When to Event Source, and When Not To

Guide to Domain Events vs. Observations

AXONIQ

Marc Klefter

Executive Summary

TL;DR

Not everything belongs in an event store. Deciding what to capture is just as important as deciding what not to capture. Your team should event-source business decisions that need permanent audit trails, such as orders, payments, and state changes. However, you should not event-source high-frequency measurements like sensor readings, GPS pings, or log data—stream those to analytics platforms instead, or aggregate them into meaningful thresholds and translate them into business decisions before they touch your event store. The distinction prevents bloated event stores, performance problems, and runaway cloud costs.

Event sourcing is a pattern where you store every state change as an immutable event, then reconstruct the current state by replaying those events. Unlike traditional databases that store only current state, event sourcing keeps the complete history.

When to use event sourcing:

- You need complete audit trails for compliance (financial services, healthcare)
- You must reconstruct past states for disputes or analysis
- Business decisions need to be explained years later
- Regulatory requirements demand immutable history

When not to use event sourcing:

- You're capturing high-frequency measurements (IoT sensors, GPS, logs)
- Current state is all that matters (shopping cart contents, user preferences)
- Events have no business meaning beyond analytics or notifications
- Your team lacks experience with event-driven architecture patterns
- Simple CRUD operations satisfy all requirements

The problem? Most teams don't make this distinction clearly, leading to architectural decisions that create more problems than they solve.

The Core Problem: Your Team Doesn't Know Which Events Matter

Here's the question that should drive every event sourcing architecture decision:

"Will we need this exact event to make business decisions or satisfy compliance requirements five years from now?"

If yes: event source it. If no: probably don't.

But most teams never ask this question. They default to, "we're doing event sourcing, so everything goes in the event store" or "event sourcing is too complex, so nothing does."

Both approaches fail. The first creates unmaintainable systems that collapse under data volume. The second leaves you unable to reconstruct the state when auditors, regulators, or customer disputes demand it.

Event Sourcing Best Practices: The Two-Event Classification Framework

Train your team to classify every event as one of two types:

Domain Events: Business Decisions That Already Happened

These capture facts your business permanently cares about:

- "OrderPlaced"
- "PaymentProcessed"
- "BikeRented"
- "UserAccountClosed"
- "LoanApproved"

Domain event characteristics:

- Represent completed business transactions or state changes
- Required for compliance, audit trails, or dispute resolution
- Low to moderate frequency (typically < 10 per second per entity)
- Kept for years, sometimes indefinitely
- Used to reconstruct decision models for business logic

Event sourcing best practice: These belong in your event store.

Observations: Raw Signals From Systems or Sensors

These capture measurements and monitoring data:

- “GPSCoordinateUpdated”
- “TemperatureSensorReading”
- “PageViewRecorded”
- “APIResponseTimeLogged”
- “InventoryLevelChecked”

Observation characteristics:

- Snapshots of current state, not business decisions
- Individually meaningless; valuable only when aggregated
- High frequency (hundreds to millions per second)
- Short retention period (days to months)
- Fed to analytics platforms, not used for business logic

Event sourcing best practice: Don’t event source these. Stream to data lakes instead.

What Not to Event Source: Real-World Failure Modes

Failure Mode 1: The $400K Event Store Bloat

A fintech company event sourced every market data tick alongside actual trades. Their reasoning: “We’re event-driven, and these are events.”

The damage:

- Event store grew from 2TB to 47TB in nine months
- Constructing decision models went from ms to multiple seconds
- Three production incidents where timeout cascades took down order processing
- $180K in emergency storage costs
- $220K in engineering time to migrate 94% of events to a data lake

The fix: Event source trades and order state changes. Stream market data directly to their analytics platform. Build a translation layer that watches for conditions (“price drops 10% in 5 minutes”) and triggers domain events (“TradingHalted”).

Their event store is now 2.8TB. Rehydration is back to ms. No more incidents.

Failure Mode 2: The Missing Audit Trail

A healthcare platform treated prescription fulfillment as an “observation” because it happened in a third-party system. When regulators asked for a complete audit trail of a patient’s medication history, they couldn’t reconstruct it.

What went wrong: The domain events that mattered—“PrescriptionDispensed,” “RefillApproved”—had never been captured. They only had aggregated reports, not the immutable event history compliance required.

Failure Mode 3: The Real-Time Analytics Illusion

An IoT company built real-time alerting by querying their event store for every sensor reading. They thought event sourcing meant instant analytics.

The problem: Event stores are optimized for writes and decision model reconstruction, not ad-hoc queries across millions of sensor readings. Their p99 latency for “temperature above threshold” alerts was 40 seconds. Useless for operational decisions.

Domain Events vs Integration Events: Understanding the Distinction

Domain events are internal to a bounded context. They capture what happened within your business domain and are stored in your event store for state reconstruction.

Integration events (sometimes called “public events”) are published to external systems. They’re how bounded contexts communicate in an event-driven system.

The critical distinction:

- Domain events: Internal, fine-grained, optimized for event sourcing
- Integration events: External, coarse-grained, optimized for system integration

Event sourcing best practice: Don’t expose domain events directly to other systems. Instead, create integration events that:

- Contain only necessary information (not internal implementation details)
- Use stable contracts (domain events can evolve more freely)
- Are designed for consumers, not for your internal reconstruction needs

Example: Your “BikeRented” domain event might contain 15 fields for internal state management. Your “BikeRentalStarted” integration event might contain only 5 fields that external systems need.

Event-Driven Architecture Patterns: The Translation Layer

This is where most architecture diagrams lie by omission. They show “Analytics” connected to “Operations” with a clean arrow and move on.

That arrow is doing critical work in event-driven architecture patterns:

Pattern 1: Pull-Based Translation (Start Here)

Perfect for most use cases where sub-second latency isn't required.

How it works:

1. Create read model projections from your domain events (bikes currently rented, active user sessions)
2. Set up scheduled jobs (every 30 seconds, every 5 minutes—whatever makes business sense)
3. Query both projections and observation data stores
4. When conditions match, issue commands back to your operational domain

Example: Check every 2 minutes if any rented bikes are outside the geofence. If yes, send "LockBike" command.

Complexity: Low. Any mid-level engineer can build this with cron + SQL.

When to use: 80% of operational analytics needs.

Pattern 2: Push-Based Translation (When Latency Matters)

For scenarios where delays create actual business risk or customer experience problems.

How it works:

1. Stream integration events to your data platform (Kafka, Kinesis)
2. Stream observations directly to the same platform
3. Use stream processing (Flink, Databricks workflows) to join and evaluate conditions in real-time
4. Trigger webhooks or API calls back to operational systems when rules match

Example: Detect fraudulent login patterns within 200ms to block account access before damage occurs.

Complexity: Medium-High. Requires streaming infrastructure expertise.

When to use: Security, fraud detection, safety-critical systems where seconds matter.

Pattern 3: Hybrid Translation (Common in Practice)

Combine both approaches:

- Real-time push for critical alerts (fraud, safety violations)

- Scheduled pull for operational insights (capacity planning, trend analysis)
- Batch processing for historical analysis (monthly reports, ML training)

Event Sourcing Architecture Decisions: The Decision Framework

When your engineers propose event sourcing something, have them answer these questions:

Question 1: Is This a Decision or a Measurement?

- Decision (“user purchased subscription”) → Domain event, event source it
- Measurement (“CPU at 73%”) → Observation, stream to data lake

Question 2: Who Needs This Data?

- Business logic, compliance, auditors → Domain event, belongs in event store
- Analytics, ML models, dashboards → Observation, belongs in data platform

Question 3: What’s the Frequency Per Entity?

- < 10 per second → Could be domain event
- 100+ per second → Almost certainly observation

Question 4: How Long Does This Matter?

- Years → Domain event (permanent audit trail)
- Days to months → Observation (temporary analytical value)

Question 5: Does This Reconstruct State?

- Yes (“I need all payments to calculate account balance”) → Domain event
- No (“I need recent page views to show trending content”) → Observation

Question 6: What's the Business Impact of Losing It?

- High (can't process orders, compliance failure) → Domain event
- Low (missing analytics data, dashboard gaps) → Observation

Event Sourcing Best Practices: Architecture Implementation

For Domain Events:

- ✅ Store in event store (Axon Server)
- ✅ Retain per compliance requirements (often years or indefinitely)
- ✅ Expose through read model projections, never directly expose domain events
- ✅ Use for decision model reconstruction and subsequent business decisions
- ✅ Design for internal consumption first
- ❌ Don't expose directly to external systems (use integration events)
- ❌ Don't use for high-frequency analytics queries

For Observations:

- ✅ Stream to data lake/warehouse (S3 + Athena, Snowflake, Databricks)
- ✅ Retain based on analytical value (usually 30-180 days)
- ✅ Push raw—no translation needed before storage
- ✅ Aggregate before using (individual readings rarely matter)
- ✅ Optimize for query patterns and analytics workloads
- ❌ Don't event source in your operational event store
- ❌ Don't use directly for business logic decisions

For Translation Layer:

- ✅ Queries projections from event store
- ✅ Queries observation aggregations from data platform
- ✅ Evaluates business rules combining both
- ✅ Issues commands that create new domain events
- ✅ Can be simple (cron + SQL) or sophisticated (stream processing)
- ❌ Don't make it more complex than business requirements demand

Event-Driven Architecture Patterns: SQL Analytics Over Event Stores

Here's a capability that solves the "how do I query my event store for analytics?" problem without impacting production performance.

Axoniq Analytics replicates your event store to a format optimized for analytical operations. Your team can write queries like:

```
-- How many bikes are currently rented?
SELECT COUNT(*) as active_rentals
FROM bike_events
WHERE event_type = 'BikeCheckedOut'
AND bike_id NOT IN (
  SELECT bike_id FROM bike_events
  WHERE event_type = 'BikeReturned'
)
```

This query hits a separate analytical store. Zero impact on your operational event store performance.

Why this matters for event sourcing architecture decisions:

- No new infrastructure: Engineers don't need Kafka, Spark, or a data warehouse for basic analytics
- Familiar tooling: Any engineer who knows SQL can query domain events
- Fast iteration: Validate business logic during development
- BI tool integration: Postgres connector means existing dashboards work immediately

What This Means for Your Monday Sprint Planning

Immediate actions:

1. Identify translation gaps: Where are you making decisions based on observations without properly contextualizing them with domain events?
2. Set up the framework: Make “domain event vs. observation” part of every architecture review.

The Team Capability You’re Actually Building

This isn’t just about technical architecture. You’re teaching your team to think about:

- Data lifecycle management: Not all data has the same retention or access patterns
- Appropriate tooling: Event stores, data lakes, and stream processors solve different problems
- Separation of concerns: Operational systems and analytical systems have different requirements
- Business context: Technical decisions should map to business value

Teams that internalize this distinction ship features faster (they’re not fighting their architecture), scale more efficiently (right tool for each job), and handle compliance with confidence (complete audit trails for what matters).

Teams that don’t end up with data gravity problems, spiraling costs, and brittle systems that break when usage patterns change.

Key Takeaways: Event Sourcing Architecture Decisions

Event source when:

- You need permanent audit trails for compliance or disputes
- The event represents a completed business decision
- You'll need to reconstruct this state in the future
- Frequency is manageable (< 10/second per entity)
- The event has clear business meaning

Don't event source when:

- It's a high-frequency measurement or sensor reading
- Only current state matters (not the history)
- The event is purely for analytics, not business logic
- Volume would cause performance or cost problems
- It's better suited for a time-series database or data lake

Build translation layers that:

- Combine domain events with observations
- Evaluate business rules in the appropriate context
- Generate new domain events when conditions are met
- Start simple (pull-based) and evolve to real-time only when needed

Store Your Domain Events Where They Actually Belong

Once your team has identified which events truly matter—the business decisions, state changes, and compliance-critical facts—you need infrastructure that's purpose-built for event sourcing, not repurposed from other patterns.

Axon Server: Best-of-Breed Event Storage for Domain Events

Axon Server is designed specifically for the event sourcing pattern. Unlike general-purpose databases or message brokers pressed into event store duty, it's optimized for:

- Immutable append-only storage with built-in clustering and replication
- Event versioning and schema evolution that doesn't break your system when business requirements change
- Multi-tenancy and context isolation so different bounded contexts don't interfere with each other
- Horizontal scalability that grows with your domain event volume without performance degradation

Where other teams spend months building event store infrastructure on top of Postgres or Kafka, Axon Server gives you production-ready event sourcing out of the box. Your team focuses on business logic, not infrastructure plumbing.

Axoniq Insights: Enterprise-Grade Analytics for Event-Sourced Systems

Here's where the "don't query your operational event store for analytics" guidance meets reality. You need analytics over your domain events without impacting production. Axoniq Insights makes this possible.

Line-by-Line Traceability for Compliance and Debugging

Every state change, business rule evaluation, and decision is traceable back to the exact sequence of domain events that caused it. When auditors ask "why did this account get flagged?" or engineers debug "how did this order end up in this state?", you have complete answers, not approximations from aggregated logs.

Built for real-world schema evolution, audit trails, and regulatory traceability. Your domain events evolve as your business evolves, and Insights tracks that evolution without losing historical context.

AI-Powered Insight Agent for Business Users and BI Teams

Business analysts and product managers can query your event store in natural language:

- “How many users completed checkout after abandoning their cart in the last 30 days?”
- “Show me all loan applications that were approved then declined within 48 hours.”
- “What’s the average time between BikeRented and BikeReturned events by city?”

The Axoniq Insights agent generates SQL from a natural language question, executes against the analytically-optimized, replicated event store, and returns results, always with links back to the raw events. Traceable, auditable, and explainable.

No more “the data team will get back to you in two weeks.” Business users get answers in seconds, engineers verify the underlying events in minutes.

Built for LLMs and AI Agents to Reason Over Your Domain

Axoniq Insights is designed for AI-native workflows:

- Reusable views like cohorts, funnels, and root-cause timelines that agents can query programmatically
- Temporal reasoning support: “before,” “after,” “when,” “while”—queries that match how humans think about time
- Structured context that LLMs can request, filter, and reason over without hallucinating

Your AI agents can pull precise backend context—actual events, actual state changes—instead of generating plausible-sounding nonsense based on training data.

Designed for Scale, Governance, and Security

Immutable events with full audit trail across services: Every domain event, every projection update, every query—logged and traceable. When compliance audits happen, you have answers.

Keeps sensitive business logic in your stack: Unlike piping your entire event history to external LLM APIs, Axoniq Analytics runs in your infrastructure. Your business logic, your customer data, your competitive intelligence stays private. The LLM sees only what you explicitly share.

Optimized for distributed systems: Track causality across microservices, correlate events across bounded contexts, understand what triggered what, even when decisions span multiple systems.

The Complete Event Sourcing Stack

For operational domain events:

- Store in Axon Server for millisecond decision model reconstruction
- Retain indefinitely per compliance requirements
- Use for business logic and state management

For analytical queries over domain events:

- Replicate to Axoniq Analytics (zero production impact)
- Query with SQL or natural language
- Build dashboards, reports, and AI-powered insights

For high-volume observations:

- Stream to your data lake (S3, Snowflake, etc.)
- Keep Axon Server focused on business-critical domain events
- Join observation data with domain event projections in your translation layer

This architecture scales. A financial services company processes 50M domain events per day through Axon Server with p99 latency under 100ms. Their analytics team queries those same events through Insights without affecting production. Their fraud detection system combines real-time transaction observations with event-sourced account history through a translation layer that catches suspicious patterns in under 200ms.

That's the difference between infrastructure that fights you and infrastructure that enables you.

Common Questions About When to Use Event Sourcing

Q: Can I event source some aggregates but not others?

Yes. Event sourcing is a pattern you apply selectively. Event source aggregates where you need audit trails and state reconstruction (Orders, Accounts, Inventory). Use traditional persistence for aggregates where current state suffices (User Preferences, Cache Data).

Q: What if I need both real-time analytics AND historical audit trails?

Event source the domain events in your event store. Stream those same events (or integration event representations) to your analytics platform. You get both: immutable history for compliance and optimized analytics queries.

Q: How do I handle observations that occasionally become important?

Design your translation layer to create domain events when observations cross business-meaningful thresholds. Example: Temperature readings are observations. “TemperatureThresholdExceeded” is a domain event worth event sourcing.

Q: Should I event source everything “just in case”?

No. This creates performance problems, cost overruns, and operational complexity. Event source based on clear business requirements: compliance, audit trails, state reconstruction, dispute resolution. Not “maybe we’ll need this someday.”

Q: What’s the retention policy for domain events vs observations?

Domain events: Years or indefinitely (driven by compliance requirements). Observations: Days to months (driven by analytical value and storage costs). The translation layer bridges these different lifecycles.

Q: How do I migrate from event sourcing everything to this pattern?

Ideally you will not end up in this situation to begin with, given the aforementioned guidance.

However, if this is indeed the case, start by auditing your current event store. Identify high-frequency observations. Remove or migrate them to a data lake. Keep domain events in the event store. Build a translation layer to connect the data lake and the event store.

About Axoniq

Engineering teams building distributed systems ship faster when they stop managing infrastructure and start owning outcomes. Axoniq provides the event sourcing infrastructure that makes complex workflows visible, traceable, and operationally manageable, consolidating event store, message routing, and service coordination into a single, production-hardened system.

With complete causal history for every decision, teams debug distributed behavior in minutes, enforce correct architectural patterns by default, and build AI-ready systems without compliance surprises.

The result: engineering organizations that deliver faster, scale without proportional cost increases, and turn regulatory requirements into a competitive advantage they already own.