ARCHITECTURE RETROSPECTIVE

Rebuilding a Two-Decade-Old Telecom Workflow for 2026

What I would preserve, what I would replace, and why the right modernization starts with state, boundaries, and failure—not a catalog of fashionable tools.

By Chandrakanth Anne | Enterprise Applications Architect | August 2026

More than twenty years ago, I helped lead a three-person team that automated the full lifecycle of telephone-equipment orders: analyst queues, manager approvals, document retrieval, business-process routing, and mainframe transactions. The technology belonged to its era. The architectural problem still belongs to ours.

> **The central lesson:** modernization is not a translation exercise. Replacing JSP with React and EJBs with services changes syntax. A durable redesign changes where workflow state lives, how external effects are retried, how identity is enforced, and how operators understand failure.

The original system solved a genuinely hard problem

The platform coordinated work between analysts and managers from intake through completion. It retrieved supporting documents, moved work items through a business-process model, and exchanged transactions with a mainframe. Its presentation layer used server-rendered Java pages and an MVC framework; its middle tier used enterprise Java components on an application server; messaging connected asynchronous work; and a relational database stored operational data. A digitally signed browser applet opened the mainframe interaction window.

For the early 2000s, that was a serious enterprise architecture. It centralized transaction management, offered managed deployment, represented workflow explicitly, and used asynchronous messaging where remote work could not be treated as an immediate local call. Those instincts remain sound. What has changed is the quality of the boundaries we can put around them.

What I would keep

- An explicit process model. The business journey must be visible, versioned, testable, and understandable to both operations and engineering.
- Asynchronous integration. A mainframe transaction is not a browser interaction and should never be coupled to a user session.
- A relational system of record. Orders, assignments, approvals, and audit facts need strong consistency and queryable history.
- Document separation. Documents and their lifecycle deserve a dedicated storage and metadata boundary instead of being incidental application blobs.
- A human-in-the-loop design. Exceptions, escalations, approvals, and accountability are part of the product—not operational leftovers.

What I would change first

I would remove every direct dependency between the browser and the mainframe. I would move long-running process state into a durable workflow engine, isolate domain rules from process diagrams, publish integration facts through an outbox rather than dual writes, and make identity and telemetry part of the platform contract. I would also resist the reflex to create a microservice for every box in the workflow.

A pragmatic reference architecture

The architecture below is intentionally modern but not maximalist. It begins with a modular core, scales the components with genuinely different workloads, and keeps the workflow engine, event broker, and systems of record in distinct roles.

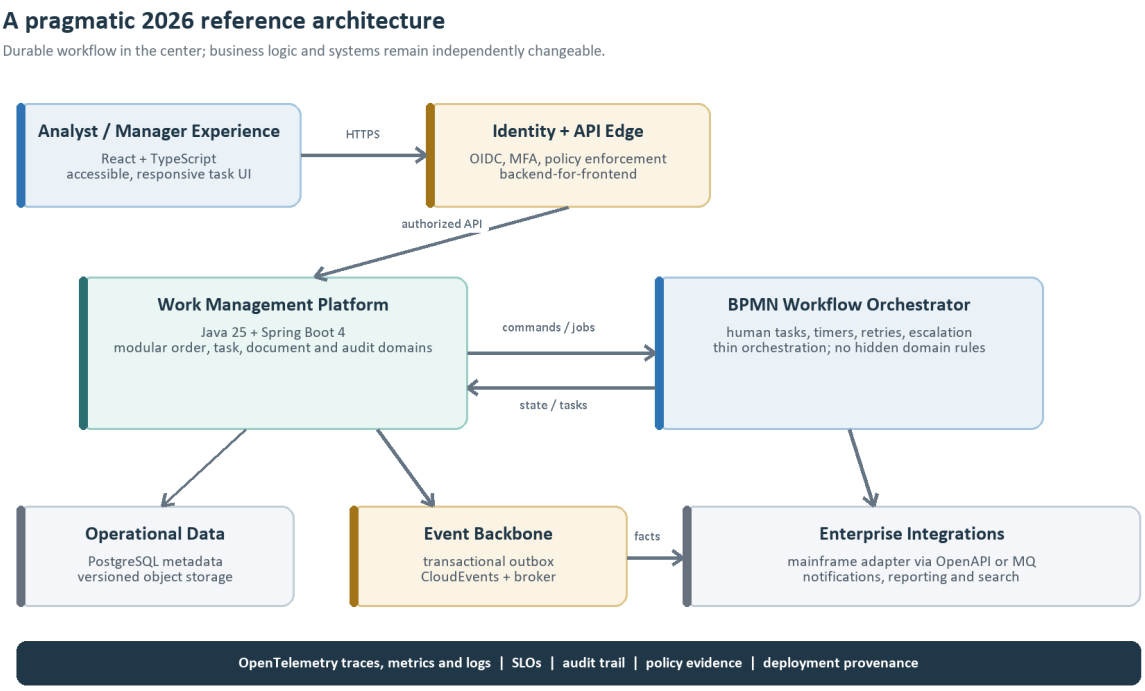

Figure 1. A modern order-processing workflow separates durable orchestration, domain logic, data, integrations, and operational evidence.

1. A secure, browser-native work experience

Analysts and managers would use an accessible React and TypeScript interface, backed by a small backend-for-frontend. That backend would shape task views, enforce server-side authorization, and prevent the browser from learning internal service topology. Authentication would use the organization's identity provider through OpenID Connect, with MFA and short-lived sessions. Authorization would combine roles with business attributes such as queue, geography, order type, and approval limit.

The signed applet disappears entirely. Mainframe access happens behind a controlled server-side adapter. If a user must make a legally significant approval, the platform records the identity, authentication context, intent, timestamp, and immutable audit evidence; if a true electronic signature is required, it uses an approved signing service rather than downloadable client code.

2. A modular core before a microservice fleet

For a team of three, I would start with Java 25 LTS and Spring Boot 4, organized as a modular application. Order intake, work management, documents, reference data, notifications, and audit would be explicit modules with enforced dependency rules. Spring Modulith is one current option for making those boundaries executable and testable.

This is not nostalgia for a monolith. It is a way to preserve a fast local transaction path, simple deployment, and coherent domain behavior while the system is still finding its shape. A module becomes a separately deployed service only when it has an independent scaling profile, security boundary, release cadence, or ownership model. The architecture earns distribution; it does not assume it.

3. Durable BPMN orchestration for people and systems

This problem is a strong fit for BPMN-based orchestration because the workflow contains human tasks, timers, escalations, exception paths, and system steps that operations teams need to see. A current platform such as Camunda 8 can keep process state durable while workers execute business operations outside the engine. The process model should coordinate work; it should not become a hidden repository for pricing, eligibility, or order-validation rules.

A code-centric durable workflow platform such as Temporal is a credible alternative when developers own the process and visual business modeling is less important. For this case, I would choose BPMN because the process itself is a shared artifact between engineering and operations. The selection criterion is collaboration and operational clarity—not which engine has the longest feature list.

4. An anti-corruption layer around the mainframe

The mainframe remains a system of record, but its native transaction model no longer leaks into the user interface or the domain model. A dedicated adapter exposes a stable OpenAPI contract through a mainframe API layer where available, or bridges to existing enterprise queues when messaging is the safer integration. IBM z/OS Connect is one example of exposing mainframe assets through OpenAPI-compatible REST interfaces without forcing every caller to understand native data structures.

Every command carries an idempotency key and correlation identifier. Retries are bounded and policy-driven. Timeouts move work to an observable exception state. Dead-letter handling is paired with replay tooling and reconciliation, not treated as a message graveyard. The platform records both the business outcome and the technical attempt history so support teams can answer two different questions: what happened to the order, and what happened to the integration?

5. Data, documents, events, and audit each have a job

PostgreSQL would hold transactional order metadata, assignments, approvals, idempotency records, and the outbox. Versioned object storage would hold documents with checksums, retention policies, malware scanning, and legal-hold support where required. A search index would be introduced only for search workloads that the transactional database should not serve.

When a transaction changes the order and must also notify other capabilities, it writes the business change and an outbox record atomically. Change-data capture publishes the event to a broker after commit. Consumers are idempotent, event schemas are versioned, and a CloudEvents envelope gives events consistent identity, source, type, and time metadata. This removes the classic failure window in which the database commits but the message does not—or vice versa.

6. Observability becomes part of the workflow contract

A trace should follow an order from browser request to process instance, worker, mainframe call, event publication, and notification. OpenTelemetry provides a vendor-neutral foundation for traces, metrics, and logs. Technical telemetry is then joined with business measures: work-item age, queue depth, approval time, retry count, reconciliation drift, and the percentage of orders requiring manual intervention.

The most useful dashboard is not a collection of CPU charts. It answers whether orders are moving, where they are waiting, what failures are accumulating, and whether operators can safely recover them. Service-level objectives should cover both API latency and end-to-end process timeliness.

7. Cloud-native, without making Kubernetes the objective

The application, workers, and adapters would ship as signed container images with software bills of materials, automated vulnerability checks, and deployment provenance. They could run on Kubernetes when the enterprise already operates it well and needs standardized scaling, policy, and rollout controls. A managed container platform is often the better first home for a small team. Cloud-native should mean automated, observable, resilient, and replaceable—not 'Kubernetes at any cost.'

The 2003-to-2026 translation

The exact products will change. The architectural responsibilities should not.

Then	A 2026 implementation	Why the boundary changes
JSP, Struts, browser applet	React + TypeScript, backend-for-frontend	A secure browser experience with server-side policy; no client-to-mainframe coupling.
EJBs on an application server	Java 25, Spring Boot 4, modular domain core	Lower deployment friction and explicit module boundaries before selective service extraction.
Enterprise BPM suite	BPMN orchestration with external workers	Durable human and system workflows without hiding domain logic in the process engine.
JMS and enterprise queues	Broker plus retained MQ bridge where needed	Events for broad distribution; queues for controlled legacy interoperability.

Then	A 2026 implementation	Why the boundary changes
Oracle operational database	PostgreSQL by default; keep Oracle where constraints justify it	Modernization should not create database migration risk without a business reason.
Image-service connector	Versioned object storage + document metadata service	Independent scaling, integrity checks, retention, and lifecycle controls.
XML exception catalog	Problem-details responses, domain error codes, trace IDs	Errors become stable API contracts and operationally searchable evidence.
Centralized version configuration	Git, CI/CD, infrastructure as code, signed releases	Every change is reviewable, reproducible, promotable, and attributable.

Illustrative stack as of August 2026; product versions should be governed through an enterprise support policy, not copied from an article.

What I would not do

- I would not rewrite everything at once. A big-bang cutover converts known constraints into unknown failure modes.
- I would not make each BPMN step a microservice. Workflow granularity is not a deployment topology.
- I would not let the browser call the mainframe, even through a modern-looking API gateway.
- I would not dual-write the database and broker and hope distributed timing behaves.
- I would not place generative AI on the approval path without deterministic guardrails, human accountability, and recorded evidence.
- I would not confuse containerization with modernization. The hard work is domain separation, recoverability, and operational design.

Where AI belongs—and where it does not

AI can add value at the edges of this workflow: classifying incoming documents, extracting candidate metadata, summarizing exception history, drafting analyst notes, and identifying unusual retry patterns. Each use should publish confidence, provenance, model and prompt versions, and the human decision that followed.

AI should not silently approve an order, alter a mainframe transaction, or reinterpret a policy. Deterministic rules and explicit human authority remain the control plane. An AI suggestion is evidence presented to a decision-maker, not a substitute for the decision contract.

A migration path that protects the business

1. Map the current process and failure modes. Capture real order states, queue behavior, manual workarounds, reconciliation steps, and operational metrics before changing the implementation.
2. Establish stable interfaces. Put an API or queue adapter around mainframe operations and define idempotency, timeout, and error contracts.
3. Build a read-only work experience. Reproduce queues and document views first, validating authorization and data fidelity without taking control of transactions.
4. Move one bounded order type. Run a new process model and worker path for a low-risk cohort with explicit rollback and reconciliation.
5. Publish facts through the outbox. Feed reporting, notifications, and search from committed events instead of point-to-point database access.
6. Expand by capability, not by layer. Migrate complete vertical slices and retire old paths only after operational evidence proves equivalence.

How I would test it

The test strategy would treat the workflow as a system. Process-model tests would verify timers, boundary events, escalations, and compensation. Contract tests would protect mainframe and document interfaces. Integration tests would run the application, database, broker, workflow engine, and adapters together in disposable environments. End-to-end tests would exercise the analyst and manager journeys through the browser.

The most valuable tests would inject failure: duplicate messages, delayed replies, expired credentials, mainframe timeouts, partial document uploads, worker restarts, and event redelivery. A modern architecture is not proven by the happy path. It is proven by its ability to resume without losing intent or producing an untraceable side effect.

The enduring idea

Two decades ago, the system's purpose was to make a complicated, document-heavy, human-and-mainframe process visible and executable. That purpose has not changed. The best 2026 architecture would not discard the original insight; it would make that insight more durable, secure, observable, and easier to evolve.

> **If I were leading the project today:** I would put durable workflow at the center, keep domain logic in well-bounded application modules, isolate the mainframe behind idempotent adapters, use events for committed facts, and make identity, audit, and observability non-negotiable. I would

> start simpler than the final topology and let evidence—not fashion—decide what must become distributed.

Sources and further reading

These official sources informed the current technology and architecture references in this article:

- Oracle Java SE Support Roadmap — identifies Java 25 as an LTS release.
- Spring Boot system requirements — documents the current Spring Boot 4.1 line and supported Java range.
- Spring Modulith reference — describes an opinionated toolkit for domain-driven modular Spring applications.
- Camunda 8 architecture — describes clients, gateways, brokers, exporters, workers, partitions, and fault-tolerant workflow execution.
- Temporal Workflows — documents code-defined durable workflows and event-history replay.
- OpenID Connect Core 1.0 — defines interoperable authentication on top of OAuth 2.0.
- NIST SP 800-207A — applies identity-centric zero-trust principles to cloud-native applications.
- IBM z/OS Connect overview — describes OpenAPI-based access to mainframe applications and data.
- CloudEvents — defines a common envelope for describing event data.
- Debezium Outbox Event Router — documents a change-data-capture implementation of the transactional outbox pattern.
- OpenTelemetry — provides a vendor-neutral observability framework for traces, metrics, and logs.
- Kubernetes concepts — provides the official conceptual model for container orchestration when its operational cost is justified.