
- ❌ **Difficult debugging:** No way to see which step failed or why, leading to painful investigations.
- ❌ **Blocking human review:** The system had to inefficiently poll the database to see if a human review was complete.

A Single Timeout Could Erase an Entire Transaction

The lack of durability was not theoretical. If an external API call failed, the state was unrecoverable and the entire transaction was lost. There was no way to resume from the point of failure.

```
# Before: If the OpenAI API timed out, the transaction was lost forever.
try:
    embedding = openai_client.generate_embedding(text) # <--- FAILS HERE
    similar = search_similar(embedding)                # Never runs
    decision = analyze(transaction)                    # Never runs
    save_decision(decision)                            # Never runs
except Exception:
    # Transaction is lost. There is no way to recover or resume.
    pass
```

We Needed a System Built for Durable, Long-Running Work

The transaction pipeline's characteristics made it a perfect fit for a workflow-based approach. We chose Temporal because our process required:

1. **Multi-step orchestration** with complex dependencies.
2. **Calls to external services** (OpenAI, Couchbase) prone to transient failures.
3. **Long-running operations** (AI analysis) that exceeded API timeouts.
4. **Human-in-the-loop** steps that needed to wait for days without polling.
5. **Critical operations** where failure was not an option.
6. **Complete audit history** for compliance and debugging.

Deconstructing the Monolith into Workflows and Activities

The migration strategy involved two key steps:

Step 1: Isolate Side Effects into Activities.

We identified all operations with external dependencies (API calls, I/O) and defined them as independent, retryable Activities.

Step 2: Orchestrate Logic in a Workflow.

The core business logic, sequencing, and state management were moved into a durable Workflow, which became the 'brain' of the operation.

Identified Activities

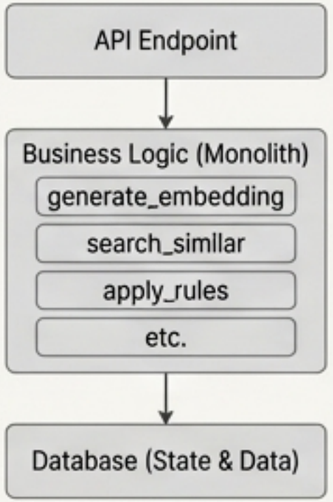
- `generate_embedding`
- `search_similar_transactions`
- `apply_business_rules`
- `analyze_transaction_with_ai`
- `save_decision`
- `update_transaction_status`
- `create_human_review`

The Workflow's Role

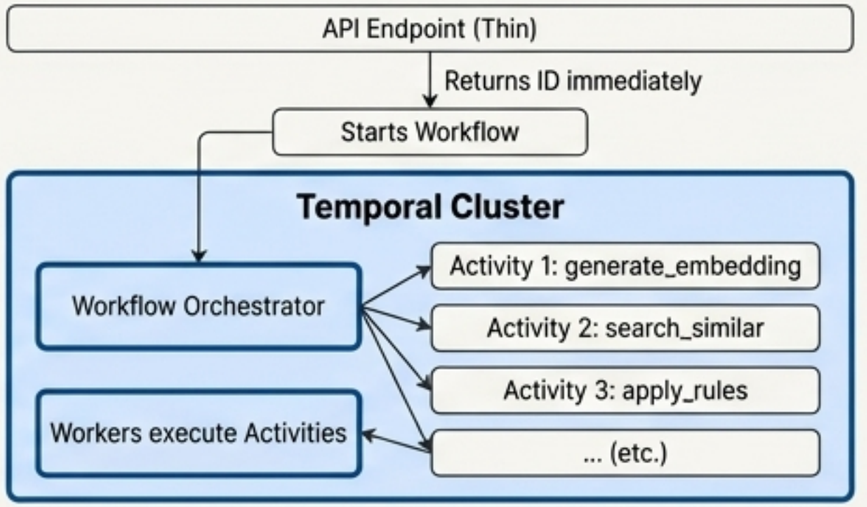
- Orchestrates Activities
- Manages state
- Handles conditional logic
- Provides durability

The Result: A Clean Separation of Concerns

Before



After



Key Changes

- **API becomes thin:** Just starts the workflow and returns immediately.
- **Workflow orchestrates**: All business logic now resides in the durable workflow.
- **Activities are isolated**: Each step is independent, testable, and retryable.
- **State managed by Temporal**: No more manual state tracking in our database.

Improvement 1: From Manual Retries to Automatic Resilience

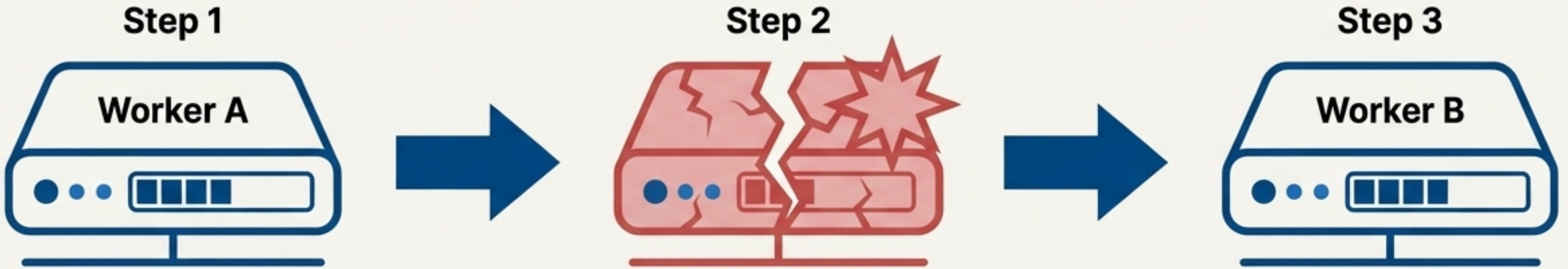
Instead of writing complex retry logic, we now declare a policy. Temporal automatically retries failed Activities with exponential backoff, handling transient network or API errors without any custom code.

```
# After: Temporal automatically retries with a configurable policy.
await workflow.execute_activity(
    generate_embedding,
    args=[transaction_data],
    start_to_close_timeout=timedelta(seconds=30),
    retry_policy=RetryPolicy(
        initial_interval=timedelta(seconds=1),
        backoff_coefficient=2.0,
        maximum_attempts=3
    )
)
```

✅ Automatic Retries ✅ Exponential Backoff ✅ Configurable Per Activity

Improvement 2: From Lost Work to Guaranteed Durability

If a worker crashes mid-execution, the transaction is no longer lost. Temporal automatically persists the workflow's state and history, allowing a new worker to pick up exactly where the last one left off.



- ✅ Zero Data Loss
- ✅ Automatic Recovery
- ✅ Complete Audit Trail

Improvement 3: From Inefficient Polling to Event-Driven Signals

Before: Manual Polling

```
# Inefficient, blocks worker, no timeout
while True:
    review = db.get_review(transaction_id)
    if review.status == "completed":
        break
    time.sleep(5) # Wastes resources!
```

After: Durable Wait with Signal

```
# Event-driven, durable, with a built-in timeout
await workflow.wait_condition(
    lambda: self._human_review_signal_received,
    timeout=timedelta(days=7)
)
# If timeout occurs, defaults to reject.
```

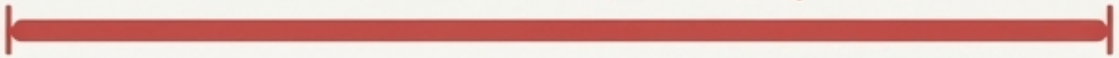
✅ Event-driven (no polling)
✅ Durable wait state
✅ Automatic timeout handling

Improvement 4: From Blocking APIs to Immediate Responses

Our API no longer waits for the 60-second process to complete. It now starts the workflow and returns a workflow ID in milliseconds, freeing up server resources and providing a vastly better user experience. The client can then query the workflow status asynchronously.

Before: Blocks for 60s

❌ API request timed out!



```
@app.post("/transaction")
async def process():
    # This takes 60 seconds - API times out!
    result = await long_running_analysis()
    return result
```

After: Returns Immediately

✅ Response < 100ms

```
@app.post("/transaction")
async def process():
    # Returns in milliseconds
    handle = await client.start_workflow(...)
    return {"workflow_id": handle.id}
```

✅ **Fast API responses** ✅ **Better UX** ✅ **Higher scalability**

The Engineering Realities: Navigating the Trade-offs

Adopting a workflow engine involves deliberate design choices. Here are the key considerations we navigated:

Eventual vs. Immediate Consistency

Trade-off:
The API returns success before the transaction is fully processed.

Mitigation:
We return a workflow ID for status polling and use webhooks for final notification.

Workflow Determinism

Trade-off:
Workflow code cannot contain non-deterministic calls (e.g., `datetime.now()`, random numbers, external APIs).

Solution: All such operations are moved into Activities. We use `workflow.now()` for deterministic time.

Additional Infrastructure

Trade-off:
Requires running and maintaining a Temporal server cluster.

Solution:
We use Docker for local development and Temporal Cloud for managed production infrastructure. The reliability gains far outweigh the cost.

Onboarding Developers: From Core Concepts to Advanced Features

We teach the new system using a progressive approach, backed by comprehensive documentation.

Teaching Approach

- **Level 1: Core Concepts**
  Workflows, Activities, and Workers.
- **Level 2: Our Application**
  The Before vs. After architecture.
- **Level 3: Advanced Features**
  Retry Policies, Signals, and Queries.
- **Level 4: Live Demo**
  Walk through a real execution in the Temporal UI.

Key Teaching Points

1. **Determinism is Critical:** Use `workflow.now()`.
2. **Activities are for Side Effects:** Isolate all I/O.
3. **State is Automatic:** Let Temporal manage persistence.
4. **Retries are Automatic:** Just configure the policy.
5. **Timeouts Prevent Hanging:** Set them for every Activity.

The Transformation: From a Fragile Pipeline to Durable Orchestration

Before Temporal	After Temporal
✗ Manual state management	✓ Automatic state persistence
✗ Lost work on failures	✓ Guaranteed execution completion
✗ Complex, custom retry logic	✓ Built-in, declarative retry policies
✗ Blocking API calls	✓ Non-blocking async execution
✗ Hard to test and debug	✓ Natively testable code
✗ No visibility into execution	✓ Complete observability via UI

The result is a production-ready system that handles failures gracefully, scales horizontally, and provides complete visibility into every transaction.