International Journal of Research Publications in Engineering, Technology and Management (IJRPETM)

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 8, Issue 4, July-August 2025||

DOI:10.15662/IJRPETM.2025.0804023

Event-Driven Architecture in Long-Running Enterprise Validation Workflows

Makarand Gujarathi

Independent Researcher, USA

ABSTRACT: Long-running enterprise validation workflows — vendor onboarding, KYC, claims adjudication, regulatory filings, procure-to-pay — routinely span hours to weeks, cross multiple systems, and involve dozens of validation steps whose outcomes cannot be predicted in advance. Request/response and synchronous orchestration models buckle under these properties: they hold connections open across human waits, they retry timeouts as if they were failures, and they force retry semantics onto business steps that were never designed to be idempotent. Event-Driven Architecture (EDA), paired with durable execution, is the pattern that these workflows structurally require, not merely a performance optimization. This article treats EDA in the specific context of long-running validation workflows and identifies the architectural elements that distinguish a system that survives production from one that quietly corrupts state. We define five architectural building blocks — the event backbone, the workflow orchestrator, the validation-step service pattern, the compensation and saga path, and the observability surface — and derive their design constraints from the workflow properties. We then present an end-to-end reference architecture that composes the five blocks, illustrate it in a vendor-onboarding worked scenario, and discuss the trade-offs against synchronous orchestration and pure choreography alternatives. The intended reader is the enterprise architect or platform engineer who must decide whether to adopt EDA, how deeply to commit to durable execution, and where the operational cost of the resulting system will fall.

KEYWORDS: event-driven architecture, long-running workflows, validation workflows, event sourcing, saga pattern, durable execution, choreography, orchestration, enterprise integration

I. INTRODUCTION

Enterprise validation workflows are, by their nature, long-running. A new vendor onboarding may take three to fifteen business days as the workflow waits on tax-ID verification, banking-detail confirmation, sanctions screening, insurance-certificate review, credit-check callback, procurement-policy sign-off, and finally system provisioning. A claims adjudication may take days to weeks as it waits on document uploads, third-party appraisals, medical-record retrievals, and adjuster reviews. A regulatory filing may take hours to days as it waits on data-quality validations, second-line reviews, and external-agency acknowledgments. In every case the wall-clock duration is dominated not by compute but by waits — on humans, on external systems, and on validations whose completion is asynchronous by construction.

Two architectural properties follow directly from this. First, the workflow's state must be durable across arbitrarily long periods and must survive the failure or restart of any single component. Second, the workflow's progress must be event-driven — advanced by the arrival of validation outcomes rather than by a caller polling for completion. Synchronous orchestration models violate both properties. A caller that opens a connection and waits for a validation to complete either times out or holds resources for hours, and a system that retries a timed-out validation as if it were a failure produces duplicate work, duplicate external calls, and — in the worst case — duplicate business commitments. Systems designed around request/response semantics can be forced to serve long-running validation workflows through polling loops, callback URLs, and ad hoc state tables, but the resulting architecture accumulates the failure modes of every ad hoc mechanism and eventually collapses under a workflow it was not designed to serve.

Event-Driven Architecture in this regime is not merely a performance choice; it is the pattern that the workflow's structure demands. The workflow is naturally decomposed into steps that produce and consume events (a validation completed, a document received, a review approved, a compensation required); the state of the workflow is naturally represented as the accumulated effect of these events on a durable log; the workflow's progress is naturally advanced by event arrival rather than by caller polling. EDA is, in short, the architecture that fits the workflow's structural properties, and the systems that succeed with long-running validation workflows are the systems that adopt EDA deliberately rather than the systems that arrive at it by accident.

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 8, Issue 4, July-August 2025||

DOI:10.15662/IJRPETM.2025.0804023

The pattern is not new. The BPM (Business Process Management) literature has recommended event-driven decomposition for two decades. Event sourcing, saga patterns, and durable execution are well-established in the industry. What is new — and what motivates this article — is the specific combination of demands that modern enterprise validation workflows now place on the pattern: mixed synchronous and asynchronous validation steps within the same workflow instance; regulatory requirements for auditable state history at each step; integration with AI-agent components whose completion semantics are probabilistic; and observability requirements that must expose the state of every in-flight workflow instance to on-call engineers who did not design the workflow. Building EDA correctly for these workflows requires attention to five architectural blocks whose design constraints derive directly from the workflow properties.

This article makes three contributions. First, it identifies the five architectural blocks that a production EDA implementation for long-running validation workflows must include, and derives the design constraints of each block from the workflow's properties rather than from generic EDA principles. Second, it presents an end-to-end reference architecture that composes the five blocks and shows how the blocks interact in a running workflow instance. Third, it illustrates the reference architecture in a vendor-onboarding worked scenario, and quantifies the operational trade-offs against the synchronous-orchestration and pure-choreography alternatives that enterprises typically consider first.

Section 2 reviews related work. Section 3 identifies the workflow properties that constrain the architecture. Section 4 presents the five architectural building blocks. Section 5 presents the end-to-end reference architecture. Section 6 walks through the worked scenario. Section 7 discusses trade-offs and limitations, and Section 8 concludes.

II. RELATED WORK

The architectural traditions that inform EDA for long-running validation workflows come from four directions. The first is the workflow-management and BPM literature, represented by van der Aalst's foundational work on workflow patterns, the BPMN 2.0 specification from OMG, and the more recent work on process choreography versus orchestration [van der Aalst 2013; OMG 2011]. This tradition provides the vocabulary — activities, gateways, events, compensations — but is largely silent on the runtime concerns of long-running execution.

The second tradition is event sourcing and command-query responsibility segregation (CQRS), represented by Fowler, Young, and the DDD community [Fowler 2005; Young 2010]. This tradition provides the durable-log discipline that a long-running workflow's state depends on. Event sourcing treats the workflow's state as the accumulated effect of an append-only event log, which gives the workflow the durability, auditability, and replay properties that regulated validation workflows require.

The third tradition is saga and compensation patterns, represented by Garcia-Molina and Salem's original saga paper, Richardson's more recent enterprise treatment, and the practical experience distilled by Kleppmann [Garcia-Molina & Salem 1987; Richardson 2018; Kleppmann 2017]. Sagas provide the compensation semantics that long-running workflows need in place of the distributed transactions that they cannot use, and they compose naturally with event sourcing when the compensation actions are themselves events on the log.

The fourth tradition is durable execution and workflow engines, represented in the current generation by Temporal, Cadence, AWS Step Functions, Camunda, and Netflix Conductor [Temporal 2023; AWS 2024; Camunda 2024]. This tradition provides the runtime primitives — durable timers, workflow histories, versioning, signal-and-query APIs — that make it feasible to run a workflow instance for weeks without losing state or duplicating work. Durable execution engines are the practical bridge between the theoretical tradition (event sourcing, sagas, BPM) and the production requirements (durability, scalability, observability) that this article's workflows impose.

Three gaps recur across these traditions. First, the traditions treat their concerns largely in isolation: BPM work rarely discusses event sourcing implementation, event sourcing work rarely discusses long-running workflow orchestration, and durable execution documentation rarely discusses the validation-workflow specifics that shape the design constraints. Second, the interaction between EDA and the mixed synchronous/asynchronous validation step patterns that real enterprise workflows contain is under-treated. Third, the observability requirements — particularly the state-of-every-in-flight-instance requirement that on-call engineers depend on — are usually treated as a general concern rather than as a workflow-specific concern with specific design implications. This article addresses these gaps with an integrated treatment scoped to the long-running validation workflow class.

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 8, Issue 4, July-August 2025||

DOI:10.15662/IJRPETM.2025.0804023

III. WORKFLOW PROPERTIES THAT CONSTRAIN THE ARCHITECTURE

Before presenting the architectural building blocks, we identify the properties of long-running enterprise validation workflows that make EDA the structurally appropriate pattern. These properties are not universal to all enterprise workflows; they characterize the class of workflows this article addresses.

Property 1 — Wall-clock duration far exceeds compute duration. A vendor onboarding may execute a few hundred milliseconds of aggregate compute across a fifteen-day duration. The workflow spends 99.9 percent of its wall-clock time waiting: for a callback, for a document, for a human, for an external system. Any architecture that holds resources (connections, threads, memory) proportional to wall-clock duration will exhaust those resources long before the workflow completes.

Property 2 — Progress is event-triggered, not time-triggered. The workflow advances when an event arrives (a validation result, a document upload, a human decision), not on a schedule. Time-based polling can approximate event-driven progress but produces one of two failure modes: too-frequent polling wastes resources and pressure-tests unrelated systems, and too-infrequent polling introduces latency between the availability of a result and its consumption by the workflow.

Property 3 — Validation steps are heterogeneous in latency and semantics. A single workflow may combine sub-second synchronous validations (a schema check, a range test), seconds-to-minutes asynchronous validations (an external API call with retries), minutes-to-hours human validations (a review queue), and days-to-weeks external validations (an agency response). The architecture must handle all four latency classes with a consistent state model.

Property 4 — State history is a regulatory artifact. For KYC, claims, financial filings, and similar workflows, the history of the workflow's state — which validations ran, when, with what inputs, with what outcomes — is required for audit and must be preserved for years. State that is stored implicitly (in an application's runtime memory, in an ephemeral cache) is unavailable to the auditor and produces the class of incidents that regulatory examiners treat most severely.

Property 5 — Steps are not naturally idempotent. Business validation steps are often stateful with side effects: a sanctions screen may create an audit record; a credit check may consume a paid API quota; a document review may deduct from a reviewer's queue. Retrying a business step as if it were an idempotent RPC produces duplicate audit records, duplicate paid API consumption, and duplicate assignments. The architecture must provide idempotency where the business step cannot.

Property 6 — Compensation is required, not optional. When a workflow fails partway or is canceled, the completed steps often have external side effects that must be actively reversed (a hold on a customer's account must be released; a provisional access grant must be revoked; a paid appraisal must be canceled or accounted for). The architecture must provide first-class compensation paths, not rely on the caller to reason about them ad hoc.

Property 7 — Workflow versions coexist in flight. A change to the workflow definition — a new validation step, a modified compensation path, a different SLA — must be deployable while thousands of workflow instances built against the previous version are still in flight. The architecture must therefore support versioned workflow definitions and route each instance to its instantiated version until completion.

These seven properties, taken together, are the design constraints that the five architectural building blocks in Section 4 respond to.

IV. FIVE ARCHITECTURAL BUILDING BLOCKS

We identify five architectural building blocks that together support the seven workflow properties above. The blocks are: the event backbone, the workflow orchestrator, the validation-step service pattern, the compensation and saga path, and the observability surface. Each block is described in terms of the workflow property it primarily serves, its internal design constraints, and its interfaces to the other blocks.

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 8, Issue 4, July-August 2025||

DOI:10.15662/IJRPETM.2025.0804023

4.1 Block 1 — The Event Backbone

Description. The event backbone is the durable, append-only, ordered log of events that record every state-relevant occurrence in the workflow. Every validation completion, every human decision, every external callback, every compensation invocation, and every workflow signal is recorded as an event on the backbone before it is acted upon.

Design constraints. The backbone must be durable — an event, once acknowledged, must survive the failure of any single component. The backbone must preserve ordering within a workflow instance, because the state of the workflow is a function of the event order (a "document received" event before a "review requested" event has different semantics than the reverse). The backbone should provide per-instance partitioning so that ordering is preserved instance-locally but the log scales globally. In practice this means Kafka with an instance-id partition key, Kinesis with a shard-per-instance mapping, or a durable execution engine's native history log (Temporal's history, Cadence's event log).

Interfaces. The backbone is written by the workflow orchestrator and by the validation-step services; it is read by the workflow orchestrator (to reconstruct workflow state), by the observability surface (to expose in-flight state), and by the audit-and-regulatory consumers (to satisfy state-history requirements).

Failure modes and mitigations. The classic failure mode is order violation — an event written to the backbone in a different order than the business logic assumed, usually because the writer did not partition correctly. The mitigation is a strict partition key based on workflow-instance identity. A second failure mode is retention exhaustion — the backbone runs out of storage because retention was configured for short-lived operational events but the workflow's history requires long retention. The mitigation is per-topic or per-log retention configured against the regulatory horizon (often years, not days).

4.2 Block 2 — The Workflow Orchestrator

Description. The workflow orchestrator is the component that defines the workflow's sequence of steps, dispatches work to validation-step services, waits for their completion, decides the next step based on outcome, and commits each step's decision to the event backbone. In durable-execution frameworks, the orchestrator is a workflow definition (Temporal workflow, Step Functions state machine, Camunda process definition) whose execution is replayed from the event log on any restart.

Design constraints. The orchestrator must be able to wait indefinitely for events without holding runtime resources — a workflow that has been suspended for six days waiting for an external agency response must occupy no memory or connection while it waits. The orchestrator must be deterministic in its replay behavior — replaying the event log must produce the same decisions, regardless of when the replay occurs — because non-deterministic orchestrator code produces workflow instances whose progression cannot be reconstructed. The orchestrator must support versioning so that an in-flight instance built against version N can complete against version N even after version N+1 has been deployed.

Interfaces. The orchestrator writes decisions and signals to the event backbone, dispatches work to validation-step services (via a task queue, an HTTP call, or a durable-execution activity), and exposes query and signal APIs to the observability surface and to external triggers.

Failure modes and mitigations. The classic failure mode is non-deterministic orchestrator code — a workflow that reads the wall-clock time, calls a random-number generator, or reads external state directly in the orchestrator layer — which produces workflow instances whose replays diverge from their originals. The mitigation is a strict discipline that reads and side effects belong in validation-step services (which are deterministic-replay-safe by being called through the durable-execution primitives), not in the orchestrator itself. A second failure mode is inadequate versioning — a workflow definition change breaks in-flight instances that were built against the previous version. The mitigation is explicit workflow versioning primitives (Temporal's GetVersion, Cadence's ChangeVersion, or equivalent in other engines).

4.3 Block 3 — The Validation-Step Service Pattern

Description. The validation-step service pattern is the shape that each validation step conforms to. Each step is an independent service — synchronous or asynchronous, human or machine, internal or external — that accepts an input event, performs the validation, and emits an outcome event on the backbone. The pattern standardizes how steps are invoked, how their outcomes are recorded, and how their failure modes are handled.

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 8, Issue 4, July-August 2025||

DOI:10.15662/IJRPETM.2025.0804023

Design constraints. Every step must accept an idempotency key — usually the workflow-instance identity plus a step-invocation identity — so that retries are safely absorbed. Every step must produce exactly one outcome event on the backbone per completed invocation; the backbone's writes must therefore be transactionally coupled to the step's side effects (either via an outbox pattern with a durable local queue, or via a durable-execution activity that atomically commits the side effect and the event). Every step must declare its expected latency class (sub-second, seconds-to-minutes, minutes-to-hours, days-to-weeks) so that the orchestrator can apply appropriate timeouts and escalation logic.

Interfaces. The step is invoked by the orchestrator via a task queue or activity dispatch; it writes its outcome to the event backbone; it is observed by the observability surface.

Failure modes and mitigations. The classic failure mode is a step that produces a side effect but fails to write its outcome event, leaving the workflow orchestrator waiting for a completion signal that will never arrive while the side effect has already occurred. The mitigation is the outbox pattern or the equivalent durable-execution primitive that couples the side effect and the event write into a single transaction (or into two operations with a compensating retry that idempotently resolves both). A second failure mode is a step whose latency class is mis-declared — a step declared as sub-second that is actually seconds-to-minutes will cause the orchestrator to treat legitimate results as timeouts. The mitigation is empirical latency classification with periodic review.

4.4 Block 4 — The Compensation and Saga Path

Description. The compensation and saga path is the sequence of compensating actions that must run when a workflow fails or is canceled after some validation steps have committed external side effects. Compensations are themselves validation-step services (or a specialized variant) that undo the side effect of a specific completed step, and their invocation is driven by the same event-driven machinery as the forward path.

Design constraints. Each forward step whose side effect requires reversal must declare its compensating step and the conditions under which the compensation applies. Compensations must be composable: a canceled workflow may require multiple compensations in reverse order of the forward steps, and the orchestrator must be able to sequence them reliably. Compensations must themselves be idempotent, because a compensation that fails partway and is retried must not double-reverse a side effect. Compensations must produce their own outcome events on the backbone, so that the workflow's history contains a complete audit of forward-and-compensation actions.

Interfaces. The compensation path is invoked by the orchestrator on cancellation or on unrecoverable failure; it writes compensation outcomes to the event backbone; it is observed by the observability surface with the same visibility as the forward path.

Failure modes and mitigations. The classic failure mode is a compensation that cannot reverse a completed side effect because the effect is externally irrevocable — a payment released to an external supplier, a document sent to a regulator. The mitigation is to identify these steps at design time and to require Approval-and-Gate (see companion article on HITL patterns) before their execution, so that irreversible side effects are never executed by the workflow autonomously. A second failure mode is compensation-order violation — compensations executed out of order, producing an intermediate state that is inconsistent with any legitimate workflow history. The mitigation is explicit reverse-order sequencing enforced by the orchestrator.

4.5 Block 5 — The Observability Surface

Description. The observability surface exposes the state of every in-flight workflow instance, the aggregate throughput and latency of the workflow definitions, the rate of compensations and unrecoverable failures, and the health of every validation-step service. It is the surface through which on-call engineers, business operators, and auditors interact with the running system.

Design constraints. The surface must expose per-instance state — where each in-flight instance is in its workflow, when it last progressed, and what it is waiting for — because an on-call engineer investigating a stuck instance needs to answer these questions in seconds, not minutes. The surface must expose aggregate metrics across a workflow definition — throughput, step-level latency distributions, error rates, compensation rates — because platform owners need to reason about the system as a whole. The surface must support query-by-attribute — find all instances waiting on validation X, or all instances for tenant Y, or all instances started in the last hour — because incident response and

International Journal of Research Publications in Engineering, Technology and Management (IJRPETM)

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 8, Issue 4, July-August 2025||

DOI:10.15662/IJRPETM.2025.0804023

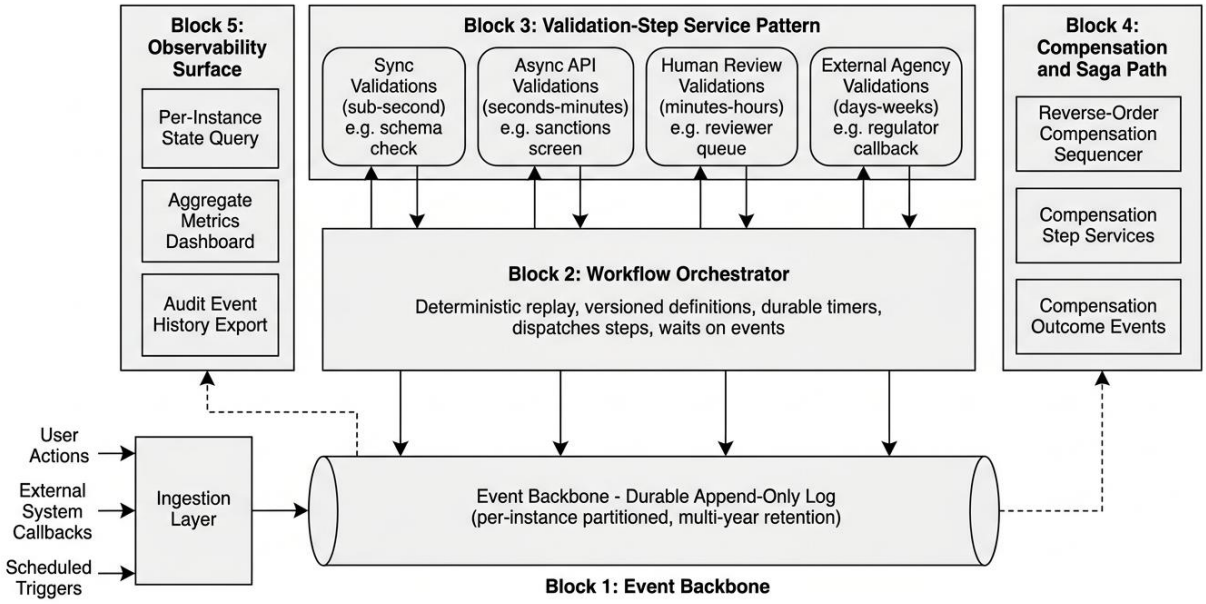
business inquiries both require it. The surface must derive its data from the event backbone, not from a separate database, so that it is always consistent with the workflow's authoritative state.

Interfaces. The surface reads from the event backbone (directly or via a projection into a queryable store), consumes orchestrator query APIs, and exposes dashboards, query interfaces, and alerting endpoints.

Failure modes and mitigations. The classic failure mode is a surface that is inconsistent with the workflow's actual state — a dashboard that shows a workflow as "in progress" when the workflow has actually completed, or an audit query that returns different results than the underlying event log. The mitigation is to derive the surface's data directly from the event backbone, either via read-time projection or via write-time materialization that is transactionally coupled to the backbone. A second failure mode is a surface that lacks per-instance query — the surface shows aggregate metrics but cannot answer "why is instance X stuck?" — which forces on-call engineers into ad hoc log grepping and multiplies incident duration. The mitigation is per-instance query as a required first-class capability of the surface, not an optional feature.

V. THE REFERENCE ARCHITECTURE

Figure 1. Reference Architecture for Event-Driven Long-Running Validation Workflows



Caption: End-to-end reference architecture for event-driven long-running enterprise validation workflows. The workflow orchestrator (center) drives per-instance state advancement; the event backbone (bottom) is the durable append-only log that every block reads from and writes to; validation-step services (top) implement individual validations across four latency classes; the compensation and saga path (right) handles reverse execution; the observability surface (left) provides per-instance and aggregate visibility. External event sources (user actions, external system callbacks, scheduled triggers) enter the system through defined ingestion points; downstream consumers (dashboards, audit exports, notification services) consume from the event backbone.

The reference architecture composes the five building blocks into a system that runs a long-running validation workflow end-to-end. External events enter the system through a well-defined ingestion layer: user actions (a submitted application), external system callbacks (an agency response), scheduled triggers (a nightly reconciliation), and manual signals (an operator canceling an instance). Each ingestion path writes the corresponding event to the event backbone and, where a new workflow instance must be created, triggers the workflow orchestrator to start a new instance.

Once started, a workflow instance progresses through its steps under the orchestrator's control. The orchestrator, reading the workflow definition and the event backbone for its instance, dispatches each validation step to the

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 8, Issue 4, July-August 2025||

DOI:10.15662/IJRPETM.2025.0804023

appropriate validation-step service. Sub-second synchronous validations complete inline; asynchronous validations return control to the orchestrator, which suspends the instance until an outcome event arrives. Human validations dispatch a task to a review queue (which is itself a validation-step service pattern) and suspend until the reviewer's decision produces an outcome event. External-system validations issue a request to the external system and suspend until a callback event arrives on the backbone via the ingestion layer.

The workflow instance's state at any moment is fully reconstructible from the event backbone — the sequence of events for that instance, replayed against the workflow definition, produces the exact current state. This property is what allows the orchestrator to be stateless in its runtime, to survive restarts, and to be redeployed without losing in-flight instances. The property is also what allows the observability surface to expose per-instance state without a separate state store: the surface projects the event log into whatever query shape is needed and can always rebuild the projection from the log.

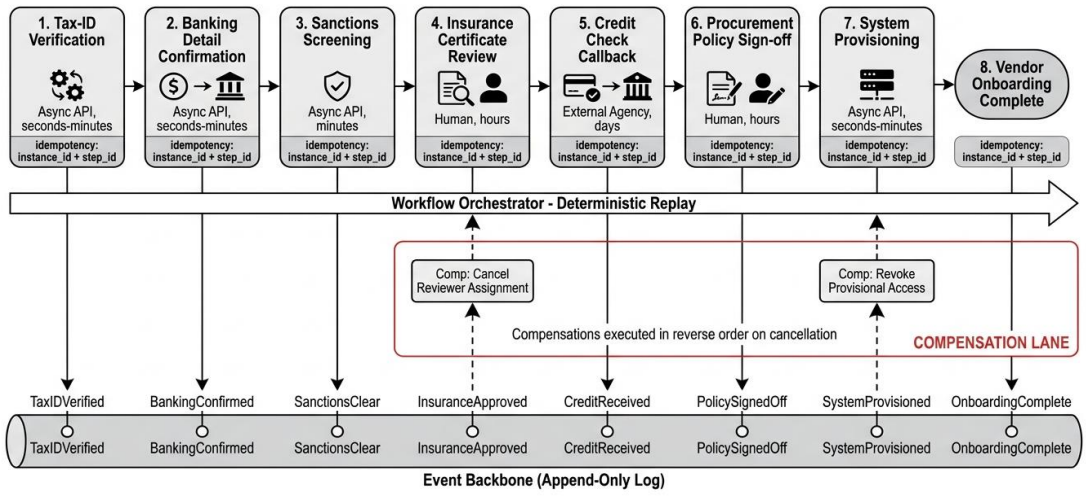
When a workflow instance fails unrecoverably or is canceled, the orchestrator invokes the compensation and saga path. The path reads the event log for the instance, identifies which forward steps have completed and have declared compensating actions, and dispatches those compensations in reverse order. Each compensation is itself a validation-step service, obeys the same patterns (idempotency, outcome event, latency class declaration), and produces its own event on the backbone. When all applicable compensations have completed, the workflow instance transitions to a canceled or failed state, and the observability surface reflects the transition.

The observability surface, running throughout, projects the event backbone into queryable indices (per-instance state, per-workflow-definition aggregates, per-tenant slices, per-latency-class step summaries). On-call engineers use the surface to answer "why is instance X stuck?" (the surface shows the last event received, the current wait condition, and the timeouts that apply); platform owners use it to answer "what is the throughput and error rate of workflow definition Y?" (the surface shows the aggregate metrics with time-series history); auditors use it to answer "what happened to instance Z between January and March?" (the surface shows the full event history with timestamps and identities).

The reference architecture is deliberately independent of the specific durable-execution technology chosen (Temporal, Cadence, Step Functions, Camunda, Netflix Conductor, or a proprietary equivalent). Each technology instantiates the blocks differently — Temporal, for example, unifies Blocks 1 and 2 into a single history-log-and-orchestrator abstraction — but the block-level responsibilities and interfaces remain the same. The choice of technology should be made on the basis of the technology's support for the block-level design constraints (durability, replay determinism, versioning, per-instance queryability, compensation primitives) rather than on the basis of superficial feature comparisons.

VI. WORKED SCENARIO — VENDOR ONBOARDING

Figure 2. Vendor Onboarding Workflow on Event-Driven Reference Architecture



|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 8, Issue 4, July-August 2025||

DOI:10.15662/IJRPETM.2025.0804023

Caption: A vendor-onboarding workflow instance progressing through eight validation steps across the four latency classes, with two compensation paths active. Each step is annotated with its latency class, its idempotency key derivation, and its declared compensation. The event backbone at the bottom shows the append-only log of events for one instance from ingestion to completion.

A mid-sized enterprise runs approximately fifty vendor-onboarding workflows per week, each spanning three to fifteen business days and involving between six and twelve validation steps depending on vendor category. The workflows had originally been implemented as a synchronous chain of REST calls behind a Spring Boot orchestrator, with state persisted to a relational database via a set of ad hoc columns representing the current step. Within eighteen months of launch, four failure modes were routine.

First failure mode — stuck instances. A vendor's insurance-certificate review would time out after the orchestrator's twenty-minute request timeout, but the review service had actually completed after twenty-five minutes; the review's completion was written to the review service's database but never propagated to the orchestrator, and the workflow instance remained in an "awaiting insurance review" state indefinitely. On-call engineers had to reconcile the two databases manually, and the reconciliation itself often produced duplicate follow-on steps because the reconciliation did not know which subsequent steps had already fired.

Second failure mode — duplicate external calls. When a validation-step service (sanctions screening, credit check) returned a transient error, the orchestrator retried the whole workflow step from the point of the error. Because the underlying external calls were not idempotent — the sanctions screening consumed a paid quota per invocation, and the credit check created an audit record per invocation — retries produced duplicate consumption and duplicate audit records. Finance and compliance both raised the issue independently within six months.

Third failure mode — compensation gaps. When a workflow was canceled midway (because the vendor withdrew or the buyer withdrew), the compensation of already-completed steps was handled by an ad hoc "cancel handler" that had been added incrementally and that did not cover every step. A canceled workflow might leave a provisional access grant active in the ERP system, a hold active on the vendor's tax record, or a pending review sitting in a reviewer's queue with no attached workflow. Each gap required manual cleanup by the operations team.

Fourth failure mode — auditor-unfriendly state history. Regulatory audits of the vendor-onboarding process required a state-by-state history of what validations ran for each vendor, when, and with what outcomes. The system's state was represented as current-step columns in the orchestrator database, not as a history, so the audit team had to reconstruct history from a mix of application logs, database change-data capture, and screenshots. The reconstruction was slow, error-prone, and consumed a substantial fraction of the audit team's quarterly capacity.

Redesigned state. The team migrated the vendor-onboarding workflow to a durable-execution EDA implementation, deploying all five architectural blocks.

Block 1 — the event backbone was implemented on the durable-execution engine's native history log, with per-workflow-instance partitioning and multi-year retention configured to meet the regulatory horizon. Every validation completion, human decision, external callback, and compensation invocation was written to the backbone before being acted upon.

Block 2 — the workflow orchestrator was implemented as a durable workflow definition, with the vendor-onboarding sequence encoded as a versioned workflow. In-flight instances continued to run against their instantiated version even after the definition was updated. The orchestrator was strictly deterministic in replay: no wall-clock reads, no random-number generators, and no direct external calls from the orchestrator layer.

Block 3 — the validation-step service pattern was applied to every step. Each step received an idempotency key derived from the workflow-instance identity and the step-invocation identity, so that a retried step invocation was safely absorbed without duplicate side effect. Each step's outcome was written to the backbone via the durable-execution activity primitive, which atomically committed the outcome event and the step's completion signal. Each step declared its latency class, and the orchestrator applied appropriate timeouts (twenty seconds for synchronous validations, twenty minutes for asynchronous API calls, four hours for human reviews with escalation, seven days for external agency responses).

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 8, Issue 4, July-August 2025||

DOI:10.15662/IJRPETM.2025.0804023

Block 4 — the compensation and saga path was implemented step-by-step. Each forward step whose side effect required reversal declared its compensating step: the provisional-access-grant step declared an access-revocation compensation; the tax-record-hold step declared a hold-release compensation; the reviewer-assignment step declared a queue-unassignment compensation. On cancellation, the orchestrator invoked the compensations in reverse order of completion, each with its own idempotency key and its own outcome event on the backbone.

Block 5 — the observability surface was built as a projection of the event backbone into a queryable index. On-call engineers could query "why is instance X stuck?" and see the last event received (a human-review dispatch), the current wait condition (awaiting reviewer decision), and the applicable timeout (four hours, with 47 minutes remaining). Platform owners could query "what is the compensation rate of the vendor-onboarding workflow this quarter?" and see the aggregate percentage of instances that had entered the compensation path. Auditors could query "what happened to vendor Z between March and April?" and see the complete event history with timestamps.

Results after twelve months in production. Stuck-instance incidents dropped from a weekly frequency of four to five to below one per month, and every remaining incident was reproducibly caused by an identifiable defect in a specific validation-step service rather than by an unreconcilable orchestrator-versus-service state divergence. Duplicate-external-call incidents dropped to zero measurable rate after the idempotency key discipline was enforced on all steps. Compensation gaps were eliminated as a class of incident because every forward step's compensation was declared and invoked by the orchestrator rather than by ad hoc cancellation handlers. Auditor-unfriendly state history was eliminated: the audit team's quarterly capacity consumption on vendor-onboarding audits dropped by approximately two-thirds, and the audit reports themselves became substantially cleaner because they were rendered directly from the event backbone.

The scenario illustrates three principles that hold beyond this single deployment. First, the failure modes of ad hoc long-running workflow implementations are not evenly distributed: they concentrate in the interactions between components (orchestrator versus step service, forward path versus compensation path, application state versus regulatory state), which is precisely where EDA's block-level discipline provides the strongest protection. Second, the operational cost of a full EDA implementation is not marginal — five blocks, each with its own design constraints and its own operational surface — but the incident cost of the ad hoc alternative is worse and grows non-linearly with workflow complexity. Third, the choice of durable-execution technology matters less than the discipline of the block-level design; teams that adopt Temporal without the discipline produce the same failure modes as teams that adopted Spring Boot orchestrators, and teams that adopt the discipline against a proprietary durable-execution engine can produce the same production quality as teams on the industry-leading platforms.

VII. TRADE-OFFS AND LIMITATIONS

The five-block EDA treatment is, in our experience, the right level of abstraction for long-running enterprise validation workflows, but the trade-offs deserve to be stated. The first trade-off is against synchronous orchestration. Synchronous orchestration is simpler to build initially, easier to reason about locally, and matches the mental model of most enterprise developers who came up on request/response systems. It works well for workflows whose wall-clock duration is comparable to compute duration and whose steps are naturally synchronous. For long-running validation workflows, synchronous orchestration produces the failure modes described in the worked scenario: stuck instances, duplicate side effects, compensation gaps, and audit-unfriendly state history. The trade-off is real, and enterprises that adopt EDA prematurely — for workflows that are actually short-running and synchronous — pay complexity cost for no benefit; enterprises that resist EDA for genuinely long-running workflows pay incident cost that grows without bound.

The second trade-off is against pure choreography. Pure choreography is the alternative that removes the workflow orchestrator entirely: each validation-step service reads from and writes to the event backbone, and the workflow emerges from the composition of the services without a central definition. Choreography is highly scalable and highly decoupled but sacrifices the workflow-level discipline that the orchestrator provides: the workflow definition is implicit in the emergent behavior of the services rather than explicit in a single artifact, versioning becomes distributed across services, compensations become the responsibility of each service rather than of a central definition, and auditors face the difficult task of reconstructing the workflow from the services' individual behaviors. The trade-off is context-specific. For workflows with a stable, well-understood shape and a small number of services, choreography can work;

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 8, Issue 4, July-August 2025||

DOI:10.15662/IJRPETM.2025.0804023

for workflows with regulatory audit requirements and complex compensation paths, orchestrated EDA (as presented in this article) is the more defensible choice.

The third trade-off is the operational cost of durable execution as a platform capability. The durable-execution engines (Temporal, Cadence, Step Functions, Camunda) are themselves systems that must be operated at production quality: they need capacity planning, patching, backup and recovery, versioning discipline, and observability. Enterprises that adopt EDA seriously must invest in this platform capability, either through vendor-hosted offerings (which reduce the operational burden but introduce vendor coupling and cost) or through internal platform teams (which retain flexibility but require substantial ongoing investment). Enterprises that under-invest in this platform layer produce a durable-execution deployment whose reliability is lower than the ad hoc orchestrator it replaced.

The fourth trade-off is the interaction with AI-agent validation steps. Modern validation workflows increasingly include steps whose completion semantics are probabilistic — an AI agent that classifies a document, an LLM that extracts a field, a machine-learning model that scores a risk. The durable-execution primitives handle these steps well at the mechanical level (they are activities like any other), but the semantic reasoning becomes more complex: the outcome event now includes a confidence score, the compensation path may need to differ for high-confidence versus low-confidence outcomes, and the observability surface must expose confidence distributions alongside outcome distributions. The trade-off is that the EDA architecture accommodates AI-agent steps naturally, but the workflow design work required to accommodate them is not trivial and cannot be treated as identical to designing for deterministic validation steps.

A principal limitation of this article is its emphasis on validation workflows as the archetype. Other long-running workflow classes — provisioning workflows, remediation workflows, migration workflows — share many of the properties described here and would benefit from the same block-level treatment, but the specific block-level tuning may differ. A second limitation is the absence of quantitative cost modeling for the block-level architecture; we have described the trade-offs qualitatively but not with numerical models that would let a specific enterprise calculate the crossover point between synchronous orchestration and EDA. A third limitation is the evolving relationship between durable execution and vendor cloud platforms; specific engines gain and lose features on a quarter-by-quarter basis, and readers should verify current platform capabilities against the block-level constraints rather than assume equivalence.

VIII. CONCLUSION

Long-running enterprise validation workflows have properties — wall-clock duration far exceeding compute duration, event-triggered progress, heterogeneous validation-step latencies, regulatory state-history requirements, non-idempotent business steps, mandatory compensation paths, in-flight version coexistence — that make Event-Driven Architecture the structurally appropriate pattern. Systems that adopt EDA deliberately, with block-level discipline, run these workflows with orders-of-magnitude fewer stuck-instance incidents, duplicate-side-effect incidents, compensation-gap incidents, and audit-friction incidents than systems that adopt request/response orchestration and try to force it to serve long-running workflows.

We have argued that the right level of abstraction is the block, not the platform, and we have identified five architectural building blocks that a production EDA implementation must include. The event backbone is the durable, ordered, append-only log that every other block reads from and writes to. The workflow orchestrator is the deterministic, versionable definition of the workflow's step sequence and decision logic. The validation-step service pattern is the standardized shape of every step, with idempotency, outcome events, and latency-class declarations. The compensation and saga path is the reverse-order, idempotent, event-driven reversal of forward steps' side effects. The observability surface is the projection of the event backbone that exposes per-instance and aggregate state to on-call engineers, platform owners, and auditors.

The reference architecture we have presented composes the five blocks into an end-to-end system that is independent of the specific durable-execution technology chosen. The worked scenario from a vendor-onboarding workflow illustrated the block-level composition and the operational realities of running fifty workflows per week across three-to-fifteen-day durations, with substantial improvements in incident rate, compensation completeness, and audit efficiency.

The practitioners we expect to use this material face three pressures — to build workflows that survive their long wall-clock durations, to defend the workflow's state history to regulators and auditors, and to maintain operational reliability

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 8, Issue 4, July-August 2025||

DOI:10.15662/IJRPETM.2025.0804023

at a scale where ad hoc mechanisms fail — and these pressures pull in different directions. The five-block EDA treatment offers a path that does not sacrifice any of the three. The choice of durable-execution technology matters less than the discipline of block-level design, and the choice between EDA and synchronous orchestration should be made on the basis of the workflow's structural properties rather than on the basis of team familiarity. The unifying recommendation is to treat long-running validation workflows as event-driven by construction, to invest in the five blocks with matching depth, and to operate them with the observability discipline that on-call engineers and auditors both require, because in the current regime of enterprise validation workflows the failure modes of ad hoc orchestration are quieter and more corrosive than the failure modes of the individual validation steps themselves, and they demand attention before the next incident or audit forces it.

REFERENCES

1. van der Aalst, W. M. P. (2013). Business Process Management: A Comprehensive Survey. *ISRN Software Engineering*, Article 507984.
2. OMG. (2011). *Business Process Model and Notation (BPMN) Version 2.0*. Object Management Group.
3. Fowler, M. (2005). Event Sourcing. *martinfowler.com* (widely cited practitioner reference).
4. Young, G. (2010). CQRS and Event Sourcing. *Command Query Responsibility Segregation Documents*.
5. Garcia-Molina, H., & Salem, K. (1987). Sagas. *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, 249–259.
6. Richardson, C. (2018). *Microservices Patterns*. Manning Publications.
7. Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media.
8. Hohpe, G., & Woolf, B. (2003). *Enterprise Integration Patterns*. Addison-Wesley.
9. Newman, S. (2021). *Building Microservices* (2nd ed.). O'Reilly Media.
10. Temporal Technologies. (2023). *Temporal Documentation: Workflows, Activities, and Signals*. Temporal Technologies Inc.
11. AWS. (2024). *AWS Step Functions Developer Guide*. Amazon Web Services.
12. Camunda. (2024). *Camunda Platform 8 Documentation*. Camunda Services GmbH.
13. Netflix Technology Blog. (2024). *Netflix Conductor: A Microservices Orchestrator*. Netflix Inc.
14. Vogels, W. (2009). Eventually Consistent. *ACM Queue*, 7(6).
15. Bernstein, P. A., & Newcomer, E. (2009). *Principles of Transaction Processing* (2nd ed.). Morgan Kaufmann.