A Unified Software Architecture Theory

Authority, Invariants, and Operational Boundaries in Distributed Systems

Tomer Ben-David, AI

April 2026

Abstract

Modern software systems become hard to reason about when business invariants, authoritative state, and transition rules are spread across too many infrastructure boundaries. When one business operation is split across services, queues, workflows, callbacks, and analytical systems without a clear authority for what must remain coherent, the system becomes harder to verify, operate, and change safely.

This whitepaper presents the Unified Software Architecture Theory (USAT), a practical theory for keeping business authority clear in distributed systems. USAT keeps business logic in cohesive services, keeps authoritative truth in a directly queryable transactional center, and assigns asynchronous and analytical systems to explicit supporting roles. Its central claim is simple: architecture should be organized around business invariants first and infrastructure primitives second.

The theory is illustrated through an Autonomous Financial Risk & Payment Orchestration (AFRP) example with financial correctness, asynchronous review flows, human intervention, callbacks, and real time dashboards. It shows how to define the transactional center, how to accept asynchronous inputs without losing authority, and how to separate action systems, real time reads, and historical analytics from the core write path.

Executive Summary

USAT is an intent first architectural theory for systems that distribute one business operation across several technical boundaries. It is for teams that need correctness, operator legibility, and maintainability without giving up queues, workflows, callbacks, or analytics. Its value is a disciplined rule for where business truth lives, where business logic lives, and how the rest of the system scales around them.

1. Introduction

The core problem this whitepaper addresses is not infrastructure choice by itself. It is the loss of coherent business truth when one business operation is distributed across services, queues, workflows, callbacks, and analytical systems without one clear authority for state, rules, and transitions.

USAT does not argue that all processing, teams, or systems should collapse into one place. It argues that even in a distributed architecture, each important business operation still needs one clear authority for the state, rules, and transitions that must remain coherent.

A workflow that is conceptually one business action is often implemented as API calls, event publications, background workers, third party callbacks, and analytical projections. Each component may work in isolation while the overall business operation becomes difficult to reason about, debug, and keep consistent.

This is not an argument for an undifferentiated monolith. It is an argument that boundaries should follow invariants and authority rather than default infrastructure decomposition.

2. Scope and Central Claim

The Unified Software Architecture Theory (USAT) is a practical design theory for systems whose business operations cross several technical boundaries and still need one coherent way to place authority, explain current state, and evolve safely. It is most useful when a system combines business flows that must remain authoritative, asynchronous subsystems that react to those flows, and a need to keep business truth legible as the surrounding architecture

becomes more distributed. Typical examples include payments, risk and compliance platforms, order and inventory systems, logistics workflows, and other software where business truth must remain queryable even as the system scales outward.

The central claim of USAT is:

> When a business operation contains a small set of state transitions that must remain mutually consistent, that operation should have one authoritative transactional center. Distribution should happen around that center, not through it.

That center is more than a storage boundary. It is also the place where the owning service keeps the business rules, where authoritative truth remains queryable, and where operators can explain what the system currently believes.

The contribution is practical. This whitepaper does not claim a new formal theory of distributed systems. It combines several known ideas into one architectural decision framework centered on business invariants.

USAT is therefore not a reliability pattern for event publication. The transactional bridge is one mechanism inside it. The larger theory is about business authority, service owned logic, operational legibility, and system boundaries in distributed software.

2.1 USAT At A Glance

USAT is best understood as one connected decision framework:

Component	What it answers	What USAT says
Business invariants	What must remain true?	Name the business invariants before choosing infrastructure boundaries.
Service owned business logic	Where do the rules live?	Keep business rules and state transitions in cohesive services, not in queue topology, workflow definitions, or callback glue.
Transactional center	Where does authority live?	Keep mutually dependent state transitions and authoritative truth in one queryable transactional center.
Transactional bridge	How does committed truth leave the center?	Bind committed business state and outbound intent in one local transaction, typically through an outbox or equivalent local commit pattern.
External reentry rule	How does async work become truth?	Accept callbacks, manual decisions, and worker results only through a new local commit.
Operational legibility	How is the system explained in production?	Keep current state, accepted inputs, and pending actions explainable from the authoritative system of record.
Four plane model	Where does the rest of the system go?	Let actions, real time reads, and historical analytics scale outward in separate operational planes.
Contract separation	What should other systems depend on?	Keep public event contracts separate from raw internal table replication.

The transactional bridge is one mechanism inside this structure. It matters because it binds committed truth to outbound intent, but it is not the whole theory.

3. Running Example of Autonomous Financial Risk and Payment Orchestration

Consider an Autonomous Financial Risk & Payment Orchestration (AFRP) system.

This system must support all of the following:

- users uploading invoices or proof of funds documents,
- asynchronous OCR and document extraction,
- fraud and risk analysis,
- manual compliance review for flagged cases,
- payment authorization and settlement tracking,
- asynchronous bank callbacks or settlement imports,
- executive and customer dashboards with low latency reads.

This is a useful stress case because it combines concerns that often pull architecture in different directions at the same time: financial correctness, asynchronous external dependencies, human review loop transitions, high concurrency reads, long term analytics and auditability, and the operational need for troubleshooting, clarity, and maintainability. The challenge is preserving coherent business truth while the system interacts with many asynchronous actors.

To make the example concrete, the whitepaper assumes a small set of explicit invariants:

1. A cleared risk decision must never exist without an authoritative payment record or an explicit recorded reason why payment did not proceed.
2. A settlement update must never silently overwrite an incompatible prior payment state.
3. A manual review decision must become queryable business truth in the core system rather than remain trapped inside a workflow tool.
4. A support or operations engineer should be able to explain the current state of a document case from the authoritative system of record without reconstructing the flow from multiple infrastructure logs.
5. The main business decisions for the case should remain in cohesive service logic rather than being hidden inside queue topology, workflow definitions, or infrastructure callbacks.

For the rest of this whitepaper, the AFRP system serves as the running example used to anchor the theory concretely.

4. Positioning Relative to Existing Architectural Ideas

USAT does not reject familiar patterns. It orders them around one question: where does authoritative business truth live, and what should scale around it?

Existing idea	What USAT keeps	What USAT adds
Domain driven design	Business language, explicit domain boundaries, and domain owned rules	Tie those ideas directly to transactional authority, operational legibility, and distributed system boundaries
Modular monolith	Strong local consistency for tightly coupled transactional flows	Use shared physical boundaries only where invariants justify them
Transactional outbox	Reliable bridge from committed state to asynchronous delivery	Treat the bridge as one component inside a broader authority model
CQRS	Separation between write truth and read models	Keep read separation subordinate to business invariants and operational legibility
CDC	Useful replication path for search, analytics, and projections	Keep raw domain replication separate from public event contracts
Stream processing	Strong support for reactions, policy evaluation, and automation	Keep stream systems reactive rather than authoritative for core business truth
Data intensive application design	Strong treatment of data models, logs, replication, batch and stream processing, and system tradeoffs	Recenter those mechanics around business authority rather than infrastructure shape alone

Existing idea	What USAT keeps	What USAT adds
Warehouse / lakehouse	Deep historical analysis and large scale reporting	Keep long horizon analysis outside the transactional and operational read path

USAT turns these ideas into one ordered operating model: establish the transactional center around business invariants, keep the important rules in cohesive services, commit truth locally, export intent deliberately, and let specialized downstream systems scale around that center.

5. The Problem of Fragmented Business Truth

Many systems are built by mapping each workflow step directly onto an infrastructure primitive. A request writes a row, publishes to a queue, triggers storage events, starts a workflow, calls third party APIs, and updates downstream dashboards.

That style can scale operationally, but it has a common failure mode: the business process becomes distributed in ways that separate invariants, authority, and transition logic from the services that should own them, and the business logic itself starts to migrate out of cohesive services and into infrastructure choreography.

5.1 A Common Failure Pattern

In a typical fragmented implementation:

1. The core API writes an initial `document_cases` row and then separately tries to publish a queue event.
2. The user uploads a file to object storage, which emits a storage event outside the authoritative write path.
3. `ocr-worker` starts extraction work from that storage event and writes results into a separate workflow or worker history.
4. `risk-worker` calls the external risk provider and may store partial results outside `risk_decisions`.
5. A payment-facing service creates `payment_intents` based on asynchronous progress inferred from prior events.
6. A later bank callback updates payment state through callback glue before the rest of the case state is fully reconciled.
7. A reviewer approves or rejects the case through a separate review tool, creating business history outside the main system of record.
8. `operations-dashboard`, `customer-status-view`, and `reconciliation-warehouse` reconstruct status from these scattered signals.

Each step is individually reasonable. The problem is that the system may no longer have one place that can answer basic operator questions:

- What is the current business state?
- Why is it in that state?
- Which next actions are pending?
- Which actions were already committed?
- Which transitions are valid from here?
- Where does the actual business logic for this transition live?

When those answers are spread across queue state, workflow history, object storage notifications, service logs, and partial database rows, the system becomes harder to operate than the business problem itself.

5.2 The Dual Write Failure

The most familiar instance of this problem is the dual write problem (see References 7, 9, 10, and 11):

- the application commits business state in the database, and
- the application independently publishes the corresponding message over the network.

If one succeeds and the other fails, the system diverges. Retries, tracing, and orchestration can reduce the blast radius, but they do not remove the underlying inconsistency window.

5.3 Implanted Architecture as a Failure Mode

Here, **Implanted Architecture** refers to an architecture in which core business intent is embedded primarily in infrastructure choreography rather than in a cohesive, queryable application state model.

The term is not meant as a blanket rejection of queues, workflow engines, or cloud primitives. It names a specific failure mode: infrastructure becomes the place where the business process actually lives.

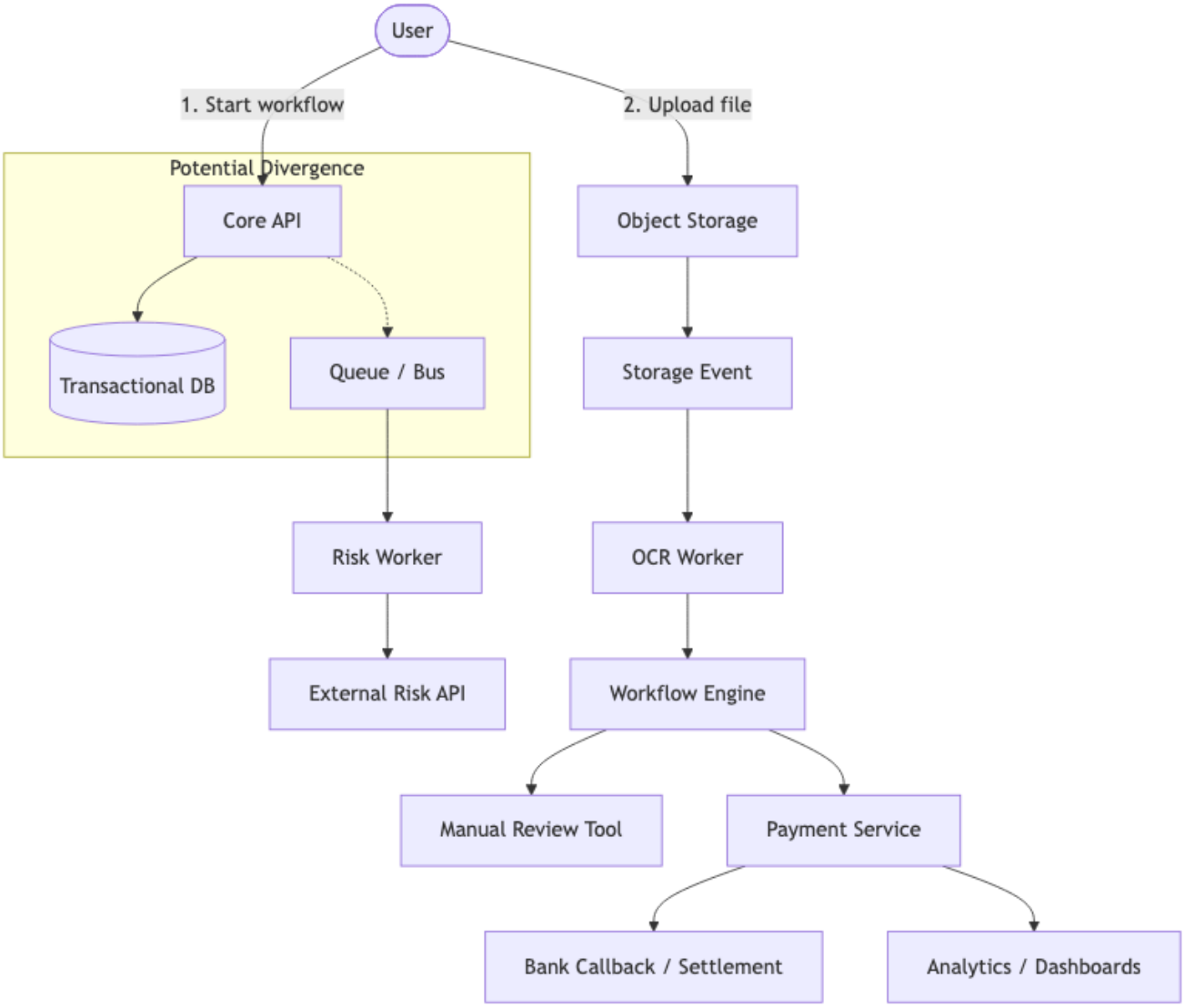

Figure 1: Figure 1. Schematic fragmented workflow.

The first figure is intentionally schematic. It shows fragmentation of authority and business flow, not every queue, topic, retry path, or transport hop that may exist in a real deployment. The deeper problem is that authority and explanation become scattered across several systems. The second figure makes that failure mode explicit.

6. The USAT Response

The starting question is simple:

> Which state transitions must remain mutually consistent for the business to remain correct?

Those transitions define the **transactional center**. The related business logic should remain in the services that own those transitions rather than being dissolved into transport systems.

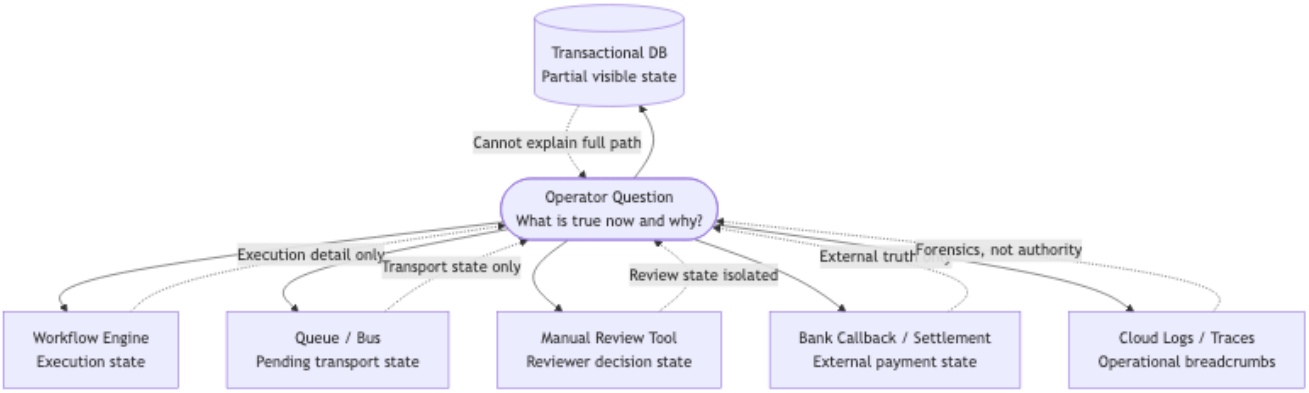

Figure 2: Figure 2. Authority fragmentation and operational blind spots.

That response has three parts:

- keep the important business reasoning in cohesive services,
- keep authoritative truth and auditability in one queryable transactional boundary,
- let asynchronous and analytical systems scale around that boundary in explicitly supporting roles.

In the AFRP example, some operations are naturally asynchronous, such as OCR, external fraud scoring, or bank settlement callbacks. But when those inputs are accepted as new business truth, that acceptance should happen through one local database transaction that records:

- the authoritative state mutation,
- the audit context,
- the publish intent for downstream consumers.

This gives the system one queryable center of truth while still allowing the rest of the architecture to scale outward. It also preserves a clear place where the business logic can be read, tested, and reasoned about as application code rather than inferred from infrastructure behavior.

6.1 The Same Workflow in USAT

The same AFRP flow looks different when the architecture is organized around one authoritative center:

1. The core API calls the `Case Service`, which writes `document_cases`, `audit_log`, and `outbox_events` in one local commit.
2. Document upload still happens through object storage, but accepted document state reaches the transactional center only when `Case Service` or `Risk Decision Service` validates it and writes it locally.
3. `ocr-worker` and `risk-worker` still run outside the write path, but they submit candidate results back to the authoritative center instead of defining business truth themselves.
4. `Risk Decision Service` records accepted risk state in `risk_decisions` only after checking the current case state and business rules.
5. `Payment Service` writes `payment_intents`, `audit_log`, and `outbox_events` in the same transactional center when the invariant requires risk and payment state to remain coherent.
6. Manual reviewer actions reenter through the transactional center as `manual_review_actions`, so the accepted decision becomes queryable business truth instead of staying inside the review tool.
7. Settlement callbacks are validated by `Payment Service` against current payment state and accepted through a new local commit into `settlement_events` and `payment_intents`.
8. `notification-worker`, `operations-dashboard`, `customer-status-view`, and `reconciliation-warehouse` consume committed truth from outside the center rather than becoming the place where truth is decided.

The difference is not that the asynchronous parts disappear. The difference is that one part of the system remains clearly authoritative while the rest reacts, projects, or analyzes around it.

6.2 Shift in Responsibility and Authority

The fragmented version and the USAT version may involve many of the same technologies, but they assign responsibility differently.

In the fragmented version, queues, workflow engines, storage events, and callbacks collectively determine business progress. The team often has to reconstruct the real workflow by correlating several systems, and the answer to "why is this case in its current state?" is spread across infrastructure history.

In the USAT version, those same asynchronous systems still exist, but they do not own authoritative truth. The important difference is that business state changes become explicit local commits in the transactional center, and asynchronous systems either propose inputs, react to committed truth, or project it into read and analytical planes.

That shift produces four concrete advantages:

1. The authoritative state is queryable in one place instead of inferred from event trails.
2. Business rules remain readable in service code instead of being distributed across workflow definitions and queue topology.
3. External callbacks, worker results, and manual actions become explicit reentry points with validation instead of silent state mutations.
4. Operators and developers can explain the current state from the system of record and audit trail rather than reconstructing it from infrastructure choreography.

7. Service Owned Business Logic and Operational Legibility

Transactional discipline solves only part of the problem. The theory also depends on where business reasoning lives and how the system remains explainable after distribution.

7.1 Service Owned Business Logic

The core business rules of a workflow should live in cohesive application services that own the relevant state transitions.

That means the service code should answer questions such as:

- when a case is allowed to move from `UNDER_REVIEW` to `RISK_ACCEPTED`,
- when a payment authorization may be created,
- when a settlement callback is compatible with the current state,
- when manual review is required before any further action is taken.

Those rules should not be hidden primarily inside:

- queue topology,
- workflow engine definitions,
- storage event chains,
- retry configuration,
- or infrastructure specific callback glue.

Infrastructure can trigger work, transport events, and coordinate background execution. It should not become the place where the business policy itself is encoded.

In AFRP, the fact that a bank callback arrived is not itself the business decision. The business decision is whether that callback is valid for the current payment state, whether it advances the case, whether it requires review, and what new authoritative truth should be recorded. That reasoning belongs in the service that owns payment and case state.

> A useful design check is simple: if the team has to inspect queue graphs, workflow history, and cloud event chains to explain why a case entered its current state, then business logic has already leaked too far into infrastructure.

7.2 Operational Legibility

USAT also insists that a production system remain legible to the people operating it.

Operational legibility means a support engineer, operator, or developer should be able to answer the following from the authoritative system of record and its audit trail:

- what the current business state is,
- why the system believes that state is correct,
- which external or asynchronous inputs have already been accepted,
- which next actions are pending or already requested,
- and which service owned rule produced the last important transition.

This requirement is not cosmetic. It changes architecture.

If the only way to explain a case is to combine database rows, queue offsets, workflow engine history, object storage notifications, and ad hoc service logs, then the architecture is no longer preserving a coherent operational model.

For AFRP, operational legibility means an operator should be able to inspect a document case and see, in one authoritative place, the accepted risk decision, the payment state, the last reviewed action, the audit justification for the last transition, and the downstream intent that was already committed. The operator may still use workflow tools and dashboards for context, but those tools should enrich the picture rather than define the truth.

Taken together, service owned business logic and operational legibility explain why USAT is broader than a durability pattern. The theory is trying to preserve coherent business reasoning, clear authority, and explainable state as the system grows.

8. The Transactional Bridge and Event Contracts

At the mechanics level, the most important bridge pattern in USAT is the transactional outbox. It sits alongside a second concern: event contract discipline. Together, they define how committed truth leaves the transactional center without turning raw internal schema into the public architecture.

8.1 Why the Outbox Matters

The outbox is not primarily about elegance. It is about binding two things together in one commit:

- "the business state changed", and
- "the system intends to publish that fact."

Instead of updating the database and separately publishing to Kafka, SQS, or another bus, the application writes the business change and the outbound event record in the same local transaction. A publisher or CDC process then relays the outbox record downstream (see References 7, 9, 10, and 11).

If the transaction fails, neither state nor intent is committed. If the transaction succeeds, both are durable.

8.2 Canonical Local Commit

```sql
BEGIN;

UPDATE payments
SET status = 'AUTHORIZED'
WHERE id = 'pay_789';

INSERT INTO payment_audit_log (payment_id, actor_id, action, timestamp)
VALUES ('pay_789', 'user_42', 'PAYMENT_AUTHORIZED', now());

INSERT INTO payment_outbox (aggregate_type, aggregate_id, event_type, payload, metadata, created_at)
VALUES ('Payment', 'pay_789', 'PaymentAuthorized', '{...}', '{...}', now());

COMMIT;
```

This pattern gives three useful properties:

1. The business state is immediately queryable in the transactional store.
2. The audit trail is recorded at the same boundary as the state mutation.

3. The intent to notify downstream systems is durable and can be delivered asynchronously.

8.3 Public Events and Domain CDC

A common alternative is to stream changes directly from domain tables through CDC. That can be useful, but it should not automatically become the public event contract. That distinction follows naturally from outbox and CDC patterns: one exists to publish intentional external events, while the other is often used for replication and projection (see References 7, 9, 10, and 11).

If downstream consumers depend directly on internal table schemas, then ordinary refactoring of internal columns becomes externally breaking behavior.

USAT therefore separates:

- **Outbox events for public or boundary crossing intent**
- **Domain table CDC for internal replication and analytical synchronization**

This distinction keeps internal schemas free to evolve while maintaining stable external contracts.

8.4 When Direct Domain CDC Is Appropriate

Direct domain CDC remains useful for internal use cases such as:

- replicating dimensions into analytical stores,
- hydrating search indexes or read replicas,
- forensic row level auditing.

The rule is simple: use the **outbox** for intent you want other systems to consume as a stable contract, and use **domain CDC** for internal state replication.

For AFRP specifically, the outbox would carry events such as `RiskReviewRequested`, `RiskDecisionRecorded`, `PaymentAuthorized`, or `SettlementRecorded`, while domain CDC might still replicate normalized tables like `customers`, `vendors`, or `payments` into analytical systems for joins.

9. External Inputs

External signals require a different discipline because they do not originate inside the transactional center. Some of them arrive from outside it:

- object storage notifications,
- bank webhooks,
- settlement files,
- manual reviewer actions.

Those signals are not in the same database transaction as the core system, and the theory does not pretend otherwise.

The rule is instead:

> When an external or asynchronous signal changes business truth, the receiving boundary must perform one local transaction that records the new truth, the relevant audit context, and any resulting publish intent.

That applies to:

- OCR completion becoming accepted structured data,
- manual approval or rejection,
- bank settlement or failure notifications.

This includes handling asynchronous issues gracefully. If a downstream worker fails, that failure does not roll back an already committed core transaction. Instead, the failure must be handled through an explicit follow up path such as retry, manual review, or a new domain level state transition recorded through another local commit. The key point is not that every failure becomes the same business state. The key point is that once a failure becomes authoritative business truth, it should be recorded through the same clear transactional boundary.

USAT does not eliminate asynchrony. It gives asynchrony a clean point at which it becomes authoritative business state, without conflating infrastructure failure with domain failure.

10. Shared Database and Distributed Sagas

This section does not claim that all systems should share one database. It makes a narrower claim:

When two or more domain operations must remain strongly consistent within one user visible business action, physically splitting them across independent database commits often introduces avoidable complexity.

In those cases, a modular monolith or shared transactional center may be the simpler and safer design, especially when the alternative is to push one user visible operation across saga style coordination boundaries (see References 4 and 8).

That means:

- logical module boundaries should remain strict,
- code ownership should remain explicit,
- but physical database isolation should not be treated as a virtue by default.

The tradeoff is real. Shared databases can constrain autonomous deployment and increase coupling if boundaries are weak. But distributed sagas also have a cost: more inconsistency windows, more compensation logic, more debugging, and more difficulty establishing what the business currently believes to be true (see References 4 and 8).

USAT prefers to pay the local modularity cost when the alternative would be to distribute an operation whose correctness depends on atomicity.

In AFRP, that means risk acceptance and payment creation may belong to one transactional center if the business cannot tolerate a cleared risk state without a corresponding authoritative payment record. If the business can tolerate eventual consistency there, the boundary could be split. The architecture should follow that business fact rather than a preset doctrine.

11. The Four Plane Model

Once the transactional center is defined, USAT separates the rest of the system into four operational planes. The clearest way to understand those planes is not as one more workflow, but as four distinct ownership boundaries that sit side by side around the same business truth.

The four plane model is as central to USAT as the transactional bridge. It defines where different kinds of responsibility should live:

- Transactional Center (Plane 1) owns authoritative business truth.
- Action Systems (Plane 2) own asynchronous reactions and automated actions.
- Real Time Reads (Plane 3) own fast analytical reads for dashboards.
- Historical Analytics (Plane 4) own long horizon analysis and historical depth.

Without this separation, teams tend to collapse transactional concerns, action concerns, and analytical concerns into one architecture, or they spread core business decisions across infrastructure layers that were never meant to own them.

11.1 Transactional Center (Plane 1)

The Transactional Center is where authoritative business truth lives.

- Typical technology: PostgreSQL or another relational OLTP store
- Responsibilities: state transitions, invariants, audit records, outbox writes
- Optimization target: correctness, low latency local commits, clear write ownership

It should remain narrow. This is the place for business truth, not for arbitrary side effects or heavy analytics.

In AFRP, Plane 1 would contain the canonical document case state, risk decision acceptance, payment authorization state, settlement acceptance, and the audit and outbox records associated with those changes.

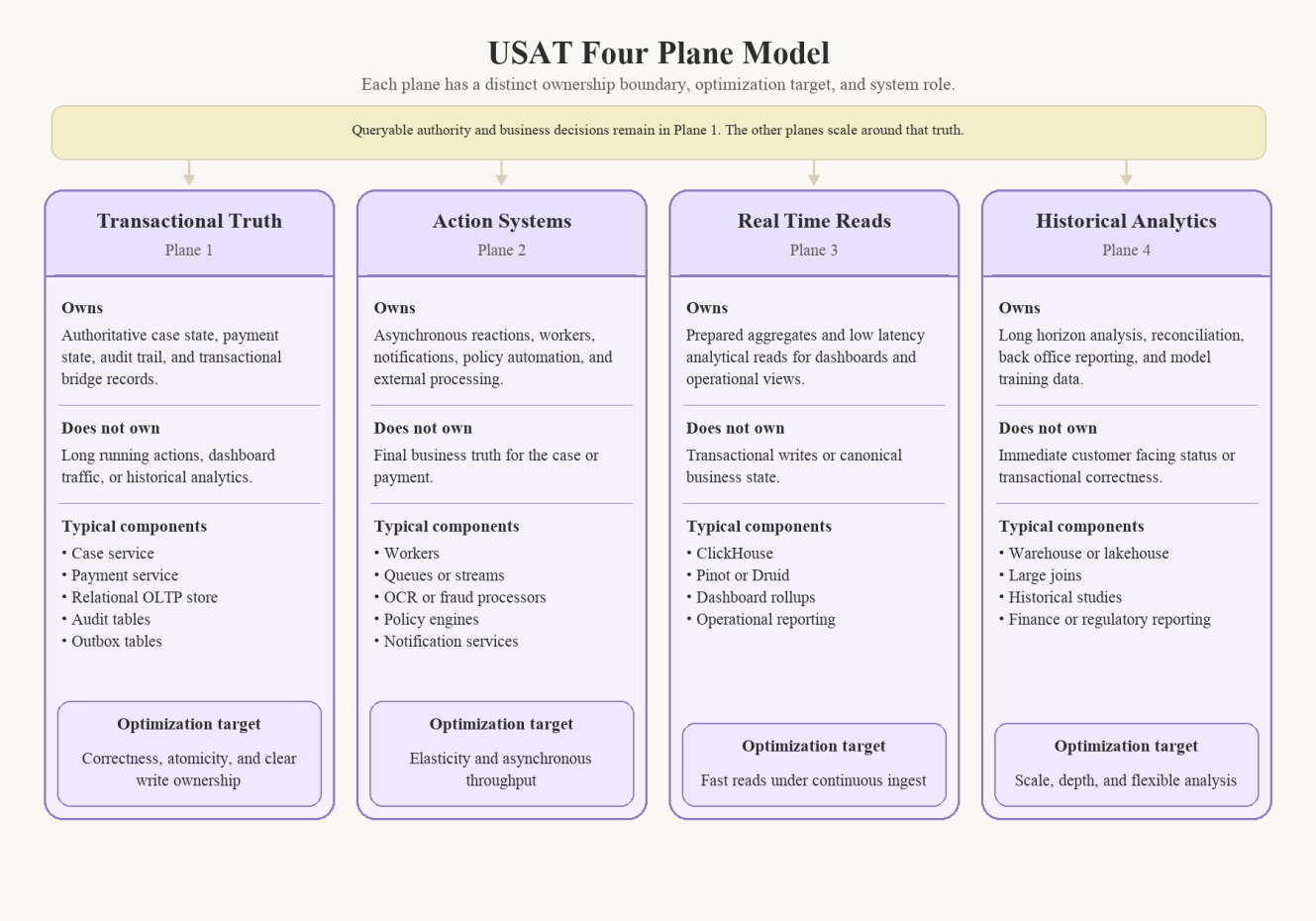

Figure 3: Figure 3. USAT four plane model and ownership boundaries.

11.2 Action Systems (Plane 2)

Action Systems consume committed events and perform asynchronous reactions.

- Typical technology: Kafka, queue systems, stream processors, background workers
- Responsibilities: fraud scoring, alerting, policy reactions, notifications, downstream automation
- Optimization target: elasticity, asynchronous processing, autonomous reactions

Stream processors such as Flink are useful here when the job is to detect patterns and trigger automated actions. The key point is that this plane should react to committed truth rather than define the truth of the core transaction itself (see Reference 12).

In AFRP, Plane 2 would include OCR processing, fraud scoring, alert generation, duplicate document detection, policy reactions, and notifications to internal reviewers or downstream systems.

11.3 Real Time Reads (Plane 3)

Real Time Reads serve low latency dashboards and high concurrency analytical reads.

- Typical technology: ClickHouse, Pinot, Druid, or PostgreSQL with a real time analytical extension when the workload remains narrower
- Responsibilities: aggregated metrics prepared in advance, dashboard queries, customer and executive reporting
- Optimization target: fast reads under sustained ingest

Read optimization matters more here than truth ownership. Modern PostgreSQL stacks can go much further than older architecture advice often assumes, especially when they add continuous aggregates, columnar storage, partitioning, or other real time analytical extensions. For teams with moderate scale and a relatively narrow dashboard workload, that can be a very good Plane 3 choice because it preserves operational simplicity. Once the read path becomes heavily analytical in concurrency, retention, scan volume, or aggregation breadth, a purpose built analytical engine remains the clearer fit (see Reference 13).

In AFRP, Plane 3 would serve dashboards such as approval funnel counts, payment volume summaries, settlement lag, flagged case throughput, reviewer workload, and customer visible payment status timelines.

11.4 Historical Analytics (Plane 4)

Historical Analytics serves long range analysis, back office reporting, model training, and deep historical queries.

- Typical technology: BigQuery, Snowflake, Redshift, data lakehouse systems
- Responsibilities: ad hoc analysis, large joins, historical depth, data science workloads
- Optimization target: scale and flexibility over per request latency

Customer facing dashboard traffic is often a poor fit here, but large historical analysis belongs in this plane.

In AFRP, Plane 4 would support monthly reconciliation, fraud model training datasets, longitudinal customer risk studies, retention analysis, and regulatory or finance reporting over long historical ranges.

Implementation guidance, the end to end AFRP flow, and the concrete AFRP architecture sketch are included in the appendices so the main body can stay focused on the theory itself.

The same separation disciplines the read path. The transactional center should not become the general purpose engine for multi year analytics or high concurrency dashboards. Committed events and selected CDC streams should feed analytical systems, real time dashboards should read from Real Time Reads (Plane 3), long range analysis should read from Historical Analytics (Plane 4), and the OLTP system should remain focused on correct writes and narrow operational reads.

12. Tradeoffs and Limits

No universal prescription is claimed here.

It is strongest when:

- the business contains high value invariants,
- correctness matters more than maximal deployment independence,

- the number of strongly coupled transactional operations is small enough to keep the core narrow,
- the system still needs large downstream analytical or asynchronous capabilities.

It is less compelling when:

- the domains are naturally independent,
- eventual consistency is acceptable in the user visible workflow,
- cross domain atomicity is not a real business requirement,
- the organizational cost of a shared core outweighs the consistency benefit.

The theory therefore recommends deliberate centralization only where invariants justify it.

13. Comparison Criteria for Common Alternatives

The cleanest way to evaluate USAT is to compare it against a few common alternatives on explicit criteria rather than broad ideology.

Approach	Main strength	Main weakness	Best fit
USAT / Intent First Transactional Center	Strong business truth, clear state ownership, better troubleshooting, and deliberate scaling around the center	Requires discipline about what belongs in the center and what must remain asynchronous	Systems with high value invariants and meaningful operational complexity
Service per Database plus Saga Coordination	High deployment autonomy and clear service ownership	More inconsistency windows, more compensation logic, and harder end to end reasoning	Naturally independent domains where eventual consistency is acceptable
Queue First / Fully Async Workflow	Strong decoupling and elastic ingestion	Weaker synchronous UX and more difficulty exposing immediate authoritative state	Fire and forget or ingestion heavy workflows where immediate entity truth is not required
Single OLTP for Writes and Reads	Simple initial architecture and fewer moving parts	Read workloads can distort write path latency and operational safety over time	Small systems with modest scale and limited analytical pressure

The point is not to declare one universal winner. The point is to make the decision rule explicit: when correctness, legibility, and authoritative state matter more than maximum deployment autonomy, USAT becomes more attractive.

14. Conclusion

The main argument of USAT is straightforward: software architecture should be organized around business intent, not around the accidental shape of infrastructure.

Queues, workflow engines, object storage, stream processors, and analytical warehouses are all valuable. The mistake is letting those tools become the place where the business process actually lives.

A system remains easier to reason about when it has one authoritative and queryable center for business truth, cohesive services that own the important business decisions, one explicit bridge from committed truth to asynchronous distribution, and clearly separated planes for actions, real time reads, and historical analytics.

USAT is therefore not against cloud systems and not against distribution. It is a proposal for where distribution should begin: after the system has already established, in one durable place, what it believes to be true. Appendix B through Appendix D show how to apply that rule in practice without changing the theory-led focus of the main text.

15. References

1. Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2003.
2. Martin Kleppmann, *Designing Data-Intensive Applications.* O'Reilly Media, 2017.
3. Vaughn Vernon, *Implementing Domain-Driven Design.* Addison-Wesley, 2013.
4. Martin Fowler, "Monolith First." https://martinfowler.com/bliki/MonolithFirst.html
5. Martin Fowler, "CQRS." https://martinfowler.com/bliki/CQRS.html
6. Martin Fowler, "What do you mean by 'Event-Driven'?" https://martinfowler.com/articles/201701-event-driven.html
7. Chris Richardson, "Pattern: Transactional outbox." https://microservices.io/patterns/data/transactional-outbox
8. Chris Richardson, "Pattern: Saga." https://microservices.io/patterns/data/saga.html
9. AWS Prescriptive Guidance, "Transactional outbox pattern." https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/transactional-outbox.html
10. Debezium Documentation, "Outbox Quarkus Extension." https://debezium.io/documentation/reference/stable/integrations/outbox.html
11. Debezium Documentation, "Outbox Event Router." https://debezium.io/documentation/reference/stable/transformations/outbox-event-router.html
12. Apache Flink Documentation, "Overview." https://nightlies.apache.org/flink/flink-docs-stable/docs/concepts/overview/
13. ClickHouse Documentation, "Materialized Views." https://clickhouse.com/docs/materialized-views

Appendix A Key Terms

- **Business Invariant:** A statement that must remain true for the business to remain correct, regardless of infrastructure shape.
- **Authoritative State:** The state the system recognizes as the source of truth for current business meaning.
- **Service Owned Business Logic:** The principle that business rules and state transitions should live in cohesive application services rather than in transport or orchestration systems.
- **Transactional Center:** The narrow part of the architecture where mutually dependent state transitions are committed together and where authoritative business state is owned.
- **Transactional Outbox:** A table written in the same local transaction as a business mutation, used to persist outbound publish intent durably.
- **Implanted Architecture:** A failure mode in which core business logic is expressed mainly through distributed infrastructure choreography instead of a cohesive application state model.
- **Operational Legibility:** The ability for engineers and operators to explain current business state and pending actions from the authoritative system of record.
- **Public Event Contract:** The stable event shape intentionally exposed to downstream consumers.
- **Domain CDC:** Internal change capture from domain tables, useful for replication or analytics but not automatically suitable as the public contract.
- **Four Plane Model:** The separation of the transactional center, event driven actions, real time analytical reads, and batch analytics into different operational planes.

Appendix B Applying USAT

B.1 Implementation Rules

1. Identify the business invariants before choosing the deployment boundary.
2. Keep core business rules in cohesive services rather than distributing them across transport and orchestration systems.
3. Keep state transitions that must remain mutually consistent inside one local transaction whenever practical.
4. Make authoritative business truth directly queryable and auditable from the transactional system of record.
5. Write outbound intent in the same transaction as the state mutation.
6. Accept external signals only by converting them into one new authoritative local commit.
7. Separate public event contracts from internal table schemas and keep analytics and asynchronous actions outside the transactional write path.
8. Preserve operational legibility so current state and pending actions remain explainable without reconstructing the business process from infrastructure history.

B.2 How To Apply USAT

1. Start by naming the business operation that must remain coherent. In AFRP, that might be the path from risk acceptance to payment authorization to settlement acceptance.
2. Write down the invariants that cannot be violated without causing real business damage. These should be phrased as concrete statements such as "a cleared risk decision must not exist without an authoritative payment record."
3. Identify the authoritative entities that carry those invariants. In AFRP, those would include the document case, the risk decision, the payment state, and the audit record.
4. Define the transactional center around the state transitions that must remain mutually consistent. This is the boundary where those entities are updated together and where auditability is recorded.
5. Move the business rules for those transitions into cohesive services. If a transition still depends on queue topology, workflow definitions, or callback glue, the design is not finished yet.
6. Add the transactional bridge that records outbound intent in the same local commit. Only after that commit succeeds should downstream actions react asynchronously.
7. Define the external reentry points. Each callback, manual review action, OCR result, or settlement import should become authoritative only through a new local commit.
8. Separate the downstream responsibilities by plane. Decide which consumers belong in action systems, real time read systems, and historical analytics.
9. Check operational legibility before calling the design complete. An operator should be able to inspect the authoritative system of record and explain the current business state without reconstructing the workflow from scattered infrastructure tools.

B.3 Implementation Checklist

Before calling a USAT design complete, a team should be able to answer yes to most of the following:

- Is there one clearly named business operation the design is trying to keep coherent?
- Are the business invariants written down explicitly rather than assumed implicitly?
- Is there one authoritative system of record for the state transitions that must remain consistent?
- Do the key business rules live in cohesive service code rather than queue topology, workflow definitions, or callback glue?
- Does every important asynchronous result reenter the system through a validating local commit?
- Is outbound intent recorded in the same transaction as the state mutation it depends on?
- Are public event contracts separated from raw internal table replication, and are dashboards and historical analytics clearly outside the transactional write path?
- Can an operator explain the current state and pending actions from the authoritative system of record?
- Can the team name the authoritative tables and local commits for its most important business operation?

Appendix C AFRP End To End Flow

1. A user submits a payment case and uploads supporting documents. The core service creates the document case in Plane 1 and records the initial audit context in one local transaction.
2. The same transaction records outbound intent for downstream work. Outbox events request OCR, risk review, or policy checks without making those downstream systems the source of business truth.
3. Plane 2 workers process the asynchronous tasks. OCR extraction, fraud scoring, or duplicate detection run outside the transactional write path.
4. A worker or callback produces a result that matters to business state. The result does not become authoritative merely because a worker finished. It becomes authoritative only when the receiving service validates it and commits it locally.
5. Manual review enters through the same rule. A reviewer decision is accepted through Plane 1, recorded in audit state, and exposed as queryable business truth rather than being left inside the review tool.
6. Payment authorization is committed in the transactional center. If the business requires risk acceptance and payment authorization to remain coherent, they are committed together or in one tightly controlled local boundary.
7. Settlement callbacks reenter through a new local commit. The service checks that the callback is compatible with the current payment state, records the accepted transition, and emits any further downstream intent.

8. Plane 3 and Plane 4 update outward from committed truth. Dashboards, operational analytics, and historical reporting consume committed events or selected CDC streams without becoming the system of record.

At the end of this flow, an operator should be able to open the authoritative case record and answer three questions quickly: what state the business believes, why it believes it, and what downstream actions have already been requested.

Appendix D Concrete AFRP Architecture Sketch

To make the implementation more explicit, the AFRP example can be mapped onto a small, concrete architecture with named services, authoritative data structures, transaction boundaries, and event contracts.

D.1 Plane 1 Authoritative Services

The transactional center can be implemented as one deployable application with strongly separated modules, or as a very small set of tightly coordinated services that still share one authoritative transactional database.

For the AFRP example, a practical Plane 1 split would be:

- `Case Service` Owns document case creation, lifecycle state, and the overall authority for the case.
- `Risk Decision Service` Owns accepted risk outcomes, reviewer decisions, and the logic that determines whether a case may proceed.
- `Payment Service` Owns payment authorization state, settlement acceptance, and payment related invariants.
- `Audit Service` Owns the append only audit record and the metadata needed to explain accepted transitions.
- `Outbox Publisher` Relays committed outbox rows downstream after the local transaction commits.

Those modules may live inside one modular application and one transactional database if the business requires their core state transitions to remain coherent.

D.2 Plane 1 Authoritative Database

A concrete authoritative database for AFRP could be PostgreSQL with tables such as:

- `document_cases` Canonical case record. Holds case identifier, current lifecycle status, customer reference, and ownership metadata.
- `risk_decisions` Accepted risk outcome for the case, including decision status, risk score, reviewer source, and effective timestamp.
- `payment_intents` Canonical payment authorization state, amount, currency, beneficiary, and current payment lifecycle status.
- `settlement_events` Accepted settlement updates that have been validated against the current payment state.
- `manual_review_actions` Reviewer decisions and escalation actions that have been accepted into business truth.
- `case_documents` Metadata about uploaded documents and accepted extraction results.
- `audit_log` Append only record of the accepted transitions, actor identity, reason, and trace metadata.
- `outbox_events` Durable outbound intent written in the same local transaction as the business mutation.

In this setup, the operational question "what does the business currently believe about case `case_123`?" should be answerable from `document_cases`, `risk_decisions`, `payment_intents`, `settlement_events`, `manual_review_actions`, and `audit_log` without needing queue history to reconstruct the truth.

D.3 Plane 2 Action Systems

The asynchronous action layer can be made explicit as a set of consumers that do not own authoritative state:

- `ocr-worker` Consumes `DocumentUploaded` or `ExtractionRequested`, extracts fields, and submits structured results back through a validating Plane 1 write.
- `risk-worker` Consumes `RiskReviewRequested`, calls external fraud or sanctions providers, and submits proposed results back through Plane 1.
- `notification-worker` Consumes `ManualReviewRequested`, `PaymentAuthorized`, or `SettlementRecorded` and sends email, Slack, webhook, or reviewer notifications.

- `policy-worker` Consumes committed events and performs pattern detection or secondary automation without changing Plane 1 truth directly.

These workers may use Kafka, SQS, or another queue system, but their outputs are authoritative only when the receiving Plane 1 service validates and commits them.

D.4 Plane 3 Real Time Read Systems

The real time read layer can be made explicit as a separate serving system for dashboards and operational views:

- `operations-dashboard` Serves reviewer workload, approval funnel counts, settlement lag, and flagged case throughput.
- `customer-status-view` Serves customer visible payment status timelines and case progress views.
- `read-aggregator` Consumes committed events or selected CDC streams, builds prepared aggregates, and publishes read optimized projections.

These services may use ClickHouse, Pinot, Druid, or PostgreSQL with a real time analytical extension when the workload remains narrower. Their role is fast analytical reads, not authoritative business truth.

D.5 Plane 4 Historical Analytics

The historical analytics layer can be made explicit as a separate long horizon system:

- `reconciliation-warehouse` Serves monthly reconciliation, finance reporting, and regulatory analysis.
- `risk-research-datasets` Serves fraud model training data, longitudinal risk studies, and deep historical joins.
- `backoffice-analytics` Serves retention analysis, cohort analysis, and ad hoc investigation over long time ranges.

These systems may use BigQuery, Snowflake, Redshift, or a lakehouse stack. Their role is historical depth and flexible analysis, not operational truth or customer facing status.

D.6 Public Events

For AFRP, a concrete public event contract might include:

- `CaseCreated`
- `DocumentUploaded`
- `ExtractionAccepted`
- `RiskReviewRequested`
- `RiskDecisionRecorded`
- `ManualReviewRequested`
- `ManualReviewRecorded`
- `PaymentAuthorized`
- `SettlementRecorded`
- `PaymentFailed`

Those events belong in `outbox_events`. By contrast, raw CDC from tables such as `document_cases` or `payment_intents` belongs to internal replication and analytics, not to the public contract by default.

D.7 Example Transaction Boundaries

A reader implementing the system should be able to name the exact local commits:

- `CreateCaseAndRequestReview` Writes `document_cases`, initial `audit_log`, and `outbox_events`.
- `AcceptRiskDecisionAndAuthorizePayment` Writes `risk_decisions`, `payment_intents`, `audit_log`, and `outbox_events` in one local transaction when the invariant requires them to stay coherent.
- `RecordManualReviewDecision` Writes `manual_review_actions`, updates `document_cases` or `risk_decisions`, writes `audit_log`, and emits any next-step intent.
- `AcceptSettlementCallback` Validates callback compatibility, writes `settlement_events`, updates `payment_intents`, writes `audit_log`, and records downstream publish intent.

If a team cannot name its authoritative services, tables, public events, and local commits at this level of specificity, the design is still too abstract.