# Transactional Integrity and Agent Coordination in AI Operating Systems: Architectural Requirements Across CRM, ERP, and Commerce

November 2026


## Abstract

Several agents can make individually reasonable decisions that collectively exceed stock, credit, or fulfilment capacity. This paper examines the transaction boundaries required when an AI operating system coordinates customer relationship management, enterprise resource planning, and commerce. It develops a commitment model connecting intent, validation, reservation, execution, and reconciliation. The analysis draws on database isolation, transactional messaging, and idempotent interface design, then applies those mechanisms to a constructed order workflow. The proposed contribution is an explicit separation of the business commitment from the conversational task that initiates it. The paper identifies conditions under which local atomic transactions are sufficient and conditions that require coordination across services. It also proposes an evaluation protocol using concurrent orders, changed policies, duplicate requests, and uncertain external outcomes. MatruOS is used as an architectural context. No production performance or experimental results are reported. The conclusion is that agent coordination should be evaluated by preserved business constraints and verifiable commitments across the whole workflow.



Keywords: transaction integrity, concurrency, agent coordination, reservations, idempotency, enterprise workflows.


## 1. A shared view does not settle a shared commitment

Two sales agents receive requests for the same product. Each reads a stock record showing ten available units. One prepares an order for seven units and the other prepares an order for six. Both calculations are correct relative to the value each read, but accepting both commitments would require thirteen units.

This constructed example exposes a problem that access to a common database does not resolve by itself. Information can be shared while decisions remain uncoordinated. The relevant question is what happens between reading availability and reserving the resource. A business system needs a rule that prevents concurrent actions from jointly violating the stock constraint.

Credit exposure presents a related problem. Two orders can each fall below a customer’s remaining credit while their combined value exceeds it. A fulfilment service can face the same issue with a delivery slot. These are questions about state transitions and shared resources. Language model reasoning should operate within mechanisms that enforce the relevant conditions.

The research question is how an AI operating system can preserve business constraints while agents act concurrently across customer relationship management (CRM), enterprise resource planning (ERP), and commerce. The scope covers commitments that modify business state. Informational responses are considered where their wording could imply that a commitment has already been made.

## 2. Established mechanisms and the proposed synthesis

PostgreSQL's transaction isolation documentation describes how concurrent transactions can observe and affect data under different isolation levels. Serializable isolation constrains successful concurrent outcomes to those consistent with a serial ordering, while applications must handle transaction failures and retries. These are database guarantees within defined transaction boundaries. [1]

AWS guidance on the transactional outbox addresses the problem of updating a database and notifying another service. The proposed mechanism records the business change and the intended message within one local transaction. Publication occurs subsequently, and consumers must account for possible duplicate delivery. [2]

Featonby's discussion of idempotent APIs connects repeated requests to a caller supplied identifier representing the same intent. This supports safe recovery from uncertain responses when the service contract and retention conditions are respected. It does not imply that every operation across a workflow becomes globally atomic. [3]

This paper combines these mechanisms into a business commitment model. The individual mechanisms are established engineering practice. The proposed contribution is their explicit connection to an agent's authority, the user's intended transaction, and the evidence required before the system communicates completion. The method is architectural analysis with constructed examples and a proposed experiment.

## 3. Defining the business commitment

### 3.1 Intent and identity

A request begins as an expression of intent, such as "confirm this order for delivery tomorrow." Before execution, it must resolve to a customer, products, quantities, price terms, destination, and the relevant order version. Ambiguity in any field that changes the obligation should be resolved before commitment.

The workflow needs a stable operation identifier associated with that intended transaction. Rewording the same request in a conversation should not create a second order automatically. Conversely, a genuinely new order for the same customer and products must remain possible. Equality of text or field values is therefore insufficient to determine whether two requests represent the same intent.

The operation record should connect the user instruction, proposed transaction, approvals, and execution attempts. An agent can revise its plan, but the record must show whether it is continuing the same operation or initiating a new one. This distinction becomes essential during recovery and dispute investigation.

### 3.2 Preconditions and invariants

A precondition states what must be true before an action is attempted. An invariant states what must remain true across permitted changes. For the illustrative stock workflow, a precondition might require an active product and a valid customer. An invariant might require that committed reservations do not exceed allocatable quantity under the defined inventory policy.

The invariant must reflect the business. Some firms permit backorders, substitute products, or reserve expected receipts. A blanket rule that stock can never be negative may be inappropriate if it confuses physical inventory with future commitments. The implementation should define the relevant quantity and the consequences of each status.

Credit, pricing, and delivery constraints need similar precision. A credit rule should specify whether exposure includes open orders, dispatched goods, invoices, and unapplied payments. An agent should not choose that interpretation during execution. The authorised business rule should determine it consistently.

### 3.3 Reservation and confirmation

A reservation records a claim on a resource before the final obligation is completed. Its scope, duration, and release conditions need to be explicit. A stock reservation may expire when a quotation is not accepted, while a reservation for a confirmed order may persist until fulfilment or authorised cancellation.

Confirmation should communicate the actual state achieved. “Order received,” “stock reserved,” and “dispatch confirmed” have different meanings. A conversational interface can explain these states in ordinary language, but it must not report completion beyond the evidence returned by the execution service.

Where a reservation has an expiry time, the system should handle actions arriving near that boundary. A customer acceptance received after expiry may require a new availability check. A worker continuing from an old plan should not be able to restore a reservation that another process has legitimately released.

## 4. Coordination inside a transactional boundary

The business service should validate and mutate the relevant state within a transaction appropriate to the invariant. The exact implementation may use guarded updates, locks, or serializable transactions. The choice depends on contention, data relationships, and the database’s semantics. The language model should call a constrained service rather than compose arbitrary state changes.

For the stock example, the service can accept a reservation only if the quantity remains available at the point of change. If another order consumes the remaining stock, the second operation receives a defined conflict result. The agent can then explain the outcome or seek an authorised alternative. It should not keep changing parameters until a forbidden operation happens to pass.

A transaction should avoid waiting for an extended model response or human approval while holding scarce database resources. Interpretation and approval can occur before the short commitment transaction. The service must then revalidate conditions that could have changed during that interval.

This separation creates a useful distinction between a proposed decision and an executable decision. A manager may approve a discount for a particular order version, but a subsequent change to quantity or destination can invalidate the approval. The transaction checks the approval’s scope against the current request.

Concurrency control also affects fairness. A technically valid first arrival policy may repeatedly favour one sales channel or branch. If the business requires allocation priorities, those priorities need an explicit scheduling or reservation policy. Database integrity alone does not decide how scarce resources should be distributed.

## 5. Coordination across services

### 5.1 Where local atomicity ends

An internal order update, a payment request, and a courier booking may involve separate services with different failure behaviour. A successful local transaction cannot guarantee that every external effect occurred. The workflow therefore needs durable intermediate states and procedures for discovering what happened outside its boundary.

The outbox mechanism supports this separation by preserving the intent to notify after the local change. [2] The application should track publication and acknowledgement without assuming that a single message delivery means the entire business process is complete. An acknowledgement may confirm receipt rather than execution.

Ordering requirements should also be explicit. A cancellation received before a delayed creation event can produce an inconsistent outcome if the consumer ignores event version or dependency. The workflow should define which transitions are valid and how obsolete messages are recognised.

### 5.2 Duplicate delivery and repeated requests

An idempotent operation recognises repeated attempts for the same intended effect. Its identity should include the appropriate business and tenant scope. Reusing an identifier with changed parameters should produce a conflict requiring resolution, rather than silently substituting a new transaction for the original.

The retention period for deduplication matters. A retry after the external service has discarded its key may behave as a new request. The local workflow should retain sufficient evidence to reconcile the operation before retrying outside the supported window. A key is a mechanism within a contract, not a permanent proof that duplication is impossible.

A single order may contain several operations, each needing its own identity. Payment capture, shipment creation, and customer notification should not share one undifferentiated retry state. Their outcomes and correction procedures differ, even though they belong to the same customer commitment.

### 5.3 Compensation and irreversible effects

If stock reservation succeeds but payment authorisation fails, releasing the reservation may be an appropriate corrective action. If goods have already been dispatched, correction requires a different process. The workflow must know which completed actions can be reversed and which create an obligation requiring human handling.

A compensating action is a new business event with its own authority and evidence. It should not erase the original event. A refund, for example, needs a relationship to the payment and a record of why it occurred. An attempted compensation can also fail, leaving work that remains visible and assigned.

This leads to a proposed design requirement: every consequential operation should declare its recovery class. The class can identify whether the operation is safely repeatable, queryable after uncertainty, compensable, or subject to manual resolution. These categories support planning without pretending that all failures have the same remedy.

## 6. Worked scenario with concurrent orders

Assume a distributor has ten allocatable units and two incoming orders, for seven and six units. Both customers have valid terms. The reservation service processes the first operation and records seven units against its order. The second operation encounters only three remaining units and returns a conflict.

The second agent may offer partial fulfilment only if policy and customer consent permit it. It cannot transform the requested six units into three merely to achieve a successful tool call. The proposed alternative becomes a revised intent requiring the appropriate confirmation. The record retains the original request and the accepted change.

Suppose the first customer’s payment response times out. The workflow keeps the order in an uncertain payment state and queries the provider using the recorded operation identifier. It does not release stock immediately if payment may have succeeded, nor does it submit a new capture with a fresh identifier without investigation.

Now assume that confirmation messaging fails after the order is committed. The communication operation is retried independently. Replaying the whole order creation would risk duplication. The customer-facing status should distinguish a confirmed order from a message whose delivery is pending.

Finally, suppose a credit policy changes while the second customer considers partial fulfilment. The service revalidates the revised order at commitment. Whether an earlier approval remains valid depends on its recorded scope and policy. This scenario shows how concurrency, policy changes, and recovery interact within one business process.

## 7. Application to an AI operating system

For MatruOS, the architectural implication is that shared business modules should expose controlled operations such as reserve stock, approve terms, and record fulfilment. A model can propose a sequence, but the services determine whether each transition is valid. The application should return structured status and evidence that the conversational layer can report accurately.

The design should also separate the authority of an agent from that of the user who initiated it. A user may be entitled to request a purchase without being entitled to approve it. Delegating the request to an agent should preserve that boundary. Background workers should receive only the authority needed for their operation.

Observability should connect the conversation to the operation and its downstream effects. A support employee investigating an order needs to see which attempt committed, which notifications remain pending, and whether a later attempt was rejected as a duplicate. Model transcripts alone do not provide this business state.

The proposed architecture is applicable to both shared database and integrated service deployments. Physical consolidation can change where transactions are available, but the evaluation remains focused on preserved constraints and the evidence of external effects. Product implementation claims require their own inspection and tests.

## 8. Proposed research protocol

The experiment should begin with a documented business model and a set of invariants. Generate concurrent requests that compete for the same stock, credit, or delivery capacity. Include ordinary cases, contention near limits, repeated user instructions, delayed responses, and changes between approval and execution.

Compare the same workflow with and without the proposed coordination controls while keeping models, request distributions, and service contracts constant. This comparison should occur in a test environment using synthetic records. The uncontrolled condition should never be used to create real customer commitments.

Record invariant violations, duplicate effects, abandoned reservations, incorrect completion messages, and time to a verified terminal state. A terminal state may be completed, rejected, or assigned for manual resolution. Reporting only successful orders would omit failures that consume resources or leave obligations unresolved.

Tests should continue after the initial response. Delayed events and retries may create violations that are absent at the moment the agent finishes speaking. Reconciliation should inspect the final records across every participating service and identify unresolved differences.

The protocol should also measure operational costs: database contention, retry volume, review time, and time resources remain reserved. A design that preserves stock by holding reservations indefinitely may prevent overselling while obstructing legitimate business. Integrity and resource availability must be reported together.

## 9. Limitations and conclusion

This analysis assumes that the business can define its constraints and identify authoritative services. Conflicting definitions of stock availability or credit exposure require organisational resolution. A transaction mechanism cannot determine which commercial interpretation is intended.

The framework does not guarantee successful completion during every external outage. It aims to preserve a truthful account of the operation and prevent unsupported commitments while recovery proceeds. Some actions require manual resolution, particularly when external effects are irreversible or cannot be queried reliably.

Transactional integrity in an AI operating system depends on the relationship between intent, authority, shared resources, and recorded effects. The proposed commitment model makes that relationship explicit. Its contribution should be tested through concurrent and interrupted workflows, including the period after a conversational response has been delivered.

An enterprise can delegate execution responsibly when it can identify what was requested, which conditions were checked, what actually changed, and what remains unresolved. These are the conditions under which coordinated agents can support dependable business operations.

## Author affiliation and disclosure

The author is affiliated with Code Colonies Private Limited, Ahmedabad, and leads the MatruOS team. MatruOS is discussed as an architectural context. No independent product evaluation is reported.

## References

[1] PostgreSQL Global Development Group. PostgreSQL 18 Documentation. Section 13.2, Transaction Isolation.

https://www.postgresql.org/docs/18/transaction-iso.html

[2] Amazon Web Services. Transactional outbox pattern. AWS Prescriptive Guidance. Accessed 2 November 2026.

https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/transactional-outbox.html

[3] Featonby, M. Making retries safe with idempotent APIs. Amazon Builders’ Library. Accessed 2 November 2026.

https://aws.amazon.com/builders-library/making-retries-safe-with-idempotent-APIs/