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 7, Issue 1, January-February 2024||

DOI:10.15662/IJRPETM.2024.0701013

Monolith-to-Microservices Migration in Enterprise Operations Platforms: A Workflow-Oriented Architecture

Makarand Gujarathi

Independent Researcher, USA

ABSTRACT: Monolith-to-microservices migration is widely practiced in enterprise engineering but rarely well characterized in the literature. Engineering teams often treat the migration as a refactoring exercise, when in reality it is a workflow re-architecting exercise, and the difference determines whether the migration improves or harms the operations platform. This article develops a workflow-oriented treatment of monolith-to-microservices migration in enterprise operations platforms. We position the migration as a workflow decomposition problem rather than a code decomposition problem; we identify the five workflow archetypes that dominate enterprise operations platforms (long-running approvals, batch orchestration, event-driven state transitions, human-in-the-loop review, and time-sensitive material flows), and show how each archetype maps to a distinct decomposition pattern: strangler-fig with bounded context for long-running approvals, saga orchestration for business-process flows, event-carried state transfer for cross-domain propagation, dual-write bridging for human-in-the-loop review, and shadow-write reconciliation for time-sensitive material flows. We then examine the migration mechanics that determine whether the re-architecting succeeds in production: workflow-aware dependency analysis, contract-first API design, idempotent compensation, gradual traffic shifting with feature-flagged routing, and observability that surfaces in-flight workflow state. We close with a phased migration framework that connects migration decisions to workflow properties rather than to service boundaries drawn prematurely. The result is a practitioner's map for migrating enterprise operations platforms in a way that preserves correctness during the migration while improving the architecture that the platform leaves behind.

KEYWORDS: monolith-to-microservices migration, workflow decomposition, strangler fig pattern, saga orchestration, event-carried state transfer, contract-first API, idempotent compensation, enterprise operations platforms, bounded context

I. INTRODUCTION

Enterprise operations platforms are the workflow-bearing spine of an enterprise. They coordinate purchase orders and approvals, route customer-service tickets, dispatch field technicians, run payroll batches, syndicate inventory data between warehouses, and adjudicate regulatory submissions. They are not optional infrastructure. They are the substrate on which a business can actually execute.

These platforms have traditionally been built as monoliths, often for good reasons at the time: a single deployable unit, a single set of invariants, a single place to write the business rules that govern the business, and a single team on call when something goes wrong. Over time, the monolith accumulates. New workflow types are added without retiring old ones. New regulatory constraints introduce data fields that travel through every code path. Teams, having grown from three engineers to thirty, fragment the code ownership without fragmenting the deployment unit, and the release pipeline slows from minutes to weeks. The platform starts to govern the enterprise rather than serve it.

The conventional wisdom for this situation today is to migrate to microservices. A large body of practice, several textbooks, and many conference talks describe how to do this. The literature emphasizes bounded contexts, service boundaries, API contracts, deployment independence, and team topology. It is broadly correct at the level of intent. It is broadly incomplete at the level of execution. Migrating an enterprise operations platform to microservices by following the conventional wisdom often produces a worse platform: a distributed monolith with worse observability, worse deploy cadence, and worse correctness guarantees than the original monolith it replaced.

This article argues that the dominant reason for these failures is that microservices migrations treat the problem as a code decomposition problem when it is in fact a workflow decomposition problem. Code decomposition answers the question "which classes live in which service." Workflow decomposition answers an earlier question: "what are the

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

||Volume 7, Issue 1, January-February 2024||

DOI:10.15662/IJRPETM.2024.0701013

actual workflows that traverse this platform, and where are their natural seams." Once the workflow seams are correctly identified, code decomposition becomes routine. When workflow seams are incorrectly identified — or worse, skipped entirely — code decomposition produces artifacts that are nominally independent but operationally entangled.

The contribution of this article is to develop a workflow-oriented treatment of monolith-to-microservices migration in enterprise operations platforms. The treatment has four parts. First, we characterize the migration as a workflow decomposition exercise and identify the principles that distinguish a workflow-correct migration from a code-correct but workflow-wrong one. Second, we identify five workflow archetypes that dominate enterprise operations platforms and show how each maps to a distinct decomposition pattern. Third, we examine the migration mechanics — dependency analysis, contract design, compensation, traffic shifting, observability — that determine whether the re-architecture succeeds in production. Fourth, we present a phased migration framework that connects migration decisions to workflow properties rather than to service boundaries drawn prematurely.

The remainder of the article is organized as follows. Section 2 characterizes the problem space and motivates the workflow-oriented view. Section 3 introduces the five workflow archetypes. Section 4 maps each archetype to a decomposition pattern. Section 5 examines migration mechanics. Section 6 presents the phased migration framework. Section 7 instantiates the framework in a worked scenario. Section 8 discusses trade-offs and limitations. Section 9 concludes.

III. BACKGROUND AND PROBLEM CHARACTERIZATION

The phrase monolith-to-microservices migration entered mainstream engineering vocabulary through the early-2010s wave of post-monolith retrospectives at large internet companies. The pattern has since propagated into enterprise engineering, where the workloads are different but the impulse is the same: a system that has compounded over years needs to be opened up so that change can resume at a reasonable pace.

We define the problem precisely. The starting point is an enterprise operations platform deployed as a single unit, typically a single application server (or a small number of horizontally scaled instances of the same unit) under the ownership of a single team or a small set of closely coupled teams. The platform implements a set of business workflows end to end: a purchase order moves from creation through approval, dispatch, receipt, and accounting close; a customer-service ticket moves from intake through triage, assignment, resolution, and customer confirmation. These workflows have invariants — no PO can be approved by the same person who raised it; no ticket can be closed without an entry in the resolution log — and the invariants are enforced inside the monolith's single transactional boundary.
The destination is a set of independently deployable services, each owning a well-defined slice of the platform's data and behavior, communicating through well-defined APIs, ideally aligned to bounded contexts. Migration is the process of moving from one end of this transition to the other without losing the invariants that the business needs the platform to enforce, and without disrupting the workflows that depend on it during the move. The second clause distinguishes a migration from a green-field rewrite.

Three motivations drive the migration. First, deployment cadence. The monolith's single release pipeline constrains all teams to ship at the slowest team's pace, which produces queuing, code freezes, and a culture of risk-averse change. Second, team topology. As the team grows beyond what one person can hold in working memory, the cost of cross-team coordination within a single codebase grows faster than linearly, until the marginal change becomes more expensive than the marginal value it produces. Third, failure domain isolation. A bug in one workflow type in the monolith can take down all workflow types; service decomposition narrows the blast radius when an individual workflow misbehaves.

Three failure modes recur. First, the distributed monolith. The team decomposes code into services but does not decompose the database; transactions that previously fit inside one database now span several, and the application layer must coordinate them without a robust pattern. The platform becomes a distributed monolith: more moving parts, fewer transactional guarantees, and a failure mode that is harder to diagnose than the one it replaced. Second, the strangler that strangles itself. The strangler-fig pattern is widely recommended in the literature, and rightly so, but a naive strangler routes traffic between the monolith and the new service through a feature flag without coordinating the data flows. The monolith writes to its database; the service writes to its database; neither catches the divergence; the platform now has two sources of truth for the same workflow. Third, the premature decomposition. The team decomposes along physical seams in the code (package boundaries, class hierarchies) rather than along workflow

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

||Volume 7, Issue 1, January-February 2024||

DOI:10.15662/IJRPETM.2024.0701013

seams; the resulting services couple tightly at the workflow level and end up changing together in every release that matters, providing no deployment independence in practice.

We argue that these failure modes share a common root cause: the team performed a code decomposition without first performing a workflow decomposition. Code decomposition asks where a class should live. Workflow decomposition asks where a business process naturally has a seam. The two questions are not the same. A workflow seam in a purchase-order workflow is between approval and dispatch, not between the approval class and the approval repository. Aligning service boundaries to workflow seams rather than to class boundaries is what gives microservices the properties the literature promises.

2.1 Related Work and Background Literature

The strangler-fig pattern, popularized by Martin Fowler, describes how to incrementally migrate an existing system by growing a replacement around it [1]. Our treatment extends the pattern by requiring workflow analysis as a precondition: a strangler-grown service that replaces code without understanding the workflow it sits inside reproduces the monolith's complications at a different scale.

Saga patterns, including both choreography and orchestration variants, address multi-step business transactions that span multiple services [2,3]. We treat sagas as one of five decomposition patterns rather than as the default, because not every workflow in an enterprise operations platform is a saga — long-running human approvals, batch orchestration, and event-driven state transitions each have different natural patterns.

Domain-driven design and bounded contexts provide the conceptual vocabulary for identifying workflow seams [4]. Our treatment connects bounded contexts to a small set of empirically recurring workflow archetypes, so that practitioners do not have to derive their context boundaries from first principles every time.

Workflow orchestration systems, including Camunda, Temporal, and AWS Step Functions, provide infrastructure for executing the patterns described in this article [5]. The patterns are not vendor-specific; workflow orchestration engines make the patterns easier to implement.

The phased migration framework presented here is consistent with the strangler-fig pattern and observability-first migration guidance, while adding a workflow-archetype-driven ordering that we have not seen formalized in the same way.

III. THE FIVE WORKFLOW ARCHETYPES

Enterprise operations platforms are not uniform. Different workflow types exhibit different temporal properties, different consistency requirements, different human-in-the-loop characteristics, and different failure modes. A migration that treats them as uniform produces a uniform set of decomposition decisions, which is wrong for every individual workflow. We identify five archetypes that consistently appear across enterprise operations platforms and that, when correctly identified, dictate distinct decomposition patterns.

3.1 Long-Running Approvals

Long-running approvals are workflows that wait for human decisions and may wait days or weeks. A purchase order that moves from requester to budget owner to legal review to final approver is a long-running approval. These workflows are characterized by sparse machine-readable activity, abundant human-readable activity, persistent state across the wait, and hard correctness requirements (no double-approval, no approval by the requester, no approval past a budget cap).

The decomposition seam in a long-running approval is between each approval step and the next. Each step is small, has clear inputs and outputs, and can be modeled as a state transition on a workflow instance. The natural decomposition pattern is bounded-context microservices, one or two per approval kind, sharing a workflow engine that maintains the in-flight state.

3.2 Batch Orchestration

Batch orchestration is workflow that runs at scheduled times against large data sets. End-of-day journal posting, monthly statement generation, regulatory submission archives are batch orchestration. These workflows are

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

||Volume 7, Issue 1, January-February 2024||

DOI:10.15662/IJRPETM.2024.0701013

characterized by large volumes, predictable timing, deterministic outcomes, and conventional failure modes (partial completion, mid-batch crash).

The decomposition seam in a batch orchestration workflow is between the orchestration step (when does the batch run, in what order, with what retry policy) and the per-entity processing step (how do we handle one customer, one PO, one transaction). The natural decomposition pattern is a single orchestrator that runs a workflow engine and dispatches to per-entity worker services.

3.3 Event-Driven State Transitions

Event-driven state transitions are workflows that propagate state changes across organizational boundaries without a coordinating orchestrator. Inventory updates, pricing changes, customer-profile changes, and risk-flag propagation are the canonical examples. These workflows are characterized by high frequency, low per-event value, broad consumers, and tolerance for eventual consistency.

The decomposition seam in an event-driven workflow is between the producer of a state change and the consumers that react to it. The natural decomposition pattern is event-carried state transfer: the producer publishes the state change, and consumers materialize their own local view of the relevant subset.

3.4 Human-in-the-Loop Review

Human-in-the-loop review is a workflow that combines machine processing with structured human judgment at specific points. A claims adjudication, a loan underwriting decision, a regulatory compliance check that an officer must sign off on, is a human-in-the-loop review. These workflows are characterized by moderate durations (hours to days), regulated decision points, and audit requirements that demand durable evidence of the human decision.

The decomposition seam here is between the automation that prepares the decision context and the human review that produces the decision. The natural decomposition pattern is dual-write bridging, in which both the legacy monolith and the new review service write to a shared decision log while the workflow spans both codebases.

3.5 Time-Sensitive Material Flows

Time-sensitive material flows are workflows that move physical or quasi-physical state on a deadline. A package dispatch, a perishable-goods shipment, a same-day-settlement trade execution are time-sensitive material flows. These workflows are characterized by monetary or physical loss when the deadline is missed, by environmental variation that the workflow must adapt to, and by limited tolerance for asynchronous coordination.

The decomposition seam in a time-sensitive material flow is between the front-end workflow that consumes user-facing API calls and the back-end workflow that tracks physical state. The natural decomposition pattern is shadow-write reconciliation, in which the new service writes its view of the state in parallel with the monolith for an extended overlap period, after which reconciliation replaces the monolith's view.

3.6 Archetype Overlap

Real workflows combine archetypes. A purchase order combines long-running approval (the approval steps) with event-driven state transitions (when the approver acts, downstream consumers react) and time-sensitive material flows (the dispatch must happen on a carrier's schedule). The decomposition patterns for a workflow that combines archetypes must be combined as well: bounded-context services for the approval portion, event-carried state transfer for the propagation, and shadow-write reconciliation for the dispatch.

The archetype identification is therefore a guide rather than a categorization. A workflow that is mostly event-driven with a small human-in-the-loop component is decomposed primarily for event-driven flow with a dual-write bridge around the human component. The decomposition pattern chosen for the dominant archetype is the spine; secondary patterns attach to it.

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 7, Issue 1, January-February 2024||

DOI:10.15662/IJRPETM.2024.0701013

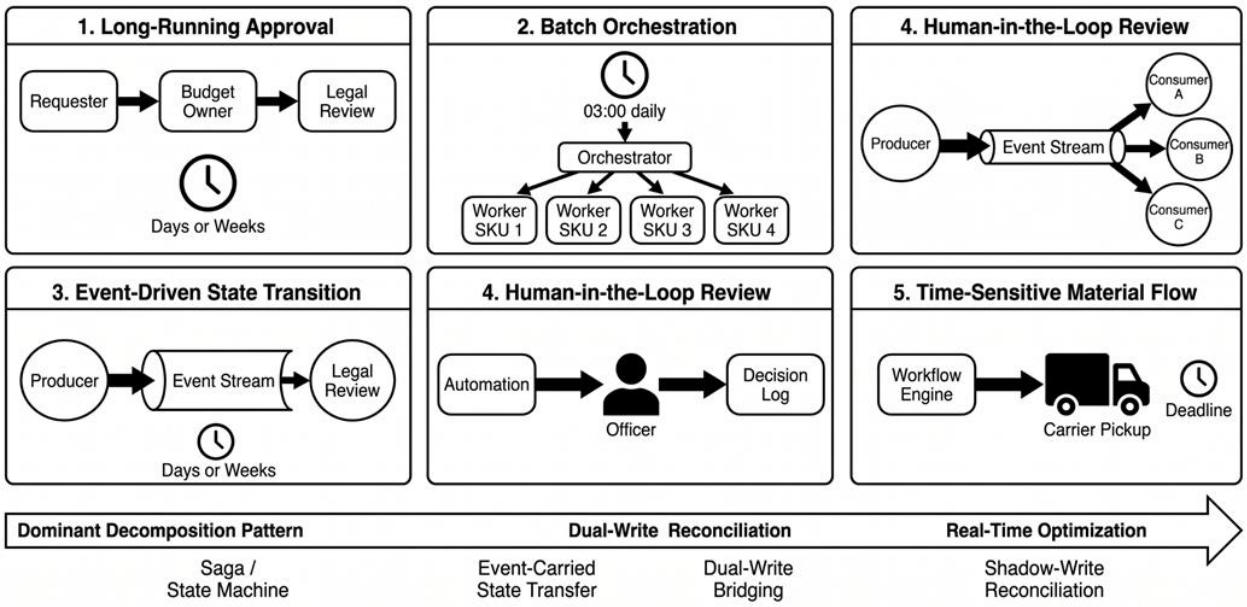

Figure.1. The Five Workflow Archetypes in Enterprise Operations Platforms

IV. DECOMPOSITION PATTERNS BY ARCHETYPE

We now treat each archetype and its decomposition pattern in detail. Each pattern is described in terms of its invariants, its data-flow shape, its compensation behavior, and its observability requirements.

4.1 Long-Running Approvals — Bounded-Context Microservices with Workflow Engine

The decomposition pattern for long-running approvals is one bounded context per approval kind, with a shared workflow engine that maintains in-flight state. The bounded context owns its data (a requester table and an approver table for a budget-approval context, for example) and its approval logic. The workflow engine persists the state machine for each in-flight approval and routes the approval to the next approver according to the rules of the context.

The invariant preserved by this pattern is approval correctness: no double approval, no approval past a cap, no approval by the requester. These invariants were previously enforced by a single transactional boundary in the monolith; the pattern preserves them by treating approval as a state machine executed by an external workflow engine rather than by code that runs in a single transaction.

The data flow is message-driven. Each approval step is implemented as a workflow-engine task; the engine persists the workflow instance state after each task; the next task is dispatched when the next approver acts. The pattern tolerates long pauses (days or weeks) at any step because the workflow state is durable and the engine resumes from persisted state on resumption.

Compensation in this pattern is rare. Long-running approvals do not usually need to roll back; an approval that turns out to be wrong simply enters a separate cancellation workflow. The pattern treats compensation as a separate workflow rather than as a transactional rollback.

Observability must include in-flight workflow state for every active approval, which is a workflow-engine-native concern. A traditional application monitoring stack that tracks only HTTP latency and error rates will miss most failures in this pattern.

4.2 Batch Orchestration — Orchestrator with Per-Entity Workers

The decomposition pattern for batch orchestration is a single orchestrator that runs a workflow engine and dispatches per-entity processing to a set of worker services. The orchestrator reads its work list, persists each task to a per-entity worker through a durable queue, persists the result, and moves to the next task. The worker services own the per-entity processing logic.

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

||Volume 7, Issue 1, January-February 2024||

DOI:10.15662/IJRPETM.2024.0701013

The invariant preserved by this pattern is exactly-once processing per entity, even across worker restarts and orchestrator restarts. In the monolith, this was typically enforced by a single database transaction per entity. The pattern preserves it by treating each entity's processing as a saga step with explicit idempotency keys and by relying on the orchestrator's persistent work-list for resumability.

The data flow is queue-mediated. The orchestrator writes per-entity tasks to a durable queue; the workers consume; results flow back through a result topic. The pattern tolerates workers and orchestrators failing mid-batch because the queue and the work list preserve state.

Compensation in this pattern is per-entity. If an entity's processing fails after several retries, the orchestrator moves it to a dead-letter flow rather than rolling back the batch. Compensation for the overall batch is not a useful concept; partial success is the norm.

Observability must include per-entity processing state, queue depth, and per-worker throughput. The orchestrator's workflow-engine logs are a primary source of truth for batch progress.

4.3 Event-Driven State Transitions — Event-Carried State Transfer

The decomposition pattern for event-driven state transitions is event-carried state transfer. The producer of a state change publishes a message describing the new state; consumers materialize their own local view of the relevant subset. The producer does not know or care which consumers exist; the consumers do not couple to the producer's data model beyond the event schema.

The invariant preserved by this pattern is convergence: independent consumers eventually agree on state with the producer, given sufficient time and absence of new events. In the monolith, an event-driven transition was typically implemented by a transaction that updated both the producing table and a propagation queue; the pattern preserves eventual consistency by depending on idempotent consumers that can re-process events.

The data flow is log-shaped rather than request-shaped. Each event flows through a log (or a stream, or a queue with retention); consumers subscribe and materialize their view. Events are ordered within a partition; cross-partition ordering is not guaranteed. The pattern tolerates producers and consumers coming and going because the log retains events.

Compensation is rare. Event-carried state is a write-only operation; consumers either succeed or retry. When a consumer falls permanently behind, the consumer's owner must re-bootstrap their state from the log rather than relying on the producer.

Observability must include log lag per consumer, consumer-side idempotency violations, and divergence between consumer materialized state and producer source state.

4.4 Human-in-the-Loop Review — Dual-Write Bridging

The decomposition pattern for human-in-the-loop review is dual-write bridging: the legacy monolith and the new review service both write to a shared decision log during an overlap period. The decision log is the durable record of the human decision; the workflow engine reads it to know what state the workflow is in. Once the overlap period completes, the service becomes the sole writer.

The invariant preserved by this pattern is audit traceability of human decisions. In the monolith, this was typically enforced by a single audit-log table updated inside the same transaction as the decision. The pattern preserves it by both the monolith and the new service writing to the same decision log, so the audit trail spans both.

The data flow is dual-writer. The monolith writes to the decision log when it performs the decision; the new service writes to the same log when it performs the decision; the workflow engine reads from the log. Synchronization between the two writers is via a coordination protocol that prevents both from deciding the same workflow at the same time.

Compensation in this pattern depends on the decision type. Decisions that can be overridden (a loan officer's change of mind) enter a separate correction workflow; decisions that cannot be overridden are write-once. The decision log records attempts and overwrites; the workflow engine distinguishes them.

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

||Volume 7, Issue 1, January-February 2024||

DOI:10.15662/IJRPETM.2024.0701013

Observability must include agreement-rate metrics for the two writers, decision-log write-rate, and conflict-resolution outcomes when both writers attempt the same workflow.

4.5 Time-Sensitive Material Flows — Shadow-Write Reconciliation

The decomposition pattern for time-sensitive material flows is shadow-write reconciliation. The new service writes its view of the state in parallel with the monolith for an extended overlap period; both writers produce their view of state; reconciliation compares them. After the overlap period, when reconciliation shows consistent agreement above a threshold for long enough, the new service becomes the sole writer and the monolith retires from this workflow.

The invariant preserved by this pattern is no loss of physical state across the migration. In the monolith, the physical state was tracked in a single table updated transactionally; the pattern preserves it by treating the new service's view as authoritative only after shadow-write agreement has been demonstrated.

The data flow is parallel-writer with reconciliation. Both writers persist their view; a reconciler compares at scheduled intervals and reports divergences. Once divergence drops below a configured threshold for a configured duration, migration proceeds.

Compensation in this pattern is rare. Time-sensitive flows do not usually tolerate compensation; if divergence is detected after the cutover, the workflow rolls forward rather than back.

Observability must include divergence metrics, reconciler lag, and physical-state accuracy against an external ground truth (a carrier API, an inventory report, a trade settlement feed).

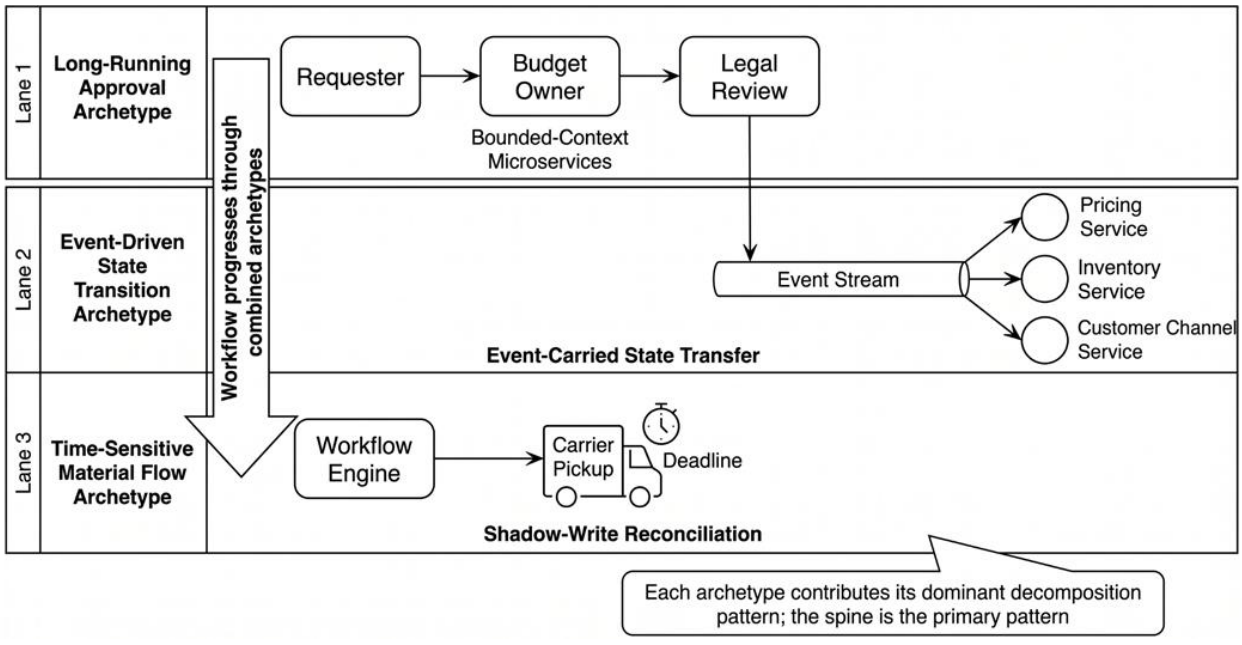

Figure.2. Decomposition Patterns Composing Along a Purchase-Order Workflow

V. MIGRATION MECHANICS

The decomposition patterns describe what the post-migration architecture looks like. The migration mechanics describe how to get from the monolith to the architecture without losing the invariants that the business depends on. These mechanics are largely orthogonal to the choice of decomposition pattern but determine whether the migration actually succeeds.

5.1 Workflow-Aware Dependency Analysis

A migration that does not identify the workflows before it identifies the services is a migration that decomposes code along physical seams. Workflow-aware dependency analysis begins with a workflow inventory: every distinct business

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

||Volume 7, Issue 1, January-February 2024||

DOI:10.15662/IJRPETM.2024.0701013

workflow the platform supports, its current state-machine shape in the monolith, its data, its human-decision points, and its invariants. From the workflow inventory, the team derives the bounded contexts that the decomposition patterns require.

Dependency analysis that begins from code structure tends to produce services that span workflow boundaries. A migration that begins from workflow analysis tends to produce services that align to those boundaries. The latter is what makes microservices deployments independent in practice.

5.2 Contract-First API Design

Once a service's responsibility is defined, its API contract must be the next artifact, not the last. Contract-first design lets the new service and the monolith agree on the interface before either code path has been written, which in turn lets the migration proceed without a freeze. A migration that designs APIs as the last step before integration tends to discover that the API cannot serve the workflows it must support, requiring late changes that are expensive once migration has started.

The API contract must be explicit about its backward-compatibility guarantees during the migration. A contract that promises stability across versions lets the monolith and the service evolve separately; a contract that does not forces the migration to be coupled on both sides.

5.3 Idempotent Compensation

Saga patterns compensate for failed steps by issuing compensating actions. The compensating action must be idempotent: a retry of the compensation must produce the same observable effect as a single application. Idempotency in compensation is what makes a partially-failed migration recoverable; without it, every retry becomes a new failure mode.

In enterprise operations platforms, idempotency keys are typically derived from a stable workflow-instance identifier and the compensation step index. The compensating service maintains an idempotency table keyed on this composite; retries that hit the table are no-ops. The table is the operational contract between the migration and the business.

5.4 Gradual Traffic Shifting with Feature-Flagged Routing

The strangler-fig pattern routes traffic between monolith and new service through a feature flag. The flag must be evaluable per request, per workflow instance, and per annotated cohort so that the migration can be graduated rather than flipped. A flag that can only be toggled globally produces a migration that is all-or-nothing, which is incompatible with enterprise operations platforms because the workflows they support cannot tolerate coordinated downtime.

Traffic shifting must include data-plane shifting, not only request shifting. When the new service begins to serve requests, the writes that those requests produce must flow to the new service's data store, not to a shared store that both writers are writing to. Shared stores during migration period create the distributed-monolith failure mode described in Section 2.

5.5 Observability of In-Flight Workflow State

Standard observability tooling tracks HTTP latency, error rates, and resource utilization. This is necessary and insufficient for migration observability. The team must additionally track: workflow-instance counts in each state of each workflow's state machine, queue depths at the migration boundaries, divergence between monolithic state-of-truth and new-service state-of-truth, and compensation activity per workflow type.

A migration that lacks in-flight workflow observability cannot distinguish "the new service is handling 5% of traffic successfully" from "the new service is handling 5% of traffic and silently diverging from the monolith on 1% of those." The latter is the failure mode that turns a migrating platform into a degraded platform.

5.6 Schema Evolution Compatibility

Schema changes in the data store supporting an in-flight migration must be forward-and-backward-compatible across the migration period. The monolith running the old schema must read the new schema gracefully; the new service running the new schema must read records written under the old schema gracefully. Standard compatibility patterns from event-stream design apply here, and the migration's compatibility guarantees are part of its operational contract.

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

||Volume 7, Issue 1, January-February 2024||

DOI:10.15662/IJRPETM.2024.0701013

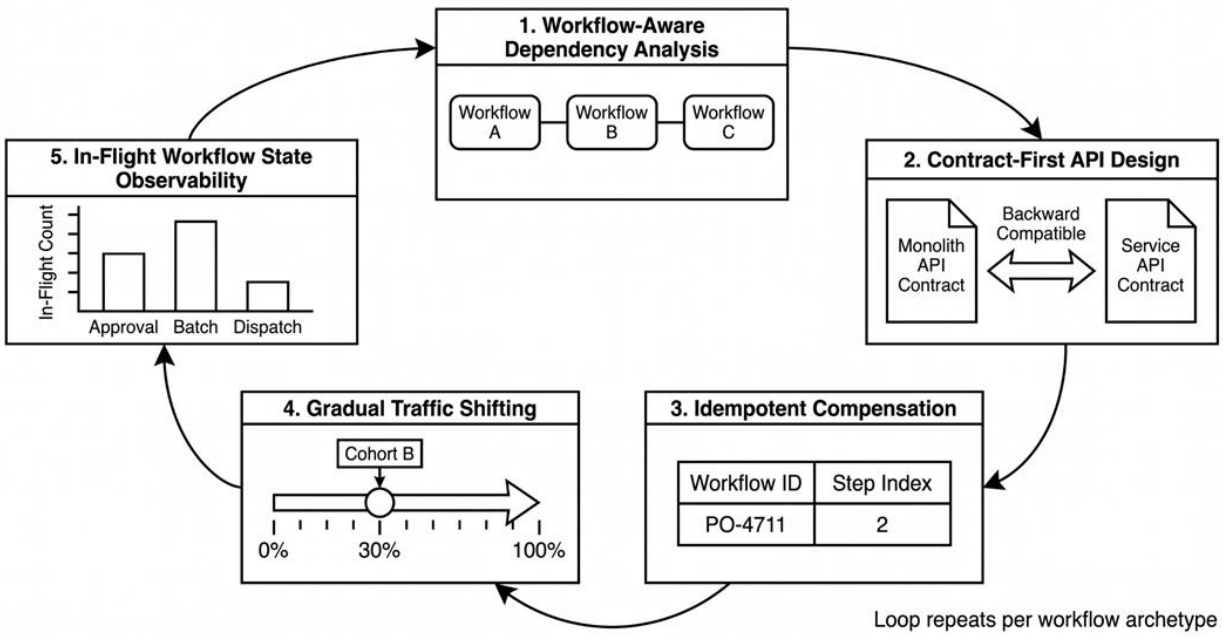

Figure.3. The Five Migration Mechanics as a Clockwise Loop

VI. PHASED MIGRATION FRAMEWORK

The decomposition patterns in Section 4 and the migration mechanics in Section 5 compose into a phased migration framework. The framework orders the migration of an enterprise operations platform according to workflow archetype rather than along code boundaries.

6.1 Phase Zero — Workflow Inventory and Archetype Classification

Before any service is split off, the team produces a workflow inventory. Each workflow is classified by archetype, and any workflow that combines archetypes is annotated with its dominant archetype and its secondary patterns. The output of Phase Zero is a workflow-archetype map of the platform, with each workflow labeled by its dominant pattern from Section 4.

This is a long phase and is often skipped, which is the proximate cause of most failed migrations. A workflow inventory on a non-trivial platform typically takes weeks, not days, and produces disagreements among team members about what counts as a workflow and which archetype dominates. The disagreements are themselves informative; they surface invariants that the team had not previously made explicit.

6.2 Phase One — Boundary Extraction and Contract-First API

For each workflow identified in Phase Zero, the team extracts its natural boundary: the seam where its dominant decomposition pattern dictates a service split. The team writes the API contract for each new service before writing its code. Contract-first design lets the monolith and the service evolve independently during Phase Two.

Phase One produces a contract document, an ownership map, and a backward-compatibility commitment for each new service. It does not produce running code. The discipline of writing the contract before the code is what catches the disagreements that would otherwise surface mid-migration.

6.3 Phase Two — Shadow-Read and Idempotent Compensation

For each new service, the team implements shadow-read: the service reads the same data the monolith reads, computes its result, and logs the result without acting on it. The team additionally implements idempotent compensation for the service's sagas, with the idempotency-table design committed before any compensation logic is written.

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

||Volume 7, Issue 1, January-February 2024||

DOI:10.15662/IJRPETM.2024.0701013

Shadow-read exposes divergence between the service's view and the monolith's view without acting on the divergence. It is the migration team's debugging tool, not the customer's. The compensation discipline keeps failures recoverable when the migration begins acting on real traffic.

6.4 Phase Three — Dual-Write and Per-Workflow Traffic Shift

For each workflow, the team moves the workflow into dual-write mode: the monolith and the new service both produce their outputs, and a workflow engine reconciles. Per-workflow-instance routing decides which writer's output is the response. The traffic fraction per workflow is incremented gradually per cohort.

Dual-write duration is workload-specific. Time-sensitive workflows need shorter dual-write periods and higher reconciliation rates; long-running approval workflows tolerate longer dual-write periods because state is preserved across pauses. The framework does not prescribe a duration; it prescribes the criteria for shortening the dual-write window.

6.5 Phase Four — Cutover and Decommissioning

Once per-workflow traffic has reached 100% on the new service for a configured duration with divergence below threshold, the monolith is cut out of the workflow. The monolith's data persists for a configurable grace period during which reversion is possible; after the grace period, the monolith's data is archived and the monolith's code is retired for that workflow.

Each workflow retired from the monolith is a milestone. The platform's monolith shrinks workflow by workflow. The team celebrates each milestone because each represents a reduction in accidental complexity.

6.6 Phase Five — Continuous Workflow Decomposition

Enterprise operations platforms are not static. New workflow types appear; existing workflows acquire new archetypes; regulatory constraints shift the seam of acceptable decomposition. A team that stops decomposing workflows once the initial migration is complete will find itself rebuilding the monolith over the next several years.

Continuous workflow decomposition is therefore the migration's terminal phase. It is operationally indistinguishable from product engineering — the team decomposes new workflows when they appear, archives workflow types that have become obsolete, and updates the workflow inventory periodically as a hygiene task.

VII. WORKED SCENARIO

To ground the framework, we instantiate it on a realistic enterprise operations platform at a mid-size industrial-supply distributor. The platform is a Java monolith deployed on a single application server with a single relational database. The team has grown from four to twenty-eight over six years, and the release pipeline now runs every two weeks with frequent rollback.

7.1 Workflow Inventory

The team produces a workflow inventory with twelve major workflows. The dominant archetypes are:

- Purchase Order Approval. Long-running approvals, with three approval steps (requester, budget owner, legal review). Invariant: no approval by the requester; no approval past cap.
- Daily Inventory Replenishment. Batch orchestration, runs at 03:00, processes 40,000 SKUs. Invariant: incremental delta correctly captured in accounting.
- Inventory Change Propagation. Event-driven state transitions; inventory updates propagate to pricing, availability, and customer-facing channels. Invariant: convergence within ten minutes.
- Large-Customer Credit Review. Human-in-the-loop review; an officer decides credit worthiness within a structured framework. Invariant: audit traceability of the decision.
- Same-Day Dispatch. Time-sensitive material flows; carrier-pickup deadline drives the workflow. Invariant: no missed pickup within 24 hours of order placement.

The remaining seven workflows also fit archetypes but follow the patterns established by these five. The team adopts the five as the spine.

7.2 Phase One — Boundary Extraction

For Purchase Order Approval, the team draws the boundary after the budget-owner step but before legal review. The new service owns the approval-request data and the two approver steps; the monolith retains legal review. The contract

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

||Volume 7, Issue 1, January-February 2024||

DOI:10.15662/IJRPETM.2024.0701013

between them is explicit: the new service hands off a half-approved PO to the monolith for legal review, and the monolith returns the final outcome for the new service to journal.

For Daily Inventory Replenishment, the team splits a per-SKU worker from the orchestrator. The orchestrator stays in the monolith initially; the worker is a new service. The orchestrator passes a per-SKU task to the worker; the worker returns the delta; the orchestrator persists the delta.

For Inventory Change Propagation, the new service is the event publisher, and downstream consumers are new services that materialize their view from the event stream.

For Large-Customer Credit Review, the new service is the review interface for officers and persists the decision log; the monolith still computes the decision context.

For Same-Day Dispatch, the new service is the dispatch engine; the monolith is the carrier integration point. Reconciliation compares the dispatch engine's record with the carrier's response.

7.3 Phase Two Result

Each new service runs in shadow-read for four to eight weeks. The team discovers divergences in pricing rounding rules in Daily Inventory Replenishment that the monolith has accumulated over time; the new service converges to the canonical rounding rule because the contract specified it. The team discovers that 0.4% of Purchase Order Approval workflows have a corner case where the requester is also the legal reviewer when an organization has only one legal officer; the contract is updated to forbid this configuration.

7.4 Phase Three Outcome

Dual-write periods range from six weeks (Same-Day Dispatch) to twelve weeks (Purchase Order Approval). Traffic fractions increment by cohort. The team observes divergence between monolith and service decrease monotonically.

7.5 Phase Four Outcome

Cutover proceeds workflow by workflow. Same-Day Dispatch is the first to migrate because the dual-write is shortest; the team's morale improves from a successful early milestone. Purchase Order Approval is the last because its dual-write is longest. After eighteen months, the monolith's active workflow count drops to nine of twelve; the team plans the remaining three over the next year. The platform's deployment cadence has improved from every two weeks with frequent rollback to multiple deploys per day with rare rollback.

7.6 Notes on What Did Not Work

Two missteps are worth recording. First, an early attempt at decomposing along code-package boundaries produced a service that spanned workflow boundaries; the team refactored after three months and reverted to workflow-aligned decomposition. The lesson reinforces Phase Zero's centrality. Second, an initial compensation design was not idempotent; a retry produced a duplicate inventory adjustment that took five days to detect. The team rebuilt the compensation with an idempotency table before continuing.

VIII. TRADE-OFFS AND LIMITATIONS

The workflow-oriented approach has known costs. Workflow analysis upfront is expensive. Phase Zero on a non-trivial platform is weeks of meetings and disagreements, with no code shipped during the analysis. Teams under deadline pressure often skip Phase Zero; the cost of skipping is borne later, when migrations reproduce the monolith's properties at a different scale. The decomposition patterns are not equally easy to implement. Bounded-context microservices require workflow-engine adoption; saga patterns require idempotent-compensation infrastructure; event-carried state transfer requires stream infrastructure. Teams that lack the infrastructure must invest in it before they can adopt the pattern.

The framework's recommendations are workload-specific. Workflows with strong transactional invariants across many steps may not be a good fit for any of the patterns. Pure two-phase-commit across services is generally inadvisable, but workflows whose invariants cannot be expressed through saga compensation or single-step atomicity require either a hybrid architecture or a re-examination of the invariant. Regulatory constraints that require synchronous cross-organization visibility may preclude event-driven decomposition. Some regulated workflows require that all participating parties observe a state change synchronously; the event-carried state transfer pattern's eventual-consistency contract is incompatible with this.

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

||Volume 7, Issue 1, January-February 2024||

DOI:10.15662/IJRPETM.2024.0701013

The framework is less prescriptive on organizational change than on architectural change. Architectural decomposition without team-topology decomposition produces services that couple at the workflow level even when code is decoupled. A migration that adopts the framework's patterns without reorganizing teams will see diminishing returns. The literature on team topology and stream-aligned teams is a useful complement.

A final limitation is the framework's dependence on workflow-engine infrastructure. Modern workflow engines (Temporal, Camunda, AWS Step Functions) make the patterns much easier to implement, but a team without such infrastructure must build or adopt one. Building a workflow engine is itself a significant engineering investment, and one that the framework does not address directly. Teams evaluating the framework should evaluate workflow-engine adoption as a coupled decision.

IX. CONCLUSION

Monolith-to-microservices migration in enterprise operations platforms is a workflow re-architecting exercise, not a code refactoring exercise. The five workflow archetypes we have identified — long-running approvals, batch orchestration, event-driven state transitions, human-in-the-loop review, and time-sensitive material flows — correspond to distinct decomposition patterns that preserve the invariants the business depends on. Mapping the workflow inventory to its dominant archetype is the precondition for a migration that improves the platform rather than reproducing the monolith's complications at a different scale.

The migration mechanics — workflow-aware dependency analysis, contract-first API design, idempotent compensation, gradual traffic shifting, in-flight workflow-state observability, and forward-and-backward-compatible schema evolution — determine whether the chosen decomposition pattern actually reaches production without losing invariants. A migration that lacks any one of these mechanics is fragile; a migration that lacks them all is rebuilding the monolith.

The phased migration framework orders the work according to workflow properties, with workflow inventory and archetype classification as Phase Zero and continuous workflow decomposition as Phase Five. The framework does not promise a particular timeline; it promises a particular ordering of decisions and a discipline of analysis before code.

Future work includes evaluating the framework against additional enterprise operations platforms to test the robustness of the five-archetype taxonomy, developing quantitative rigor around the Phase Three dual-write duration criteria, and studying the organizational and cultural conditions under which Phase Zero can be performed without becoming the migration's bottleneck. Across all of these, the central architectural insight is that microservice boundaries derive from workflow seams, and a migration that does not identify its workflow seams first has not yet decided what it is migrating to.

REFERENCES

[1] M. Fowler, "Strangler Fig Application," martinfowler.com, 2004.
[2] C. Richardson, "Pattern: Saga," microservices.io, 2018.
[3] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003.
[4] S. Newman, Building Microservices, 2nd ed., O'Reilly Media, 2021.
[5] Microsoft Learn, "Strangler Fig pattern."
[6] M. Fowler, "Branch by Abstraction," martinfowler.com.
[7] "Evolutionary Database Design," martinfowler.com.
[8] Temporal Technologies Documentation, "Workflows."
[9] AWS Step Functions Developer Guide, Amazon Web Services, 2023.