Engineering the Agentic Enterprise

Semantic MVCC: Transactional State Management for Agentic Systems

Rajesh Iyer
iyer70@gmail.com

December 2025

Abstract

Agentic systems require coordination mechanisms that traditional database concurrency control cannot provide. While Data MVCC resolves conflicts through data representation—normalizing facts to canonical addresses where conflict is structural—agentic systems face a fundamentally different challenge: multiple reasoning components reading shared state and emitting intents that may not both be permissible given system constraints. We introduce **Semantic MVCC**, a framework for transactional state management in agentic systems. The core insight is that agentic memory should serve as a *reasoning surface* where state, recall, inference, and update are unified operations. We define Semantic ACID guarantees, present a practical implementation based on append-only intent logging with deterministic state derivation, and demonstrate the framework through insurance underwriting and trading desk case studies. The architecture requires no novel primitives—it composes event sourcing, snapshot isolation, and domain-specific resolution functions into an auditable, replayable system suitable for regulated industries.

1 Introduction

Agentic AI promises transformative value for enterprises: autonomous claims processing, algorithmic trading, intelligent underwriting, adaptive compliance monitoring. Yet between laboratory demonstrations and production deployment lies a chasm that current frameworks cannot bridge.

The rush to deploy agentic systems has driven early stories of retreat. Organizations that moved quickly discovered that impressive demos do not translate to production trust. The obstacle is not the capability of individual reasoning steps—large language models perform impressively on isolated tasks. The obstacle emerges at **enterprise scale**, where real-world solutions require dozens of specialized components operating concurrently: risk assessors, fraud detectors, compliance validators, portfolio managers, execution optimizers. Each component reasons competently in isolation. But when they must coordinate against shared state, the system breaks down.

Regulated industries require guarantees that current agentic frameworks do not provide:

- **Consistent state.** When multiple reasoning components operate concurrently, what state does each observe? Can a risk agent and an alpha agent see different portfolio positions mid-cycle?
- **Deterministic replay.** When a regulator asks "why did the system make this decision on March 15th?", can you reconstruct the exact state and reasoning? Or is the answer "we logged some things but cannot reproduce it"?
- **Auditable resolution.** When components disagree, what decides? Is the resolution logic explicit, documented, and consistent—or emergent from prompt engineering?

Without these guarantees, enterprises cannot deploy agentic AI at scale. The components reason; the system does not cohere.

Every production agentic system is inherently multi-component. Even a single "agent" decomposes into planner, executor, critic, tool-caller, and memory manager—each making decisions against shared state. Traditional approaches treat agent memory as retrieval: store facts in a vector database, fetch context, inject into prompts. This imports none of the transactional guarantees enterprises require.

We propose **Semantic MVCC**, applying concurrency control semantics to agentic coordination:

> *Data MVCC detects address collision. Semantic MVCC detects intent sets that violate system constraints at commit time.*

2 Background: Data MVCC

Multi-Version Concurrency Control provides non-blocking reads through versioning. The mechanism works because of a critical property: **each fact has exactly one canonical address**.

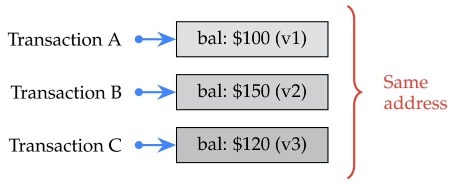

Figure 1: Data MVCC: conflict is structural. Multiple transactions writing to the same row create versions; the system selects which version is visible.

When transactions modify `account.balance`, conflict is detected through pointer comparison—they write to the same address. Resolution is version selection: last-writer-wins, first-committer-wins, or abort. The schema does the work.

3 The Agentic Problem

Agentic systems break this assumption. Consider insurance underwriting with three components: **Risk** (recommends premium), **Fraud** (checks patterns), **Capacity** (monitors portfolio limits).

Each reads current state and emits an *intent*: a proposed action or decision.

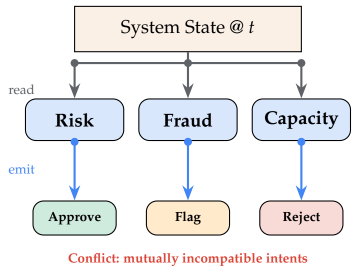

Figure 2: Agentic conflict: three components read identical state, then emit mutually incompatible intents. There is no address collision—conflict is semantic.

Risk recommends approval. Fraud flags for investigation. Capacity recommends rejection due to portfolio limits. **There is no address collision**—each component writes to its own intent stream. Yet the intents are incompatible: an application cannot be simultaneously approved and rejected.

4 Semantic MVCC

We define Semantic MVCC as concurrency control over decisions.

4.1 Definitions

System State (S_t): The complete observable state at time t, including all business objects, constraints, and derived values that any component may read.

Intent (I): A tuple (a, t, s, δ) where a identifies the component, t is the timestamp, s is a hash of the state snapshot observed, and δ is the proposed state change or decision.

Conflict: Two intents I_1 and I_2 are in conflict if both are individually permissible given S_t but their conjunction violates system constraints:

$$\text{ok}(I_1, S_t) \wedge \text{ok}(I_2, S_t) \wedge \neg\text{ok}(I_1 \wedge I_2, S_t)$$

Resolution Function (R): A deterministic function that maps a set of intents and the current state to a new state:

$$R : \mathcal{P}(I) \times S \rightarrow S'$$

4.2 Semantic ACID

We extend ACID guarantees to the agentic context:

Property	Semantic Interpretation
Atomicity	All intents in a commit window are resolved as a unit—either all are processed or none are, preventing partial state updates
Consistency	System invariants and business constraints hold after every commit, regardless of which intents won or lost
Isolation	Each component sees a stable, frozen snapshot of state while forming its intent—no mid-decision state changes
Durability	Both the committed state and the complete intent log survive system failures and restarts

Table 1: Semantic ACID guarantees for agentic systems.

4.3 The Commit Cycle

The system operates in discrete cycles, each producing a new state version:

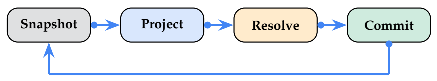

Figure 3: Commit cycle: components read a snapshot, project intents, the system resolves conflicts, commits new state, then repeats.

Snapshot: All components read identical state S_t.

Project: Components emit intents based on their logic.

Resolve: Resolution function R processes all intents.

Commit: New state S_{t+1} is derived and persisted.

5 Resolution Strategies

The resolution function R is domain-specific. Three canonical patterns cover most enterprise needs:

Timestamp Ordering. When intents conflict, the earliest timestamp wins. This is deterministic and simple but may leave value unrealized—a later, better intent loses simply because it arrived second. Best for: first-come-first-served scenarios, tiebreaking after other rules.

Allocation. When multiple intents compete for a constrained resource (capital, capacity, inventory), divide the resource proportionally or by priority weighting. No intent is fully rejected; each receives a share. Best for: resource budgeting, portfolio allocation, capacity management.

Quorum. When multiple components evaluate the same question (e.g., data extraction to prevent hallucination), require agreement above a threshold before committing. Disagreement triggers review rather than arbitrary selection. Best for: validation tasks, cross-checking, error detection.

6 Case Study: Insurance

Figure 4 shows commercial property underwriting. Four components evaluate application #1847 for a $5M property policy:

1. **Risk**: Actuarial analysis recommends approval at $48K annual premium for $5M coverage
2. **Fraud**: Pattern detection approves (fraud score 0.12 is below 0.25 threshold)
3. **Capacity**: Portfolio management flags that the carrier's net retention for hurricane-exposed coastal properties is at 78%—internal guidelines cap single-peril concentration at 80%, limiting new hurricane exposure
4. **Compliance**: Regulatory check approves

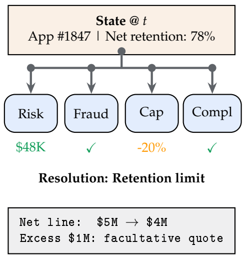
State @ t
App #1847 | Net retention: 78%
Risk
Fraud
Cap
Compl
$48K
✓
-20%
✓
Resolution: Retention limit
Net line: $5M → $4M
Excess $1M: facultative quote

Figure 4: Insurance underwriting: carrier retention limit constrains net line. Carrier writes $4M net, flags $1M for facultative reinsurance placement.

The capacity constraint limits the carrier's net retention. Resolution: the carrier writes $4M on its own book (within the 80% concentration guideline) and generates a facultative reinsurance quote request for the remaining $1M. The insured receives full $5M coverage; the carrier's exposure stays within risk appetite. Premium is allocated: $38.4K retained, with the facultative premium determined by reinsurer response.

Audit query: "Why was net retention limited to $4M?"

System response: "At commit time t, CapacityAgent observed hurricane net retention at 78% of guideline cap. Writing $5M net would exceed 80% threshold. Net line reduced to $4M per retention rule CAP-003; excess $1M flagged for facultative placement."

7 Case Study: Trading

Figure 5 shows an equity execution decision:

1. **Alpha**: Signal generation identifies momentum in NVDA, recommends accumulating 5,000 shares
2. **Risk**: Position risk calculates that adding 5,000 shares would consume $3.2M of VaR budget; only $2M remains in the desk's daily limit, supporting approximately 2,000 shares at current volatility
3. **Execution**: Order management recommends TWAP (time-weighted average price) over 30 minutes given current spread and volume
4. **Compliance**: Pre-trade check confirms no restricted list issues, position limits satisfied

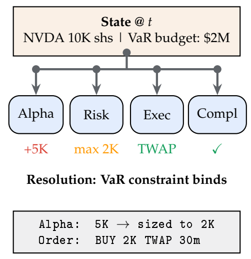

Figure 5: Trading desk: Alpha signal sized down to fit VaR budget. Signal remains valid; only quantity constrained. Execution strategy preserved.

The VaR constraint binds. Alpha's signal is not rejected—the directional view is preserved—but position size is scaled to fit the available risk budget. Standard practice: the signal informs direction and conviction; risk limits determine sizing. Execution proceeds with 2,000 shares via TWAP.

Regulatory query: "Why was the order 2,000 shares instead of 5,000?"

System response: "RiskAgent calculated 5,000 shares would require $3.2M VaR; desk limit had $2M remaining. Order sized to 2,000 shares per risk rule VAR-001. Signal validity preserved; sizing adjusted to constraint."

8 Implementation

The architecture is GPU-native throughout:

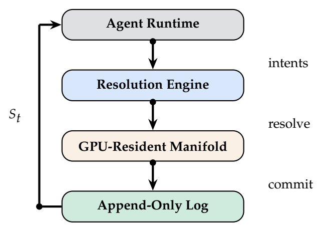

Figure 6: GPU-native stack: state lives in HBM, resolution executes as tensor operations, log persists to NVMe.

Component	Implementation
State Manifold	GPU HBM tensor
Intent Vectors	Embeddings in shared memory
Resolution Engine	CUDA kernels
Append-Only Log	NVMe with GPU-direct
Snapshot Isolation	Manifold versioning

Table 2: GPU-native implementation mapping.

8.1 Why GPU-Native

Traditional implementations serialize state to CPU memory, pay network round-trips to databases, and suffer context-switch overhead. GPU-native architecture eliminates these costs:

State as tensor. System state lives in HBM as a structured tensor. Components read via memory-mapped access—no serialization, no network hop.

Resolution as kernel. Conflict detection and resolution execute as CUDA kernels. Constraint checking is parallel predicate evaluation. Allocation is parallel reduction.

Log as GPU-direct. Committed intents write directly from GPU memory to NVMe storage, bypassing CPU entirely.

8.2 Log Entry Schema

Each intent is recorded with full context:

```
{
  "ts": "2025-12-20T14:32:01Z",
  "agent": "RiskAgent",
  "state_hash": "7f3a9b2c",
  "intent": {"action": "LIMIT",
             "max_qty": 2000},
  "resolution": {"rule": "VAR-001",
                 "outcome": "APPLIED"},
  "state_after": "9b2c4e1f"
}
```

8.3 Guarantees

The append-only log provides three critical properties:

Replayability. Any historical state can be exactly reconstructed by replaying the intent log from the beginning (or from a checkpoint) to the desired timestamp. This enables debugging production issues in test environments with perfect fidelity.

Auditability. Every decision is traceable to the state the component observed (via `state_hash`), the intent it emitted, and the rule that determined the outcome. No decision is opaque.

Debugging. When a production issue occurs, engineers can replay the exact sequence of intents against the exact state to reproduce the problem deterministically. Flaky tests become impossible when state is fully captured.

9 Design Methodology

Semantic MVCC transforms coordination from a runtime mystery into a design-time exercise:

Design Question	Produces
What components touch shared state?	Component scope and boundaries
What constraints span components?	Allocation rules for shared resources
Where do components decide the same question?	Decision function (vote, rank, or delegate)
Where do components validate each other?	Quorum threshold and escalation path

Table 3: Design-time checklist for Semantic MVCC.

Recommended development approach:

1. **Deploy with timestamp-only resolution.** Let the system run with the simplest possible conflict resolution: first intent wins.
2. **Log all conflicts.** Record every case where intents were incompatible, including what state each component saw and what each intended.
3. **Review patterns.** Analyze the conflict log to identify recurring conflict types and their business implications.
4. **Write explicit rules.** Codify resolution logic for each pattern discovered—allocation ratios, priority hierarchies, quorum thresholds.
5. **Deploy and repeat.** Rules may surface new conflict patterns. The system teaches you what it needs.

Conflict resolution is *discovered from data*, not designed from first principles. The conflict log is training data for your resolution function.

10 Related Work

Database MVCC. PostgreSQL, Oracle, and other databases implement MVCC for row-level concurrency [1]. Our work extends these semantics from data conflicts to decision conflicts, and moves execution from CPU to GPU.

Event Sourcing. The append-only log pattern derives from event sourcing [2]. We apply this pattern to multi-component intent capture rather than domain events.

Actor Model. Components resemble actors with message passing [3]. We replace asynchronous messages with synchronous shared state and explicit resolution functions.

Orchestration Frameworks. LangChain, CrewAI, and AWS Strands [4] provide agent orchestration but lack transactional state semantics—no snapshot isolation, no deterministic replay, no conflict resolution framework.

11 Limitations

Resolution function design. The framework provides structure but does not prescribe how to design resolution functions. Domain expertise remains essential for crafting rules that align with business objectives.

GPU availability. The architecture assumes GPU infrastructure. CPU-only deployments would sacrifice the performance benefits while retaining the semantic guarantees.

System boundaries. We assume a single system boundary with one authoritative state. Federated multi-system coordination would require distributed consensus protocols.

12 Conclusion

Semantic MVCC provides the missing foundation for enterprise agentic AI:

Isolation ensures each component sees a consistent, stable state snapshot while forming its intent—no race conditions, no torn reads. Without isolation, components reason against shifting ground.

Auditability means every decision is logged with full context: what state was observed, what intent was formed, what rule fired, what outcome resulted. Without auditability, regulators cannot trust the system.

Replayability allows any historical state to be reconstructed by replaying the intent log, enabling debugging and compliance review. Without replayability, "why did this happen?" has no answer.

Explainability lets the system answer regulatory "why" questions with specific references to state hashes, intent records, and resolution rules. Without explainability, the system is a black box.

The key insight is that agentic memory should serve as a reasoning surface, not a retrieval system. State, recall, inference, and update become unified operations over a shared GPU-resident substrate.

Enterprises cannot realize the transformative value of agentic AI without governance guarantees. Semantic MVCC provides them.

Data MVCC normalizes conflict away. Semantic MVCC accepts conflict and makes resolution explicit, auditable, and replayable.

Acknowledgments

The author thanks colleagues at NVIDIA and AWS for discussions that shaped this framework.

References

[1] P. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys*, vol. 13, no. 2, 1981.

[2] M. Fowler, "Event Sourcing," martinfowler.com, 2005.

[3] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," *IJCAI*, 1973.

[4] AWS, "Introducing Strands Agents," aws.amazon.com/blogs, 2025.