# **Reactive Composite Memory (RCM): A Pattern for Governed, Versioned Context in Intelligent Systems**

**Version:** 1.0  
**Date:** October 2025  
**Status:** Community Specification Draft  
**License:** CC-BY-4.0 (spec), Apache-2.0 (test code)  
**Authors:** Dan Vanderboom, Critical Insight Inc.  
**Contributing Organizations:** [TBD as ecosystem grows]

## **Abstract**

Reactive Composite Memory (RCM) is a design pattern for managing context in intelligent systems as a governed, reactive dataflow. Multiple sources compose into versioned artifacts (frames) that update when dependencies change, with built-in lineage, time semantics, and delivery guarantees. The pattern provides well-defined extension points where implementations inject cross-cutting behaviors (admission control, resource management, security) while preserving RCM's core guarantees: deterministic recomputation, idempotent delivery, and explainable provenance.

**Keywords:** reactive systems, composite memory, event-driven architecture, context management, AI systems, streaming semantics, provenance

## **Table of Contents**

### **1. Introduction**

1.1 Purpose and Scope  
1.2 The Core Problem  
1.3 Pattern Overview  
1.4 What RCM Is (and Is Not)  
1.5 Audience and Document Structure  
1.6 Conformance  
1.7 Normative Language and Conventions

### 

### **2. Problem and Design Forces**

2.1 Operational Challenges  
2.2 Why Ad-Hoc Approaches Fail  
2.3 Forces and Trade-Offs  
2.4 Success Criteria

### **3. RCM in One Picture**

3.1 Visual Model  
3.2 Core Concepts  
3.3 Illustrative Flow  
3.4 Key Implications

### **4. Normative Semantics**

4.1 Conformance  
4.2 Roles and Terms  
4.3 Portable Materialization Envelope  
4.4 Time, Ordering, and Reactive Recomputation  
4.5 Determinism, Idempotency, and Replay  
4.6 Delivery Contract  
4.7 Extension Points  
4.8 Summary of Requirements

### **5. Comparative Guide**

5.1 RCM vs. Adjacent Patterns  
5.2 Decision Heuristics  
5.3 Co-Existence Patterns  
5.4 Anti-Patterns and Common Confusions  
5.5 Edge Protocols and Pattern Fit

### **6. Conclusion**

6.1 What RCM Standardizes  
6.2 Conformance Summary  
6.3 Adoption Path  
6.4 Interoperability and Ecosystem  
6.5 How to Participate  
6.6 Acknowledgments

## 

## **Annexes**

### **Annex A (Normative): Conformance and Testing**

A.1 Test Harness Model  
A.2 Required Events and Metrics  
A.3 Measurement Rules and SLO Accounting  
A.4 SLO Obligations  
A.5 Required Test Vectors  
A.6 Conformance Report Format  
A.7 Producer and Subscriber Reference Checks  
A.8 Notes on Portability

### **Annex B (Informative): References and Crosswalks**

B.1 Normative References  
B.2 Informative References  
B.3 Crosswalk: RCM Terms to Adjacent Patterns  
B.4 Crosswalk: RCM vs. AI Frameworks

### **Annex C (Informative): Glossary**

Alphabetical definitions of key terms with authoritative source references

### **Annex D (Informative): Extension Patterns**

D.1 Common Extension Point Patterns  
D.2 Admission Control Patterns  
D.3 Resource Management Patterns  
D.4 Security and Privacy Patterns  
D.5 Observability Patterns  
D.6 Example: Multi-Stage Governance Chain  
D.7 Example: Simple Rate Limiting  
D.8 Example: Policy-as-Code Integration

## 

## **Appendices (Non-Normative)**

### **Appendix I: Implementation Guidance**

Brief pointers to separate implementation playbooks

### **Appendix II: Cognitive Analogies**

Summary of memory science parallels and design heuristics

### **Appendix III: Version History and Change Log**

v1.0 Initial Community Specification Release

## **About This Specification**

**Status:** Community Specification Draft

**License:** Creative Commons Attribution 4.0 (CC-BY-4.0) for specification text; Apache License 2.0 for code examples and test suite.

**Participation:** This specification is developed in the open. Contributions, implementations, and feedback are welcome.

**Repository:** github.com/critical-insight/rcm-spec  
**Conformance Test Kit:** github.com/critical-insight/rcm-conformance  
**Discussion Forum:** GitHub Discussions  
**Maintainers:** Dan Vanderboom (Critical Insight Inc.), [Co-maintainers TBD]

**Known Implementations:**

* TBD

**Acknowledgments:** This work draws on distributed systems research in stream processing (Dataflow Model), reactive architectures (Reactive Streams), event-driven patterns (Event Sourcing, CQRS), and cognitive science (William James, Atkinson-Shiffrin, Baddeley, Tulving). We thank the Confluent, Anthropic (MCP), and broader streaming/AI communities for ongoing collaboration.

## 

## **How to Read This Specification**

* **For Evaluators/Architects:** Read Chapters 1-3, Chapter 5 (Comparative Guide)
* **For Implementers:** Focus on Chapter 4 (Normative Semantics) and Annex A (Conformance)
* **For Users/Consumers:** Read Chapters 1-3; skim Chapter 5 for context
* **For Standards Bodies:** All chapters; Annexes A-C provide formal apparatus

**Total Estimated Length:** 177 pages

**Target Formats:**

* PDF (typeset, for archival and citation)
* HTML (web-readable, linkable sections)
* Markdown (GitHub, for community collaboration)

# **1. Introduction**

## **1.1 Purpose and Scope**

This specification defines **Reactive Composite Memory (RCM)**, a design pattern for managing context in intelligent systems—particularly AI agents, real-time operations platforms, and event-driven architectures—as a governed, reactive dataflow.

Modern systems must **remember across sessions**, **justify outcomes**, and **control cost and risk**. Ad-hoc caches and per-request "context assembly" are brittle: they lack provenance, consistent update semantics, and reliable cost attribution. RCM elevates these concerns from afterthoughts to first-class design constraints.

**Scope of this specification:**

* **Normative semantics** for reactive composition, versioned frames, time/ordering, determinism, and delivery (Chapter 4)
* **Conformance requirements** and test procedures (Annex A)
* **Comparative positioning** against adjacent patterns (Chapter 5)
* **Extension points** where implementations inject cross-cutting behaviors (§4.7, Annex D)

This specification is **platform-agnostic** and **vendor-neutral**. It defines *what* conformance means, not *how* to implement it. Implementation strategies are covered in separate playbooks.

## **1.2 The Core Problem**

Intelligent systems—AI agents, real-time dashboards, incident response platforms, personalization engines—face a common challenge: **maintaining fresh, explainable, composite context** drawn from many changing sources.

**The traditional approaches:**

* **Per-request assembly:** Every inference or query rebuilds context from scratch—expensive, high-latency, no lineage
* **Static caches:** Fast but stale; no reactive updates; opaque provenance
* **Batch ETL:** Handles historical data well but lacks real-time semantics and live consumer delivery

**What breaks at scale:**

1. **Freshness:** Context falls behind reality; decisions use stale data
2. **Explainability:** "Why did the system know X?" has no auditable answer
3. **Cost:** Redundant recomputation; every consumer pays the assembly tax
4. **Fairness:** High-priority work starves; no capacity control under load
5. **Privacy:** Retention and redaction applied inconsistently per pipeline
6. **Velocity:** Each new "memory feature" requires custom glue code

**What's needed:** A pattern that treats context as a **living, versioned, governed artifact** that updates reactively, carries its lineage, and can be reused by many consumers—agents, services, dashboards, and analytics—without per-consumer duplication.

## **1.3 Pattern Overview**

RCM treats memory not as a static store or per-request cache, but as a **reactive composition** of sources into contexts (views) that materialize as **versioned, immutable frames** whenever dependencies change.

**Core flow:**

Source Signals → Context Views → Materializer → Frames → Subscriptions → Consumers

↓ ↓ ↓ ↓ ↓ ↓

(changes) (declarative) (windowing) (versioned) (ordered) (idempotent)

(deterministic) (time-aware) (lineage) (delivery) (effects)

**Key concepts:**

* **Source Signals:** Typed, timestamped change feeds (events, CDC, file arrivals, API updates) with monotone positions
* **Context Views:** Declarative compositions over sources and other views—joins, filters, windows, summaries—expressed as deterministic plans
* **Materializer:** The executor that applies time semantics (event-time, watermarks, window closure) and evaluates plans to produce frames
* **Frames:** Immutable materializations (snapshots or deltas) carrying headers for identity, version, lineage (input ranges + plan hash), time window, TTL, and extension metadata
* **Subscriptions:** At-least-once, per-key–ordered delivery to many heterogeneous consumers
* **Consumers:** Agents, services, UIs, analytics pipelines that process frames idempotently (effect-once despite duplicate deliveries)

**What makes this reactive:**

When a source changes, affected views recompute and emit new frames automatically. Consumers subscribe to frame streams rather than polling or rebuilding context. The effect is **memory that stays fresh**, **operations that are explainable** (every frame cites its sources), and **costs that are predictable** (recomputation happens on change, not on traffic).

**Extension points:**

RCM provides well-defined hooks where implementations inject cross-cutting behaviors—admission control (policy checks, rate limits), resource management (budgets, quotas), security posture (classification, redaction), and human-in-the-loop gates. The pattern specifies *where* these extensions attach and *what telemetry* they must emit; the specific policies are implementation-defined. (See §4.7 and Annex D.)

## **1.4 What RCM Is (and Is Not)**

**RCM is:**

* ✅ A **design pattern** for reactive, versioned, governed context management
* ✅ A **specification** with normative semantics and conformance tests
* ✅ **Platform-agnostic**: implementable on Kafka, Pulsar, Flink, custom runtimes, cloud services
* ✅ **Interoperable**: plays well with Event Sourcing, CQRS, search indices, knowledge graphs, edge protocols (MCP, A2A)

**RCM is not:**

* ❌ A **product** or **framework** (though products may implement it)
* ❌ A **database** or **cache** (though it leverages both)
* ❌ A **replacement** for Event Sourcing, CQRS, or streaming engines (it complements them)
* ❌ A **UI architecture** (though it borrows unidirectional flow from Elm/Redux and extends it to durable, multi-consumer memory)

**Think of RCM as:**

* REST is to web APIs
* Reactive Streams is to backpressure
* CQRS is to read/write separation

RCM is to **context lifecycle**: it defines how composite knowledge forms, versions, persists, and flows—with time, lineage, and governance as first-class concerns.

## **1.5 Audience and Document Structure**

**Primary audiences:**

* **Architects and decision-makers** evaluating whether RCM fits their systems (read Chapters 1-3, 5)
* **Implementers** building RCM-conformant systems (focus on Chapter 4, Annex A)
* **Operators and SREs** running RCM systems in production (read Chapters 1-3; skim §4.7 and Annex D for extension patterns)
* **Standards bodies and researchers** assessing the pattern's rigor and portability (all chapters; Annexes A-C)

**Document structure:**

* **Chapter 2** clarifies the problem and design forces RCM balances
* **Chapter 3** gives a visual model and illustrative flow in plain language
* **Chapter 4** specifies normative semantics (envelope, time/ordering, determinism, delivery, extension points)
* **Chapter 5** positions RCM against adjacent patterns (Event Sourcing, CQRS, caching, etc.) with decision heuristics
* **Chapter 6** concludes with conformance summary and participation guidance
* **Annex A (Normative)** defines conformance tests, required telemetry, and SLO measurement
* **Annex B (Informative)** provides references and crosswalks to related work
* **Annex C (Informative)** is a glossary of terms with authoritative citations
* **Annex D (Informative)** documents common extension patterns (governance chains, rate limiting, policy-as-code)

## **1.6 Conformance**

RCM defines **one conformance class** covering reactive composition, versioned frames, time semantics, deterministic materialization, and idempotent delivery.

**To claim RCM conformance, an implementation MUST:**

1. Satisfy normative requirements in §§4.2-4.6 (roles, envelope, time/ordering, determinism, delivery)
2. Pass test vectors defined in Annex A.5
3. Emit required telemetry events and metrics (Annex A.2)
4. Produce a conformance report in the specified format (Annex A.6)

**Extension points** (§4.7) allow implementations to inject admission control, resource management, security, and observability behaviors. These extensions are **implementation-defined**; RCM specifies the *hooks* and *telemetry semantics*, not the *policies*. Common patterns are documented in Annex D (Informative).

**Why one class:**

Earlier drafts included "RCM-Core" and "RCM-Governed" classes. We consolidated to a single class because **governance is a cross-cutting concern** that implementations address via extension points, not a fundamental variant of the pattern. This simplifies adoption: teams can implement the core reactive semantics first and layer governance as operational needs dictate.

## **1.7 Normative Language and Conventions**

This specification uses normative keywords as defined in **RFC 2119** and **RFC 8174**:

* **MUST / REQUIRED / SHALL:** Absolute requirement for conformance
* **MUST NOT / SHALL NOT:** Absolute prohibition
* **SHOULD / RECOMMENDED:** Strong recommendation; may be valid reasons to ignore in particular circumstances, but implications must be understood
* **SHOULD NOT / NOT RECOMMENDED:** Strong discouragement; valid reasons may exist
* **MAY / OPTIONAL:** Truly optional; implementers may choose to include or omit

Only the **uppercase forms** carry the special RFC meanings. Lowercase usage (e.g., "implementations should consider...") is ordinary English.

**Document conventions:**

* **Normative sections** impose requirements for conformance (Chapters 4, 6; Annex A)
* **Informative sections** provide context, rationale, and examples but do not impose requirements (Chapters 1-3, 5; Annexes B-D; Appendices)
* **Illustrative pseudocode and diagrams** clarify intent but are not themselves normative unless explicitly stated
* **Telemetry names** (events, metrics) in Annex A are normative for semantics; exact spelling MAY vary if meaning is preserved and documented

**Versioning:**

This is version **1.0** of the RCM specification. Future versions will be numbered according to semantic versioning principles:

* **Major version** increments indicate breaking changes to conformance requirements
* **Minor version** increments add new optional features or clarifications
* **Patch version** increments fix errors or ambiguities without semantic change

**Changes are tracked in Appendix III (Version History).**

# **2. Problem and Design Forces**

## **2.1 Operational Challenges**

RCM addresses a constellation of operational pains that surface when systems scale beyond ad-hoc context assembly. Each challenge is real; collectively, they demand a coherent pattern rather than point solutions.

### **2.1.1 Freshness at Scale**

**The problem:** Composite context—joining user profiles, recent activity, policy updates, and external signals—must stay current as upstream sources change. Poll-based approaches create storms; push-based glue code proliferates.

**What breaks:**

* Agent decisions use stale entitlements or outdated threat intelligence
* Dashboards show "current state" that lags reality by minutes or hours
* Per-consumer polling creates N × M traffic (N consumers × M sources)
* Custom "watch and invalidate" logic is fragile and hard to audit

**What's needed:** Declarative reactive recomputation that propagates changes to affected contexts automatically, with bounded staleness and observable lag.

### **2.1.2 Explainability and Provenance**

**The problem:** Systems must answer "what did we know, why did we know it, and since when?" for compliance, debugging, and trust—but ad-hoc assembly leaves no trail.

**What breaks:**

* Post-incident reviews can't replay the context that fed a decision
* Compliance audits find no lineage connecting outputs to inputs
* Trust erodes when systems can't justify recommendations or actions
* Debugging requires reconstructing history from logs and intuition

**What's needed:** Every materialization must carry machine-readable lineage (source ranges, transform identity, time window) so decisions are auditable by construction.

### **2.1.3 Cost and Resource Control**

**The problem:** Context assembly burns compute, network, and storage—but without instrumentation, teams can't predict or bound spend. Traffic spikes become cost surprises.

**What breaks:**

* LLM context payloads grow unbounded, multiplying inference costs
* Redundant joins and aggregations repeat across consumers
* No clear attribution: which feature or tenant drove the spend?
* Burst traffic causes cascading overload with no graceful degradation

**What's needed:** Shift cost from per-request (traffic-driven) to per-change (data-driven), with visible budgets, leases, and capacity controls that allow predictable spend under load.

### **2.1.4 Consistency Under Load**

**The problem:** When many consumers need updates simultaneously, systems experience stampedes, priority inversion, and non-deterministic failures—no fair arbitration exists.

**What breaks:**

* Cache invalidations trigger thundering herds
* High-priority work (incident response) waits behind bulk analytics
* Retries compound load rather than shedding it
* No coordination: each team builds custom throttling

**What's needed:** Explicit concurrency control, fair scheduling with priority and aging, and queue disciplines that degrade gracefully rather than collapse.

### **2.1.5 Privacy and Policy Enforcement**

**The problem:** Retention, redaction, access control, and consent apply inconsistently across pipelines. Each team interprets policies differently; audits find gaps.

**What breaks:**

* PII lingers beyond stated retention in orphaned caches
* Redaction happens (or doesn't) based on which service path was taken
* "Right to erasure" requests miss derivative contexts
* Cross-border data movement violates policy silently

**What's needed:** Uniform policy attachment to every artifact—classification, redaction paths, TTL—enforced at materialization and egress, with telemetry proving compliance.

### **2.1.6 Velocity Without Entropy**

**The problem:** Shipping new "memory features"—additional joins, summaries, or projections—requires bespoke pipelines, deployment risk, and maintenance burden. Systems calcify.

**What breaks:**

* Each new context requires custom code, testing, and monitoring
* No reuse: similar logic duplicates across teams
* Rollback is hard; no safe dual-run or replay for validation
* Fear of breaking existing consumers slows iteration

**What's needed:** Declarative composition where new views are added, tested via replay, and promoted safely—with the platform handling timing, delivery, and lineage automatically.

## **2.2 Why Ad-Hoc Approaches Fail**

### **2.2.1 Per-Request Assembly (The LLM Anti-Pattern)**

**What it looks like:**

On inference request:

1. Fetch user profile from DB

2. Query last N messages from store

3. Retrieve relevant docs via vector search

4. Call external API for real-time signal

5. Assemble prompt

6. Send to LLM

**Why it fails:**

* **No reuse:** Every inference re-fetches and re-joins, even if nothing changed
* **No lineage:** Can't answer "what context fed this response?"
* **Cost scales with traffic, not change:** 10× requests = 10× assembly work
* **Latency compounds:** Critical path includes all fetch latencies
* **Non-deterministic:** Concurrent updates create race conditions; replays differ

### **2.2.2 Static Caches**

**What it looks like:**

Cache key: user\_123\_context

Value: {profile, lastActivity, preferences}

TTL: 5 minutes

**Why it fails:**

* **Reactive gap:** Changes upstream don't invalidate automatically; stale windows are inevitable
* **Provenance void:** Cache blobs carry no lineage; "where did this come from?" has no answer
* **Invalidation storms:** Expiration triggers N simultaneous rebuilds (thundering herd)
* **No versioning:** Consumers can't pin to a stable view or replay history

### **2.2.3 Batch ETL Pipelines**

**What it looks like:**

Every hour:

1. Extract from sources

2. Transform (joins, aggregations)

3. Load to warehouse or data lake

**Why it fails:**

* **Latency floor:** Freshness measured in hours or days, not seconds
* **No live consumers:** Agents and dashboards can't subscribe to incremental updates
* **All-or-nothing:** Can't selectively recompute one key without full batch
* **Delivery semantics unclear:** Consumers poll and diff; no at-least-once guarantee

## **2.3 Forces and Trade-Offs**

RCM must balance competing forces. The pattern doesn't eliminate trade-offs; it makes them **explicit and tunable**.

### **2.3.1 Reactivity ↔ Stability**

**Reactivity:** Recompute and emit frames immediately when sources change—minimizes staleness  
 **Stability:** Coalesce changes within small windows to avoid version churn and consumer thrash

**RCM's answer:**

* Configurable coalescing windows (e.g., tens of milliseconds) per view
* Emit snapshots periodically; emit deltas for interim corrections
* Brownout mode: widen coalescing under load to preserve SLOs

**Knob:** Coalescing duration and snapshot cadence (view-specific policy)

### **2.3.2 Consistency ↔ Availability**

**Consistency:** Wait for all inputs and perfect ordering before emitting—guarantees correctness  
 **Availability:** Emit with best-effort inputs to reduce latency—tolerates disorder

**RCM's answer:**

* Use watermarks to bound staleness explicitly (no earlier events expected)
* Define late-arrival policies per view (accept within Δ, emit corrective deltas, or ignore)
* Per-key ordering guarantees at delivery preserve causal consistency where it matters

**Knob:** Watermark advancement strategy and late-data acceptance window

### **2.3.3 Recall ↔ Privacy**

**Recall:** Retain frames indefinitely for forensics, learning, and compliance  
 **Privacy:** Minimize retention to reduce exposure and satisfy "right to erasure"

**RCM's answer:**

* Default to **short, non-zero TTLs** (auditable by default without indefinite retention)
* Explicit opt-in for longer retention (with classification and justification)
* Explicit opt-in for ephemeral (TTL=0) when policy demands it
* Redaction applies before egress; redacted fields don't propagate

**Knob:** TTL policy per view and classification; redaction rules in security posture

### **2.3.4 Centralization ↔ Federation**

**Centralization:** One memory plane for all teams—simplifies reuse and governance  
 **Federation:** Domain-specific planes with selective sharing—respects boundaries and autonomy

**RCM's answer:**

* Support both: run RCM per domain; federate via gateways that enforce policy and sign frames
* Frames carry provenance across trust boundaries; consuming domains can verify integrity
* Views can compose over local and remote sources with different retention/redaction rules

**Knob:** Deployment topology (monolithic vs. federated); gateway policies at edges

### 

### **2.3.5 Granularity ↔ Cost**

**Granularity:** Fine-grained frames (per-user, per-asset) improve traceability and reuse  
 **Cost:** Coarser frames (per-tenant, per-aggregate) reduce materialization and storage spend

**RCM's answer:**

* Choose keys that reflect natural ownership and consumption patterns
* Use hierarchical views: fine-grained local rolls; coarse aggregates atop them
* Emit deltas for high-rate fine-grained updates; snapshot periodically to cap replay cost

**Knob:** Key selection (user vs. tenant vs. global); snapshot-to-delta ratio

## **2.4 Success Criteria**

RCM succeeds when it transforms context management from ad-hoc chaos into a **disciplined, observable, cost-effective** capability. Concretely:

### **2.4.1 Freshness Becomes Measurable**

**Before RCM:**

* "Is our context fresh?" has no answer
* Staleness discovered post-incident ("we used yesterday's data")

**After RCM:**

* Freshness = watermark\_lag\_ms per context, tracked and alerted
* SLOs like "p95 lag < 2 seconds" are testable and enforceable
* Brownout mode activates before SLO breach, not after collapse

### **2.4.2 Decisions Are Auditable**

**Before RCM:**

* "Why did the agent recommend X?" requires detective work
* Compliance teams demand lineage; engineers shrug

**After RCM:**

* Every frame header cites inputs (source ranges + hashes), plan identity, and time window
* Replay by version or interval is routine, not heroic
* Auditors query frame store: "show me what the system knew at time T for user U"

### **2.4.3 Cost Is Visible and Controllable**

**Before RCM:**

* Spend correlates with traffic, not change; surprises are common
* No attribution: "why did last month cost 30% more?"

**After RCM:**

* Recomputation happens on change; frame reuse across consumers amortizes cost
* Budget leases and commit telemetry show spend per context/view/tenant
* Capacity controls (concurrency limits, quotas) prevent runaway spend

### **2.4.4 Composition Is Reusable**

**Before RCM:**

* Each team builds similar joins and summaries independently
* Drift accumulates; no shared truth

**After RCM:**

* One view definition serves agents, dashboards, analytics, and indices
* New consumers subscribe to existing frames—no duplicate pipelines
* New views compose over existing views—hierarchical reuse

### **2.4.5 Policies Apply Uniformly**

**Before RCM:**

* Retention and redaction are per-pipeline band-aids
* "Right to erasure" requires manual hunting

**After RCM:**

* Classification, redaction, and TTL attach to frames at materialization
* Edge gateways enforce policy before egress
* Telemetry proves compliance (redaction.applied.count, policy.denied.count)

### **2.4.6 Iteration Is Safe**

**Before RCM:**

* Changing a join risks breaking consumers; rollback is manual
* Testing requires full prod replay

**After RCM:**

* Dual-run new plans with planHash diffs; validate before cutover
* Golden replay vectors in CI catch regressions
* Versioned frames allow gradual consumer migration

## **2.5 Summary: The Pattern's Promise**

RCM doesn't eliminate complexity—it **organizes** it. The pattern acknowledges that:

* Context management is **hard** (multiple sources, time semantics, load, privacy)
* Ad-hoc solutions **fragment** (each team solves it differently, poorly)
* The forces are **real** (reactivity vs. stability, recall vs. privacy, cost vs. granularity)

**RCM's contribution:**

A coherent design that makes these forces **explicit**, trade-offs **tunable**, and outcomes **observable**—so teams can build intelligent systems that remember reliably, explain credibly, and scale economically.

**Next:** Chapter 3 translates these requirements into a visual model and illustrative flow.

# **3. RCM in One Picture**

## **3.1 Visual Model**

The following diagram captures RCM's structure and flow at a glance. Read left to right: changes flow from sources through composition and materialization into versioned frames, which are delivered to many consumers. Extension points (dotted lines) show where implementations inject cross-cutting behaviors.

flowchart LR

subgraph Sources

E[Event Streams]

D[Databases / CDC]

G[Graphs / Indices]

X[External APIs]

end

Sources -->|changes| V[Context Views<br/>declarative composition]

V --> M[Materializer<br/>windowing, time semantics]

M --> F[(Frames<br/>versioned, immutable)]

F --> B[Subscriptions Bus<br/>ordered delivery]

B --> C1[Agents]

B --> C2[Services]

B --> C3[Dashboards/UI]

B --> C4[Analytics]

subgraph Extensions

P[Policy / Access]

R[Resource Mgmt]

S[Security / Privacy]

O[Observability]

end

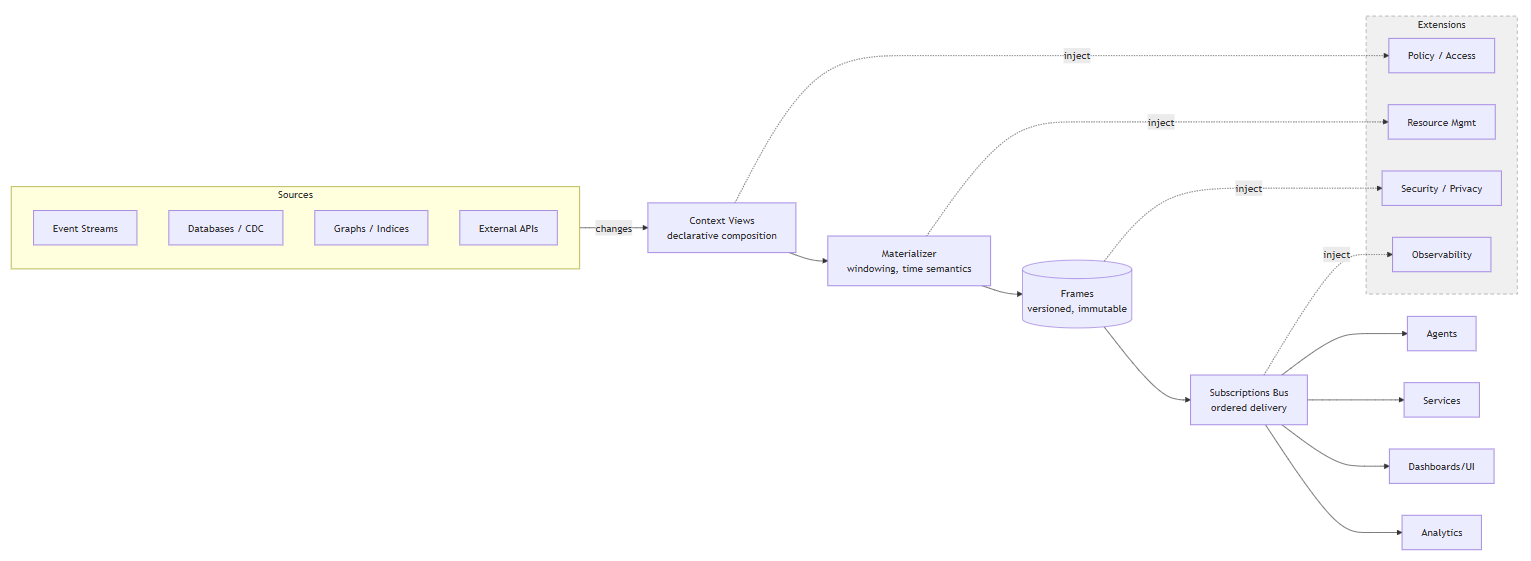
V -.->|inject| P

M -.->|inject| R

F -.->|inject| S

B -.->|inject| O

style Extensions fill:#f0f0f0,stroke:#999,stroke-dasharray: 5 5



**Key observations:**

Unidirectional flow: Changes propagate forward through the pipeline. Consumers never mutate frames; new knowledge generates new frames.

Separation of concerns: Views define *what* to compute (declarative). The Materializer handles *when* to compute (time semantics). Frames are *immutable artifacts*. Subscriptions handle *delivery*. Extensions inject *policies*.

One memory, many consumers: The same frames serve agents, operational services, dashboards, and analytics—no parallel caches to drift.

## **3.2 Core Concepts**

### **Source Signals**

Typed, timestamped change feeds that drive reactive recomputation. Examples include event streams (clicks, transactions), change-data-capture from databases, document arrivals, embedding updates, and external API webhooks.

Each source provides a monotone position (offset, sequence number, timestamp) per partition or key, enabling deterministic replay.

### 

### **Context Views**

Declarative compositions over sources and other views. A view specifies *what* to compute—joins, filters, windows, aggregations, summaries—as a deterministic plan. Views hold no mutable state; they are recipes.

Example views:

* user\_activity: windowed stream of last N events per user
* user\_profile: reference data slice (demographics, preferences)
* support\_context: join of user\_activity, user\_profile, and kb\_fixes

Views compose hierarchically: complex contexts build on simpler ones.

### **Materializer**

The executor that applies time semantics and evaluates view plans to produce frames. The Materializer:

* Tracks event-time and advances watermarks per context and key
* Closes windows when watermarks pass boundaries
* Fetches inputs (from sources or other frames)
* Evaluates the deterministic plan
* Emits immutable frames with complete envelopes

The Materializer is the system's timing and correctness hinge. It enforces "no frame emitted before window closure" and handles late/duplicate data per policy.

### **Frames**

Immutable materializations of a view at a point or interval in event-time. Each frame carries a header with:

* Identity: frameId (globally unique), contextId (view identity), key (partition), version (monotone per context+key)
* Type: snapshot (whole state) or delta (incremental change)
* Time: window (event-time interval), watermarkAt (staleness bound)
* Provenance: inputs (source ranges + hashes), planHash (transform identity)
* Determinism: idempotencyKey (stable deduplication key)
* Lifecycle: ttl (retention policy), createdAt (wall-clock emit time)
* Extensions: extensions object for implementation-defined metadata (classification, redactions, budgets, approvals)

Frames are the durable substrate of memory. Consumers reference frames by ID; operators replay by version or time interval.

### **Subscriptions**

Delivery relationships where consumers receive frames with at-least-once, per-key ordering guarantees. The subscriptions bus:

* Maintains per-key order by (window.end, version)
* Retries failed deliveries with backoff
* Routes terminal failures to dead-letter queues
* Supports filtering (consumers bind to specific contexts and key patterns)
* Decouples producers from consumers (independent scaling, fan-out)

Consumers must process frames idempotently using idempotencyKey for deduplication.

### **Extension Points**

Well-defined hooks where implementations inject cross-cutting behaviors. RCM specifies *where* these hooks attach and *what telemetry* they emit; the specific policies are implementation-defined.

Common extension points:

* **Before materialization:** Admission control (policy checks, rate gates, concurrency limits, priority scheduling)
* **During materialization:** Resource management (budget leases for compute/memory/network/tokens)
* **Before publish:** Security posture (classification, redaction, approvals)
* **At delivery:** Observability (lineage tracking, cost attribution, health metrics)

Extensions are orthogonal to the core pattern. Implementations may provide simple extensions (basic rate limiting) or sophisticated ones (multi-stage governance with fairness and budgets). Annex D documents common patterns.

## 

## **3.3 Illustrative Flow**

This section walks through RCM's behavior in steady state: a source changes, a view recomputes, a frame emits, and consumers receive it.

### **Step 1: Source Change Detected**

An event arrives in a source stream (e.g., user clicks a button, a support ticket updates, a document is ingested). The source assigns a monotone position (offset 42) and timestamp (event-time T).

The Materializer observes the change and identifies affected views. For a view support\_context keyed by (tenantId, userId), the change maps to key (acme, alice).

### **Step 2: Watermark Advancement**

The Materializer advances the watermark for (support\_context, acme/alice) based on the new event's timestamp. If the watermark now exceeds the end of an open window, the window transitions from Accepting to Closing.

Late-arrival policy determines whether to accept events that arrive after the watermark. If accepted, a corrective delta frame will emit with incremented version.

### **Step 3: View Evaluation (Deterministic Plan)**

The Materializer fetches inputs for the window:

* From source streams: event ranges [offset\_start, offset\_end]
* From other frames: specific versions of prerequisite views
* From reference data: snapshots at the watermark time

It retrieves the view's plan (compiled, with operator versions and configuration baked in) and computes planHash = hash(plan).

The plan evaluates deterministically: given the same inputs, window, and parameters, it produces the same frame body and idempotencyKey.

### 

### **Step 4: Frame Construction**

The Materializer constructs the frame envelope:

* frameId: new ULID (globally unique)
* contextId: support\_context
* key: acme/alice
* version: previous version + 1 (e.g., 47)
* type: snapshot (if periodic) or delta (if corrective/incremental)
* window: {start: T1, end: T2}
* watermarkAt: current watermark
* inputs: [{sourceId: "tickets", from: "37", to: "42", hash: "abc123"}, ...]
* planHash: sha256(plan)
* idempotencyKey: sha256(contextId, key, window, inputs\_summary, planHash)
* ttl: from view policy (e.g., PT5M for 5 minutes)
* createdAt: wall-clock now
* extensions: implementation-defined (e.g., {classification: "internal", budgetUsed: {timeMs: 120}})

The frame body contains the computed payload (user profile + recent tickets + relevant KB articles).

### **Step 5: Extension Point: Before Publish**

Before emitting the frame, the implementation invokes extension hooks (if configured):

* **Policy check:** Does the requester have access? Is the classification allowed?
* **Rate gate:** Has this tenant exceeded their emission quota?
* **Budget lease:** Reserve 120ms of compute time; commit after publish succeeds
* **Redaction:** Apply redaction paths from security policy (e.g., mask PII fields)
* **Approval (optional):** For high-risk contexts, request human approval before exposing

Each hook emits telemetry (e.g., admission.decision, budget.commit, redaction.applied). If any hook denies, the frame does not publish; the decision is logged and may trigger alerting.

### 

### **Step 6: Publish to Frame Store**

The frame (headers + body) writes to the immutable frame store. The store indexes by (contextId, key, version) and by frameId. TTL metadata schedules eventual deletion.

If an identical idempotencyKey already exists (duplicate work detected), the store deduplicates: no new artifact created, but telemetry increments a duplicate.materialization counter.

### **Step 7: Notify Subscriptions**

The Materializer publishes a lightweight notification to the subscriptions bus: (contextId=support\_context, key=acme/alice, frameId=...). The bus does not carry the full frame body; subscribers fetch by ID if needed, or the bus delivers headers + pointers.

### **Step 8: Delivery to Consumers**

Subscribers bound to support\_context and matching key patterns receive the frame in per-key order (by window.end, then version). Delivery is at-least-once; consumers may see duplicates during retries.

Consumer pseudocode:

onFrame(frame):

if seenBefore(frame.idempotencyKey):

ack(); return // idempotent deduplication

effect = processFrame(frame.body)

commitEffect(effect)

recordSeen(frame.idempotencyKey)

ack()

If processing fails, the bus retries with exponential backoff. After exhausting retries, the frame routes to a dead-letter queue with enough context (frame headers, failure reason) to remediate and re-drive.

### 

### **Step 9: Observability and Settlement**

Throughout the flow, telemetry emits:

* frame.materialized (context, key, version, window, watermark, planHash, ttl)
* view.window.closed (context, key, window)
* budget.lease, budget.commit (units: timeMs, moneyCents, tokens, etc.)
* delivery.attempt, delivery.retry, delivery.dlq
* redaction.applied (if security extension active)

Operators monitor dashboards showing:

* Freshness: watermark\_lag\_ms per context (p95, p99)
* Latency: recompute\_latency\_ms, delivery\_latency\_ms
* Cost: budget.timeMs\_committed per tenant/view
* Health: delivery.retry.count, dlq.count

SLOs like "p95 freshness < 2s" are continuously evaluated. Brownout mode activates if backlog or lag exceeds thresholds, widening coalescing windows and emitting coarse frames to preserve SLOs.

## **3.4 Key Implications**

### **Unidirectional Flow**

Events flow forward: source changes trigger view recomputation, which emits frames, which notify subscribers. There is no backward mutation. Corrections emit new frames with incremented versions.

This mirrors Elm/MVU/Redux for UI state but extends it to durable, multi-session, multi-consumer memory.

### **Time as First-Class**

Event-time (when things happened) is distinct from processing-time (when the system sees them). Watermarks and windows make staleness explicit and controllable. Late data is handled deterministically per policy, not ad hoc.

### 

### **Determinism Where It Counts**

Given identical inputs, window, and plan, the Materializer produces the same frame body and idempotencyKey. This enables:

* Replay for audit and debugging
* Golden test vectors in CI
* Safe dual-run of plan changes (compare outputs before cutover)

Non-determinism (randomness, wall-clock reads) is discouraged in plans. If unavoidable, seed with event-time or include in planHash.

### **Separation of Concerns**

**Views** define *what* to compute (logic).  
 **Materializer** defines *when* to compute (timing).  
 **Frames** are *immutable artifacts* (truth).  
 **Subscriptions** handle *delivery* (fan-out, retries).  
 **Extensions** inject *policy* (governance, security, observability).

This separation allows independent evolution: change the delivery mechanism without touching view logic; add a new extension (e.g., budget tracking) without rewriting the Materializer.

### **Reuse Across Consumers**

One set of frames serves agents, operational services, dashboards, CMS, and analytics pipelines. No parallel caches. No drift. New consumers subscribe to existing frames without triggering recomputation.

This amortizes cost: recomputation happens on change (proportional to data velocity) not on traffic (proportional to consumer requests).

### **Governance via Extension Points**

The core pattern (reactive composition, versioned frames, idempotent delivery) is lean and portable. Cross-cutting concerns (policy, budgets, security) attach at well-defined hooks.

Implementations range from simple (basic rate limiter) to sophisticated (multi-stage governance with fairness, budgets, and human approvals). The pattern doesn't mandate; it accommodates.

### 

### **Explainability by Construction**

Every frame header answers:

* **What sources?** inputs field lists ranges and hashes
* **What transform?** planHash identifies the computation
* **What time?** window bounds event-time; watermarkAt bounds staleness
* **What version?** Monotone version enables replay

Auditors and operators can ask "what did the system know at time T for key K?" and get a definitive, replayable answer.

## **3.5 From Picture to Specification**

This chapter provided intuition: the visual model, the flow, and the implications. Chapter 4 makes it normative: testable requirements for conformance.

If Chapter 3 is the "what RCM looks like," Chapter 4 is "what RCM guarantees."

# **4. Normative Semantics**

This chapter specifies the testable, vendor-neutral semantics of Reactive Composite Memory (RCM). It uses RFC-2119 keywords (MUST, SHOULD, MAY) to define conformance requirements. Implementations claiming RCM conformance MUST satisfy these requirements and pass the test procedures in Annex A.

## **4.1 Conformance**

RCM defines **one conformance class** covering reactive composition, versioned frames, time semantics, deterministic materialization, and idempotent delivery.

**To claim RCM conformance, an implementation MUST:**

1. Satisfy all normative requirements in sections 4.2 through 4.6
2. Pass the test vectors specified in Annex A.5
3. Emit the required telemetry events and metrics defined in Annex A.2
4. Produce a conformance report in the format specified in Annex A.6

**Extension points** (section 4.7) provide hooks where implementations inject cross-cutting behaviors such as admission control, resource management, and security. These extensions are **implementation-defined**; RCM specifies the hooks and required telemetry semantics but not the specific policies. Common extension patterns are documented in Annex D (Informative).

**Verification:** Conformance is demonstrated by executing the test harness defined in Annex A.1 against the implementation and producing a report that documents test results, SLO attainment, and telemetry mappings.

**Scope of requirements:** These requirements apply to the runtime behavior of systems claiming RCM conformance. They do not mandate specific implementation technologies, programming languages, or deployment topologies. Two implementations are conformant if they satisfy the same external contracts (envelope format, delivery guarantees, telemetry semantics) even if their internal mechanisms differ.

## **4.2 Roles and Terms**

This section defines the key roles and terms used throughout the specification. These definitions are **normative**: conformant implementations MUST use these semantics, though the exact naming in APIs or configuration MAY vary if documented.

### 

### **4.2.1 Source Signal**

A **Source Signal** is a typed, timestamped change feed that provides input to Context Views. Source Signals represent events, updates, or arrivals from external systems.

**Required properties:**

* **Type:** Each source MUST declare a type (schema) for its data
* **Timestamp:** Each element MUST carry an event-time timestamp
* **Position:** Each element MUST have a monotone position (offset, sequence number, or equivalent) per partition or key that enables deterministic replay
* **Ordering:** Within a partition or key, positions MUST be monotonically increasing

**Examples:** Event streams (Kafka topics, Kinesis streams), change-data-capture feeds from databases, file arrival notifications, API webhook deliveries, vector embedding updates.

### **4.2.2 Context View**

A **Context View** is a declarative specification of how to compose one or more Source Signals (and/or other Context Views) into a derived context. Views are plans, not stateful entities.

**Required properties:**

* **Determinism:** Given identical inputs and parameters, evaluating a view's plan MUST produce identical output
* **Composability:** Views MAY reference other views as inputs, creating hierarchical compositions
* **Parameterization:** Views MAY accept parameters (e.g., window size, aggregation functions) that are fixed at plan compilation time
* **No mutable state:** Views describe transformations; they do not hold state between evaluations

**Logical operations:** Views typically express joins, filters, windowing, aggregations, and projections. The specific operators and expression language are implementation-defined, but the determinism requirement is normative.

### 

### **4.2.3 Materializer**

The **Materializer** is the executor responsible for:

1. Observing changes in Source Signals
2. Determining which Context Views are affected
3. Applying time semantics (event-time, watermarks, window closure)
4. Evaluating view plans over the appropriate input ranges
5. Constructing and emitting Frames with complete envelopes
6. Invoking extension hooks at defined points

**Required behavior:**

* The Materializer MUST track event-time separately from processing-time
* The Materializer MUST maintain watermarks per (contextId, key) pair
* The Materializer MUST NOT emit frames before window closure (see section 4.4.4)
* The Materializer MUST compute frames deterministically given identical inputs and plans
* The Materializer MUST emit required telemetry (see Annex A.2)

### **4.2.4 Frame**

A **Frame** is an immutable materialization of a Context View at a specific point or interval in event-time. Frames are the durable artifacts that carry context, lineage, and version information.

**Types of frames:**

* **Snapshot Frame:** Represents complete state for the view, key, and window. Consumers can process snapshots independently.
* **Delta Frame:** Represents an incremental change relative to prior state. Consumers MUST fold deltas to reconstruct full state.

**Frame lifecycle states:**

* **Emitted:** Frame published to Frame Store and notified to Subscriptions
* **Accessible:** Frame retrievable by consumers within TTL
* **Expired:** TTL elapsed; frame eligible for deletion per retention policy

Frames MUST be immutable after emission. Corrections or updates MUST generate new frames with incremented versions.

### 

### **4.2.5 Subscription**

A **Subscription** is a delivery relationship in which consumers receive Frames for specified contexts and keys.

**Required guarantees:**

* **At-least-once delivery:** Frames MUST be delivered to subscribers at least once; duplicate deliveries are permitted
* **Per-key ordering:** For a given (contextId, key) pair, frames MUST be delivered in order by (window.end, version)
* **Discovery:** Implementations MUST provide a mechanism for consumers to discover available contexts and subscribe by contextId and optional key filters

### **4.2.6 Consumer**

A **Consumer** is any agent, service, UI component, or analytics pipeline that subscribes to and processes Frames.

**Consumer obligations:**

Consumers MUST process frames idempotently using the idempotencyKey for deduplication. Consumers MUST acknowledge successfully processed frames to allow the subscription system to advance. Consumers SHOULD handle both snapshot and delta frame types if subscribing to views that may emit either.

## **4.3 Portable Materialization Envelope**

Every emitted Frame MUST carry a header (envelope) with the fields defined in this section. The envelope makes frames self-describing, replayable, and governable.

### **4.3.1 Required Header Fields**

Implementations MUST include the following fields in every frame header. Field names shown here are illustrative; exact spelling MAY vary if the semantics are preserved and the mapping is documented in the conformance report.

**Identity and Versioning:**

* **frameId** (string): Globally unique identifier for this frame instance (e.g., ULID, UUID). MUST be unique across the entire system.
* **contextId** (string): Identifies the Context View that produced this frame. MUST be stable across recomputations of the same view.
* **key** (string): Partition or correlation key (e.g., tenant ID, user ID, asset ID). Frames with the same (contextId, key) form an ordered sequence.
* **version** (integer): Monotonically increasing counter per (contextId, key) pair. MUST increment by 1 for each new frame (snapshots and deltas share the same version sequence).

**Frame Type:**

* **type** (enum: "snapshot" | "delta"): Indicates whether this frame represents complete state or an incremental change.

**Time Semantics:**

* **window** (object): Event-time interval for this frame
  + **start** (RFC 3339 timestamp): Beginning of the event-time window (inclusive)
  + **end** (RFC 3339 timestamp): End of the event-time window (exclusive)
* **watermarkAt** (RFC 3339 timestamp): Watermark observed at frame emission. Indicates "no events with event-time earlier than this are expected" for this context and key.

**Provenance:**

* **inputs** (array of objects): Describes the source data that contributed to this frame. Each input object MUST contain:
  + **sourceId** (string): Identifier of the source signal or upstream view
  + **from** (string): Starting position (offset, sequence number, timestamp)
  + **to** (string): Ending position (inclusive or exclusive per implementation; MUST be documented)
  + **hash** (string): Cryptographic hash (e.g., SHA-256) of the input range content for integrity verification

**Determinism:**

* **planHash** (string): Cryptographic hash of the view's compiled plan, including operator versions, library dependencies, and configuration. MUST change if and only if the effective transform changes.
* **idempotencyKey** (string): Stable hash computed from contextId, key, window, canonicalized inputs summary, and planHash. MUST be identical for identical work (same inputs, same plan, same window).

**Lifecycle:**

* **ttl** (ISO 8601 duration): Time-to-live for this frame. After TTL expires, the frame MAY be deleted per retention policy. A value of PT0S indicates ephemeral (delete immediately after delivery to all current subscribers).
* **createdAt** (RFC 3339 timestamp): Wall-clock time when the frame was emitted.

**Extensions:**

* **extensions** (object, optional): Implementation-defined metadata. Common uses include security classification, redaction paths, budget usage, approval status, and custom tags. The structure and semantics of this field are implementation-defined but SHOULD be documented.

### **4.3.2 Additional Fields**

Implementations MAY include additional header fields beyond those required above. Any additional fields SHOULD be documented in the conformance report.

### **4.3.3 Immutability Requirement**

Frames MUST NOT be mutated after publication. If a correction or update is needed, the implementation MUST emit a new frame with an incremented version. The new frame MAY reference the prior version (e.g., via extensions metadata) but MUST NOT alter the prior frame's content.

### **4.3.4 Uniqueness and Monotonicity**

**frameId uniqueness:** No two frames in the system MAY share the same frameId, even across different contexts or time periods.

**version monotonicity:** For a given (contextId, key) pair, version numbers MUST form a monotonically increasing sequence with no gaps or reuse. If frame N has version=42, the next frame for that (contextId, key) MUST have version=43.

**Implication:** Implementations MUST coordinate version assignment to prevent concurrent materializers from emitting conflicting versions for the same key. Common strategies include single-writer-per-key (actor model) or optimistic concurrency with retry.

### 

### **4.3.5 Provenance Sufficiency**

The inputs field MUST contain sufficient information to:

1. Identify all source data that contributed to the frame
2. Retrieve those inputs for replay (by position ranges)
3. Verify input integrity (via hashes)

If exact replay is impossible (e.g., due to non-deterministic external API calls), the implementation MUST document this limitation and MAY include API response hashes or snapshots in the inputs to enable best-effort verification.

### **4.3.6 Stable Idempotency Key**

The idempotencyKey MUST be a pure function of:

* contextId
* key
* window (both start and end)
* Canonicalized summary of inputs (e.g., sorted array of {sourceId, from, to, hash})
* planHash

**Canonicalization:** The inputs summary MUST be deterministically ordered (e.g., lexicographically by sourceId) so that different orderings of the same inputs produce the same key.

**Purpose:** Duplicate work (same inputs, same plan, same window) MUST produce the same idempotencyKey, allowing publishers and subscribers to deduplicate without coordination.

**Illustrative algorithm (informative):**

function makeIdempotencyKey(contextId, key, window, inputs, planHash):

inputsSummary = sort(inputs, by: sourceId).map(i => [i.sourceId, i.from, i.to, i.hash])

canonical = {contextId, key, window, inputsSummary, planHash}

return sha256(serialize(canonical))

Implementations MAY use different hash algorithms (e.g., BLAKE3) if documented.

## 

## **4.4 Time, Ordering, and Reactive Recomputation**

RCM's time semantics ensure that frames are emitted at deterministic, explainable points in event-time, even when processing is out-of-order or delayed.

### **4.4.1 Event Time vs. Processing Time**

**Event time** is the timestamp reflecting when an event actually occurred in the source domain (e.g., when a user clicked, when a sensor reading was taken).

**Processing time** is the wall-clock time when the system observes or processes the event.

**Requirement:** Implementations MUST distinguish event-time from processing-time. Frames MUST be associated with event-time windows (via the window field), not processing-time intervals.

**Rationale:** Event-time semantics enable deterministic replay and correct handling of out-of-order data. Processing-time semantics produce non-reproducible results when replay order differs from original order.

### **4.4.2 Watermarks**

A **watermark** is a threshold in event-time that indicates "the system believes no events with timestamps earlier than this will arrive" for a given (contextId, key).

**Requirements:**

* Implementations MUST maintain a watermark per (contextId, key) pair
* Watermarks MUST advance monotonically (never decrease)
* Watermark advancement MAY be based on heuristics (e.g., "highest timestamp seen + epsilon") or explicit source signals (e.g., Kafka timestamps, Flink watermarks)
* The watermark observed at frame emission MUST be recorded in the frame's watermarkAt field

**Purpose:** Watermarks bound staleness and enable deterministic window closure in the presence of out-of-order events.

### 

### **4.4.3 Windows**

A **window** is an event-time interval [start, end) that defines the scope of data included in a frame.

**Window kinds:**

Implementations MAY support different windowing strategies:

* **Tumbling windows:** Fixed-size, non-overlapping intervals (e.g., every 1 minute)
* **Sliding windows:** Fixed-size, overlapping intervals (e.g., 5-minute window sliding every 1 minute)
* **Session windows:** Data-driven intervals closed after a gap of inactivity

The specific windowing strategy is a property of the Context View and MUST be documented per view.

**Requirements:**

* Each frame MUST declare its event-time window via the window field (start, end)
* Window boundaries MUST align with event-time, not processing-time
* For a given (contextId, key), windows MUST NOT overlap in time for snapshot frames (deltas MAY correct prior windows)

### **4.4.4 Window Lifecycle**

Windows progress through the following normative states:

1. **Open:** Window is defined but watermark has not yet reached window.start
2. **Accepting:** Watermark ≥ window.start and < window.end; events within the window are accumulated
3. **Closing:** Watermark ≥ window.end; no new events expected; frame emission is imminent
4. **Emitted:** Frame published with this window
5. **Expired:** Frame TTL elapsed; eligible for deletion

**Requirements:**

* Implementations MUST NOT emit a frame for a window before the window enters the **Closing** state (i.e., before watermark ≥ window.end)
* Implementations MAY emit additional frames (deltas) for the same window after **Emitted** if late data arrives within a configured lateness bound
* Late frames MUST increment version and MUST indicate their corrective nature (e.g., via type=delta or extensions metadata)

**Illustrative state diagram (informative):**

[\*] → Open → Accepting → Closing → Emitted → Expired

↑ ↓

└─ late data─┘ (optional, if policy allows)

### **4.4.5 Reactivity Policy**

Upon detecting a change in a Source Signal, the Materializer MUST determine whether to recompute and emit a new frame. The decision MUST follow a declared per-view policy:

1. **Immediate recompute:** Emit a new frame as soon as the change is observed (minimizes staleness, maximizes version churn)
2. **Coalesce:** Delay emission for a small, bounded duration to absorb multiple rapid changes into a single frame (reduces churn, adds bounded latency)
3. **Ignore:** Do not recompute if the change falls outside the view's windows or does not match filter criteria

**Requirements:**

* The reactivity policy MUST be fixed per Context View (not dynamic per event)
* If using coalescing, the coalescing duration MUST be bounded and documented
* Implementations MUST emit telemetry indicating when coalescing occurs (e.g., coalesce.triggered, coalesce.duration\_ms)

**Coalescing semantics:** When coalescing, the Materializer MUST emit a frame that reflects all changes observed during the coalescing window. The resulting frame's inputs field MUST cover the full range of data, and the idempotencyKey MUST be stable (recomputing with the same coalesced inputs produces the same key).

### **4.4.6 Late and Duplicate Data Handling**

**Late data:** Events that arrive with event-time timestamps earlier than the current watermark for their key.

**Duplicate data:** Events with identical positions or content observed more than once.

**Requirements:**

* Implementations MUST define a per-view **lateness policy** that specifies:
  + Maximum allowed lateness (e.g., "accept events up to 1 minute late")
  + Action when late data is accepted: emit corrective delta frame with incremented version
  + Action when late data exceeds the bound: drop and log, or route to a separate late-data context
* Implementations MUST detect duplicate inputs (via position or hash) and SHOULD avoid redundant recomputation by matching idempotencyKey
* Implementations MUST emit telemetry for late/duplicate handling (e.g., late\_data.accepted, late\_data.dropped, duplicate.detected)

**Corrective frames:** When late data triggers recomputation, the new frame MUST:

* Have the same contextId, key, and window as the original
* Have an incremented version
* Indicate corrective nature (e.g., type=delta with metadata explaining the correction)

### **4.4.7 Ordering Guarantee**

**Requirement:** Delivery to a subscriber MUST preserve per-key ordering by (window.end, version).

**Specification:**

For any two frames F1 and F2 with the same (contextId, key):

* If F1.window.end < F2.window.end, then F1 MUST be delivered before F2
* If F1.window.end == F2.window.end and F1.version < F2.version, then F1 MUST be delivered before F2

**Cross-key ordering:** Implementations MAY deliver frames for different keys in any order (no global ordering required).

**Implications:**

* Subscribers can rely on receiving frames in causal order for a given key
* Snapshots and deltas interleave correctly (both share the version sequence)
* Replay by version range produces the same order as original emission

**Tolerance for duplicates:** The ordering guarantee applies to the logical sequence; duplicate deliveries (same idempotencyKey) MAY occur in any position. Consumers MUST deduplicate as specified in section 4.6.3.

## **4.5 Determinism, Idempotency, and Replay**

Deterministic materialization and idempotent delivery are foundational to RCM's explainability and cost control. This section specifies the requirements that enable safe replay, dual-run validation, and effect-once semantics despite at-least-once delivery.

### **4.5.1 Deterministic Plans**

A Context View's plan MUST be **referentially transparent**: given identical inputs and parameters, evaluating the plan MUST produce identical output.

**Requirements:**

* **Pure functions:** Plan operators MUST NOT depend on mutable global state, wall-clock time (except as explicit input), random number generators with uncontrolled seeds, or external I/O with non-deterministic responses
* **Stable operators:** The plan MUST specify versions of all operators, libraries, and configuration. Changing any of these MUST produce a different planHash
* **Canonicalization:** If the plan includes operators that can be reordered (e.g., commutative aggregations), the plan representation MUST canonicalize to a stable form

**Allowed non-determinism:**

Implementations MAY permit controlled non-determinism if:

1. The non-deterministic source is explicitly declared (e.g., external API call)
2. Results are captured in the frame's inputs (e.g., API response hash or snapshot)
3. Replay uses captured results rather than re-executing non-deterministic operations
4. The limitation is documented in the conformance report

### **4.5.2 Plan Hash**

The planHash field MUST uniquely identify the effective transform applied to produce a frame.

**Requirements:**

* planHash MUST be a cryptographic hash (e.g., SHA-256) of:
  + The compiled plan's operator graph (structure and parameters)
  + Versions of all operators and libraries
  + Relevant configuration (e.g., schema versions, feature flags that affect computation)
* planHash MUST change if and only if the effective transform changes (reordering of equivalent expressions SHOULD NOT change the hash if canonicalized)
* planHash MUST be recorded in every frame header

**Purpose:** The planHash enables:

* Verification that two frames were computed with the same logic
* Detection of plan drift (unintended changes to view definitions)
* Safe dual-run of plan changes (compare frames with different planHash values before cutover)

**Illustrative algorithm (informative):**

function computePlanHash(plan):

canonical = canonicalize(plan.operators, plan.params, plan.versions, plan.config)

return sha256(serialize(canonical))

### **4.5.3 Idempotency Key**

The idempotencyKey MUST be a stable hash that uniquely identifies a unit of work, such that repeated execution of the same work produces the same key.

**Requirements:**

* The idempotencyKey MUST be computed as a pure function of:
  + contextId
  + key
  + window (both start and end)
  + Canonicalized summary of inputs (source IDs, position ranges, content hashes)
  + planHash
* The computation MUST be deterministic and order-independent where applicable (e.g., if inputs can arrive in different orders, they MUST be sorted before hashing)
* Two frames with identical idempotencyKey values MUST have identical frame bodies (ignoring non-semantic fields like frameId and createdAt)

**Idempotency semantics:**

* **Publishers:** If the Materializer computes a frame and discovers that an identical idempotencyKey already exists in the Frame Store, it MAY skip publication and increment a duplicate.materialization metric. The existing frame serves as the result.
* **Subscribers:** If a consumer receives a frame with an idempotencyKey it has already processed, it MUST skip re-processing the frame body and MUST NOT produce duplicate side effects.

**Canonical inputs summary (normative):**

The inputs summary MUST be deterministically ordered. Implementations MUST either:

1. Sort inputs lexicographically by sourceId, then by from position, OR
2. Use a content-addressed structure (e.g., Merkle tree) that is order-independent

**Illustrative algorithm (informative):**

function makeIdempotencyKey(contextId, key, window, inputs, planHash):

sortedInputs = sort(inputs, by: [sourceId, from])

summary = sortedInputs.map(i => {i.sourceId, i.from, i.to, i.hash})

canonical = {contextId, key, window.start, window.end, summary, planHash}

return sha256(serialize(canonical))

### **4.5.4 Replay Requirements**

Implementations MUST support replay: the ability to re-emit previously materialized frames by referencing their identity or time range.

**Replay by version:**

* **Specification:** Given (contextId, key, version), the implementation MUST return the frame that was originally emitted with that version number
* **Fidelity:** The replayed frame MUST have the same idempotencyKey and frame body as the original
* **New frameId:** The replayed frame MAY have a different frameId (since it's a new emission event) but MUST preserve all other header fields except createdAt

**Replay by time interval:**

* **Specification:** Given (contextId, key, startTime, endTime), the implementation MUST return all frames whose window.end falls within [startTime, endTime)
* **Ordering:** Frames MUST be returned in order by (window.end, version)
* **Completeness:** All frames emitted within the specified interval MUST be included, subject to TTL and retention policies

**Replay limitations:**

* Frames that have exceeded their TTL and been deleted MAY NOT be available for replay; implementations MUST document retention policies
* If source data required for re-evaluation has been garbage-collected, replay MUST use the stored frame rather than recomputing from sources

**Telemetry:**

Implementations MUST emit telemetry distinguishing original emissions from replays (e.g., frame.materialized vs. frame.replayed, or a replay=true flag in events).

### **4.5.5 Golden Replay Vectors**

Implementations claiming conformance MUST demonstrate determinism by executing **golden replay tests**: pre-recorded input vectors that specify sources, windows, and plans, with expected output frames.

**Test procedure:**

1. Load a golden vector: {contextId, key, window, inputs, plan, expectedIdempotencyKey, expectedBody}
2. Evaluate the plan over the inputs
3. Assert that the resulting frame's idempotencyKey matches expectedIdempotencyKey
4. Assert that the resulting frame's body matches expectedBody (exact match or semantic equivalence per documented rules)
5. Repeat evaluation (N ≥ 3 times) and assert identical results each time

Golden vectors for conformance testing are provided in Annex A.5.

### **4.5.6 Dual-Run and Plan Evolution**

When evolving a view's plan (e.g., adding a join, changing an aggregation), implementations SHOULD support **dual-run**: evaluating both old and new plans over the same inputs to verify that the change produces expected differences.

**Dual-run procedure (informative):**

1. Compute planHashOld and planHashNew
2. For a sample of keys and windows, evaluate both plans
3. Compare resulting frames:
   * If bodies are identical, the change may be a refactoring with no semantic impact
   * If bodies differ, inspect differences for correctness
4. Emit telemetry: plan.dual\_run.diff with summary statistics (keys tested, differences found, max divergence)
5. Cut over to planHashNew only after validation

**Requirement:** Implementations MUST preserve the ability to compute both planHashOld and planHashNew during dual-run (e.g., by versioning view definitions and allowing concurrent evaluation).

## **4.6 Delivery Contract**

This section specifies the guarantees that RCM subscriptions provide to consumers.

### **4.6.1 At-Least-Once Delivery**

**Guarantee:** For every frame emitted by the Materializer, each subscriber MUST receive the frame at least once.

**Implications:**

* Subscribers MAY receive duplicate deliveries of the same frame (same frameId or idempotencyKey)
* Subscribers MUST NOT assume exactly-once delivery
* Subscribers MUST implement idempotent processing using idempotencyKey for deduplication

**Failure handling:**

* If a subscriber is unavailable or fails to acknowledge, the subscription system MUST retry delivery with exponential backoff
* Retry attempts MUST be bounded (maximum retries or maximum time) to avoid indefinite blocking
* After exhausting retries, the frame MUST be routed to a dead-letter queue (DLQ) for remediation

### **4.6.2 Per-Key Ordering**

**Guarantee:** For frames with the same (contextId, key), delivery MUST preserve order by (window.end, version).

**Specification:**

Given two frames F1 and F2 where:

* F1.contextId == F2.contextId
* F1.key == F2.key
* Either F1.window.end < F2.window.end, OR F1.window.end == F2.window.end and F1.version < F2.version

Then F1 MUST be delivered to each subscriber before F2.

**Cross-key independence:** Frames with different keys MAY be delivered in any order, including concurrently.

**Ordering under retries:**

If a frame F1 fails and is retried, the subscription system MUST NOT deliver subsequent frames (F2, F3, ...) for the same key until F1 is successfully acknowledged or moved to the DLQ.

**Head-of-line blocking:** Per-key ordering implies that a single stuck frame blocks all subsequent frames for that key. Implementations SHOULD provide mechanisms to detect and handle blocked keys (e.g., timeouts, separate retry queues, alerts).

### **4.6.3 Idempotent Consumption**

**Requirement:** Consumers MUST process frames idempotently, such that receiving the same frame multiple times produces the same effect once.

**Implementation pattern (informative):**

function onFrame(frame):

if alreadyProcessed(frame.idempotencyKey):

acknowledge(frame)

return

effect = computeEffect(frame.body)

applyEffect(effect)

recordProcessed(frame.idempotencyKey)

acknowledge(frame)

**Deduplication window:**

Consumers SHOULD maintain a deduplication window (e.g., last 10,000 idempotencyKey values or last 24 hours) to efficiently detect duplicates. Implementations MAY use probabilistic structures (Bloom filters) with acceptable false-positive rates if documented.

**Stateless consumers:**

Consumers that cannot maintain deduplication state (e.g., ephemeral functions) MUST ensure that their effects are naturally idempotent (e.g., writing to an idempotent store, using database upserts with unique constraints on idempotencyKey).

### 

### **4.6.4 Retry and Dead-Letter Queue**

**Retry semantics:**

* Implementations MUST retry failed deliveries automatically
* Retry intervals SHOULD use exponential backoff with jitter (e.g., 1s, 2s, 4s, 8s, ...)
* Maximum retry attempts MUST be configurable (recommended default: 10 retries over ~15 minutes)

**Dead-Letter Queue (DLQ):**

* Frames that exceed the retry limit MUST be moved to a DLQ
* DLQ entries MUST include:
  + Full frame headers and body (or reference to the stored frame)
  + Subscriber identity
  + Failure reason (exception message, status code)
  + Timestamp of final failure
  + Count of retry attempts
* Implementations MUST provide a mechanism to inspect DLQ entries and re-drive delivery after remediation

**Telemetry:**

* Implementations MUST emit events for each retry attempt: delivery.retry with labels {contextId, key, version, subscriberId, attemptNumber, reason}
* Implementations MUST emit events for DLQ routing: delivery.dlq with labels {contextId, key, version, subscriberId, reason}

### **4.6.5 Subscription Discovery**

**Requirement:** Implementations MUST provide a mechanism for consumers to discover available contexts and subscribe.

**Discovery API (informative):**

* **List contexts:** Return all available contextId values with metadata (description, schema, typical keys)
* **Describe context:** Return schema, windowing policy, expected emission rate, typical frame size
* **Subscribe:** Bind to (contextId, keyFilter, fromVersion?) where:
  + keyFilter is an optional predicate (e.g., key LIKE 'tenant-123-%') to receive only relevant frames
  + fromVersion allows starting from a specific version (useful for catch-up or replay)

**Subscription lifecycle:**

* Consumers SHOULD be able to pause, resume, and cancel subscriptions
* Implementations SHOULD support multiple subscribers per context without interference (fan-out)
* Implementations SHOULD provide telemetry on subscriber health (lag, throughput, error rates)

### **4.6.6 Backpressure and Flow Control**

**Requirement:** Implementations SHOULD support backpressure mechanisms to prevent overwhelming slow consumers.

**Reactive Streams alignment (informative):**

If the implementation uses Reactive Streams or a similar protocol, it SHOULD follow the Publisher/Subscriber contract:

* Consumers signal demand (e.g., "I can handle 10 more frames")
* Publishers respect demand and do not push beyond capacity
* Unbounded demand is discouraged; consumers SHOULD request in bounded batches

**Alternative mechanisms:**

* Consumers MAY use acknowledgment pacing (delay ack until ready for next frame)
* Subscriptions MAY implement buffering with high-water marks and alerts

**Telemetry:**

* Implementations SHOULD emit metrics: subscription.backlog (frames pending delivery), subscriber.lag\_ms (time since frame emission to delivery)

## **4.7 Extension Points**

RCM defines **extension points**: well-defined hooks where implementations inject cross-cutting behaviors. The pattern specifies *where* these hooks attach and *what telemetry* they emit; the specific policies are **implementation-defined**.

### **4.7.1 Purpose and Scope**

Extension points allow implementations to address operational concerns—admission control, resource management, security, observability—without coupling them to the core reactive semantics.

**Design principles:**

* Extensions are **orthogonal** to composition, determinism, and delivery guarantees
* Extensions inject at defined stages (before materialization, before publish, at delivery)
* Extensions MUST emit telemetry documenting their decisions
* Extensions MUST NOT alter frame content or violate ordering guarantees (they may delay, deny, or observe, but not mutate)

### **4.7.2 Stages and Hooks**

Implementations claiming to support extension points SHOULD document which of the following hooks they provide:

**1. Pre-Materialization Hook (Before Evaluation)**

Invoked after a source change is detected but before the Materializer evaluates the view plan.

**Typical uses:**

* Policy checks (access control: "Is this context allowed for this key?")
* Rate limiting (admission control: "Has this tenant exceeded their recomputation quota?")
* Concurrency control (capacity: "Are we at maximum in-flight materializations?")
* Priority assignment (scheduling: "What priority should this work receive?")

**Extension interface (informative):**

onBeforeMaterialize(contextId, key, sourceChange):

decision = {admit: bool, reason: string, delay: duration?}

emit("admission.decision", {stage: "pre-materialize", contextId, key, decision})

return decision

**If denied:** The Materializer MUST NOT evaluate the plan. It SHOULD queue the work for retry if the denial is temporary (e.g., rate limit) or drop it permanently if policy forbids it.

**2. Pre-Publish Hook (Before Frame Emission)**

Invoked after the frame is computed but before it is written to the Frame Store and notified to subscribers.

**Typical uses:**

* Budget reservation ("Reserve 120ms compute time and 5MB network")
* Security classification ("Tag this frame as 'internal' based on content")
* Redaction ("Remove PII fields per policy before storage")
* Human-in-the-loop approval ("Request approval before exposing high-risk frames")

**Extension interface (informative):**

onBeforePublish(frameHeaders, frameBody):

lease = budgets.tryLease({timeMs: 120, moneyCents: 5, netBytes: frameSize})

if !lease.granted:

return {admit: false, reason: "budget exhausted"}

classification = classify(frameBody)

redactedBody = applyRedactions(frameBody, classification)

emit("budget.lease", {leaseId, units: {timeMs: 120, ...}})

emit("security.classification", {contextId, key, classification})

emit("security.redaction", {contextId, key, pathsRedacted: [...]})

return {admit: true, body: redactedBody, metadata: {classification, leaseId}}

**If denied:** The frame MUST NOT be published. The extension SHOULD emit telemetry explaining the denial (e.g., admission.decision with stage=pre-publish, verdict=deny, reason="budget exceeded").

**3. Post-Publish Hook (After Successful Emission)**

Invoked after the frame is stored and notified to the subscription bus, before the Materializer moves to the next work item.

**Typical uses:**

* Budget settlement (commit actual usage: "We used 95ms, not the reserved 120ms")
* Metrics collection (frame size, recomputation time, input cardinality)
* Audit logging (record high-value frame emissions)

**Extension interface (informative):**

onAfterPublish(frameId, actualUsage):

budgets.commit(leaseId, actualUsage)

emit("budget.commit", {leaseId, actual: {timeMs: 95, ...}})

emit("frame.size\_bytes", frameSize)

**4. Delivery Hook (At Subscription)**

Invoked when delivering a frame to a subscriber, before the frame is sent.

**Typical uses:**

* Edge policy enforcement (gateway checks: "Is this subscriber allowed to see this classification?")
* Budget accounting (charge for egress: "This delivery costs 2MB network")
* Observability (latency tracking: "Emit delivery.latency\_ms for this subscriber")

**Extension interface (informative):**

onBeforeDeliver(frame, subscriberId):

if !hasAccess(subscriberId, frame.extensions.classification):

emit("delivery.denied", {subscriberId, contextId, key, reason: "access denied"})

return {admit: false}

emit("delivery.attempt", {subscriberId, contextId, key, version})

return {admit: true}

### **4.7.3 Required Telemetry for Extensions**

Implementations that support extension points MUST emit the following telemetry events to make extension behavior observable:

**admission.decision**

* Labels: {stage, contextId, key, verdict, reasonCode}
* stage ∈ {pre-materialize, pre-publish, delivery}
* verdict ∈ {admit, deny, defer}
* reasonCode: implementation-defined (e.g., rate\_limit\_exceeded, policy\_denied, budget\_unavailable)

**budget.lease / budget.commit / budget.cancel**

* Labels: {contextId, key, leaseId, units}
* units: object with dimensions (e.g., {timeMs, moneyCents, tokens, netBytes, rows})
* budget.lease: reservation made
* budget.commit: actual usage recorded after successful completion
* budget.cancel: reservation released after failure or denial

**security.classification**

* Labels: {contextId, key, classification}
* classification: implementation-defined taxonomy (e.g., public, internal, confidential, restricted)

**security.redaction**

* Labels: {contextId, key, pathsRedacted}
* pathsRedacted: array of JSON pointers or field paths that were masked/removed

**human.approval.requested / human.approval.granted / human.approval.denied**

* Labels: {contextId, key, approvalId, ttl}
* Emitted when a frame requires human approval before exposure
* ttl: time window for human to respond

### **4.7.4 Implementation-Defined Policies**

RCM specifies the hooks and telemetry; the specific policies are implementation-defined. Common patterns are documented in Annex D (Informative), including:

* Multi-stage governance chains (e.g., Policy → Gate → Concurrency → Priority → Budget)
* Simple rate limiting (token bucket, leaky bucket)
* Policy-as-code integration (OPA, Cedar)
* Budget models (time, money, tokens, network, rows)
* Classification and redaction rules
* Fairness schedulers (weighted fair queueing with aging)

Implementations SHOULD document their extension policies in the conformance report, including:

* Which hooks are provided
* What policies are applied at each hook
* How policies are configured
* Examples of common configurations

### **4.7.5 Guarantees Preserved by Extensions**

Extensions MUST NOT violate RCM's core guarantees:

* **Determinism:** Extensions MAY delay or deny materialization but MUST NOT alter the frame body or idempotencyKey (redaction is permitted if applied before frame identity is finalized and documented as part of the plan)
* **Ordering:** Extensions MAY delay delivery but MUST NOT reorder frames (per-key ordering by window.end, version must hold)
* **At-least-once delivery:** Extensions MAY retry or route to DLQ but MUST ensure every admitted frame reaches subscribers or DLQ
* **Idempotency:** Extensions MAY deduplicate work but MUST preserve idempotencyKey semantics

## **4.8 Summary of Requirements**

This section provides a consolidated checklist of normative requirements for RCM conformance.

### **4.8.1 Core Pattern (All Implementations)**

**Roles and Components:**

* ✓ Source Signals with monotone positions and event-time timestamps (§4.2.1)
* ✓ Context Views as deterministic, composable plans (§4.2.2)
* ✓ Materializer with time-aware execution (§4.2.3)
* ✓ Frames as immutable, versioned artifacts (§4.2.4)
* ✓ Subscriptions with defined delivery guarantees (§4.2.5)

**Frame Envelope (§4.3):**

* ✓ Required header fields: frameId, contextId, key, version, type, window, watermarkAt, inputs, planHash, idempotencyKey, ttl, createdAt
* ✓ Immutability: frames MUST NOT mutate after publication
* ✓ Uniqueness: frameId globally unique; version monotone per (contextId, key)
* ✓ Provenance sufficiency: inputs enable replay

**Time and Ordering (§4.4):**

* ✓ Event-time vs. processing-time distinction
* ✓ Watermarks per (contextId, key), monotonically advancing
* ✓ Window lifecycle: no emission before closure
* ✓ Reactivity policy: documented per view
* ✓ Late/duplicate data handling: documented per view
* ✓ Per-key ordering: delivery by (window.end, version)

**Determinism and Replay (§4.5):**

* ✓ Deterministic plans: same inputs → same output
* ✓ Stable planHash: identifies transform identity
* ✓ Stable idempotencyKey: identifies work identity
* ✓ Replay by version: (contextId, key, version) → exact frame
* ✓ Replay by time: (contextId, key, interval) → ordered frames
* ✓ Golden replay vectors: pass conformance tests (Annex A.5)

**Delivery Contract (§4.6):**

* ✓ At-least-once delivery to all subscribers
* ✓ Per-key ordering preserved under retries
* ✓ Idempotent consumption required of consumers
* ✓ Retry with backoff and DLQ routing
* ✓ Subscription discovery mechanism

### **4.8.2 Extension Points (Optional but Recommended)**

**If supporting extensions (§4.7):**

* ✓ Document which hooks are provided (pre-materialize, pre-publish, post-publish, delivery)
* ✓ Emit required telemetry: admission.decision, budget.\*, security.\*
* ✓ Preserve core guarantees (determinism, ordering, at-least-once)
* ✓ Document policies applied at each hook

### **4.8.3 Telemetry (§4.7.3, Annex A.2)**

**Required events:**

* ✓ frame.materialized (or equivalent semantic)
* ✓ view.window.closed
* ✓ delivery.attempt, delivery.retry, delivery.dlq
* ✓ Extension events if hooks are implemented

**Required metrics:**

* ✓ watermark\_lag\_ms (freshness)
* ✓ recompute\_latency\_ms (materialization time)
* ✓ delivery\_latency\_ms (end-to-end delivery time)
* ✓ frame.size\_bytes
* ✓ delivery.retry.count, delivery.dlq.count

### **4.8.4 Conformance Verification**

**To claim conformance:**

1. ✓ Implement all requirements in §§4.2-4.6
2. ✓ Pass test vectors in Annex A.5
3. ✓ Emit telemetry per Annex A.2
4. ✓ Produce conformance report per Annex A.6

**Next:** Chapter 5 positions RCM against adjacent patterns, helping architects decide when RCM applies and how it composes with existing systems.

This completes Chapter 4 (Normative Semantics). Shall I proceed with Chapter 5 (Comparative Guide)?

# 

# **5. Comparative Guide**

This chapter positions RCM against well-known patterns in distributed systems, event-driven architectures, and UI state management. The goal is to help architects recognize when RCM applies, understand how it differs from (and complements) adjacent patterns, and avoid false equivalences.

## **5.1 RCM vs. Adjacent Patterns**

The following table provides a high-level comparison. Subsequent sections elaborate on the relationships and provide decision heuristics.

|  |  |  |
| --- | --- | --- |
| **Pattern** | **Primary Purpose** | **How RCM Relates** |
| **Event Sourcing** | Persist all state changes as append-only events; rebuild state via replay | RCM often consumes event-sourced streams but treats frames as derived, versioned read models rather than authoritative event logs. Event Sourcing says "the log is truth"; RCM says "frames are current composite knowledge derived from truth." |
| **CQRS** | Separate write models (commands) from read models (queries) for scalability and clarity | RCM frames are read models with governance and lineage. CQRS doesn't prescribe reactive updates, watermarks, or delivery contracts; RCM provides those semantics. |
| **Materialized Views (DB/Streaming)** | Precompute query results and keep them incrementally updated for read performance | RCM is a methodized materialized view pattern: it standardizes envelope format, time semantics, lineage, versioning, delivery, and governance so views become a reusable memory plane, not just a database optimization. |
| **Reactive Streams** | Non-blocking backpressure protocol for asynchronous stream processing | RCM borrows flow control concepts but adds memory lifecycle (versioning, TTL, replay) and governance. Reactive Streams standardizes demand signaling; RCM standardizes context materialization and delivery. |
| **Flux / Redux / MVU / MVI** | Unidirectional state flow for UI (actions → reduce → state → render) | Same backbone principle (events → deterministic reduce → state). RCM extends it to durable, multi-consumer, governed memory with time semantics and cross-session replay. |
| **Blackboard Systems** | Shared workspace where specialists read/write partial solutions cooperatively | RCM plays a similar "shared memory" role but with explicit reactivity, versioning, lineage, and delivery contracts. Blackboard systems often lack standardized governance and replay. |
| **Knowledge Graphs** | Graph-structured knowledge representation with rich semantics (RDF, Property Graphs) | RCM can compose views over knowledge graph queries and emit frames referencing graph provenance. KGs are storage/query substrates; RCM governs the reactive lifecycle of derived contexts. |
| **Rule Engines (Rete, Drools)** | Efficient pattern matching and forward-chaining inference for business rules | Rules can fire on frame updates (frames feed working memory); rule outputs can trigger new views. RCM provides the memory substrate; rule engines provide the inference. |
| **Caching Layers (Redis, Memcached)** | Fast key-value lookups to reduce load on authoritative stores | RCM frames can be cached, but the pattern isn't a cache. Caches lack lineage, versioning, reactive updates, and governance. RCM standardizes *how* cacheable artifacts form, not just *where* they're stored. |

### 

### **5.1.1 Event Sourcing**

**What it is:** Event Sourcing persists every state change as an immutable event in an append-only log. Current state is derived by replaying events from the beginning (or from a snapshot). The event log is the authoritative source of truth.

**Relationship to RCM:**

* **Complementary:** RCM commonly consumes event-sourced streams as Source Signals. The event log remains the write-side truth; RCM builds governed read-side frames.
* **Different concerns:** Event Sourcing focuses on *authoritative history* (what happened, in what order). RCM focuses on *composite, versioned context* (what we know now, reactively updated, with lineage and governance).
* **Co-existence pattern:** Use Event Sourcing for command handling and durable ledger; use RCM to compose views over events and serve agents, dashboards, analytics.

**Example architecture:**

Commands → Event Store (append-only)

↓

RCM Source Signal

↓

Context Views (join events + reference data)

↓

Frames (versioned read models)

↓

Agents, Services, Dashboards

**When to use Event Sourcing without RCM:** You need a complete audit trail and temporal queries ("what was the state at time T?") but don't need reactive multi-source composition or governed delivery.

**When to add RCM to Event Sourcing:** You have event-sourced data and need to serve real-time, composite contexts to many consumers with freshness SLOs, explainability, and cost control.

### 

### **5.1.2 CQRS (Command-Query Responsibility Segregation)**

**What it is:** CQRS separates the write model (commands that mutate state) from read models (queries optimized for specific access patterns). Commands and queries use different schemas and often different storage.

**Relationship to RCM:**

* **RCM implements the read side:** Frames are read models with RCM's additional semantics (versioning, lineage, time, delivery).
* **CQRS is architectural principle; RCM is pattern:** CQRS says "separate writes and reads"; RCM says "here's how to build reactive, governed read models."
* **CQRS doesn't specify:** How read models update (RCM: reactively on source change), how they version (RCM: monotone per key), how they're delivered (RCM: at-least-once, per-key ordered), or how they're governed (RCM: extension points).

**Co-existence pattern:**

Write Side (Commands):

Commands → Aggregate → Events → Event Store

Read Side (Queries):

Event Store → RCM Views → Frames → Subscriptions → Consumers

**When to use CQRS without RCM:** You need read/write separation but can serve queries with simple projections or direct database queries; reactive updates and multi-consumer fan-out aren't critical.

**When to add RCM to CQRS:** Your read side needs reactive freshness, multi-source joins, versioning for replay, and consistent delivery to many heterogeneous consumers (agents, dashboards, indices).

### 

### **5.1.3 Materialized Views (Databases and Streaming Engines)**

**What it is:** A materialized view is a precomputed query result stored for fast access. Database MVs (PostgreSQL, Oracle) update via triggers or refresh schedules. Streaming MVs (Kafka Streams, Flink) update incrementally as input streams change.

**Relationship to RCM:**

* **RCM generalizes and governs:** RCM is "materialized views as a first-class memory pattern" with standardized envelope, time semantics, lineage, versioning, delivery, and governance.
* **Database MVs lack:** Reactive cross-source composition, delivery contracts, versioning for replay, governance hooks.
* **Streaming MVs provide:** Incremental updates and time semantics (watermarks, windows) but often lack standardized lineage envelopes, multi-consumer delivery guarantees, and governance.

**RCM adds value by:**

* Standardizing the frame envelope so MVs are self-describing and replayable
* Providing delivery contracts (at-least-once, per-key ordered) for fan-out
* Adding extension points for policy, budgets, security
* Enabling cross-platform interoperability (RCM frames from Flink can be consumed by non-Flink subscribers)

**When to use simple MVs:** You need fast reads within a single database or streaming job; context is simple (single source or basic join); no cross-team reuse or governance required.

**When to adopt RCM:** You need to share materialized contexts across many consumers (agents, services, dashboards, analytics), provide lineage for compliance, control costs via budgets, and govern access/retention uniformly.

### 

### **5.1.4 Reactive Streams**

**What it is:** Reactive Streams is a specification for asynchronous stream processing with non-blocking backpressure. It defines Publisher, Subscriber, Subscription, and Processor interfaces with demand signaling.

**Relationship to RCM:**

* **Complementary, orthogonal concerns:** Reactive Streams standardizes *flow control* (how to avoid overwhelming consumers). RCM standardizes *memory lifecycle* (how context forms, versions, persists, and delivers).
* **RCM subscriptions can use Reactive Streams:** Implementations MAY use Reactive Streams protocols for delivery (subscribers signal demand; publishers respect it) but this is not required.
* **Reactive Streams doesn't specify:** What's being delivered (RCM: versioned frames with lineage), how it's computed (RCM: deterministic plans, watermarks), how it's stored (RCM: immutable frame store with TTL), or how it's governed (RCM: extension points).

**Co-existence pattern:** Use Reactive Streams for the delivery transport; use RCM semantics for what's delivered and how it's materialized.

**When to use Reactive Streams without RCM:** You're building a low-level streaming library or pipeline that needs backpressure but doesn't need durable, versioned, governed context.

**When to combine:** You're building an RCM implementation and want best-practice flow control; adopt Reactive Streams for the subscription bus.

### **5.1.5 Flux / Redux / MVU / MVI (UI State Management)**

**What it is:** Unidirectional data flow patterns for UI state:

* **Flux:** Actions → Dispatcher → Stores → Views
* **Redux:** Actions → Reducers → Single Store → Components
* **MVU (Model-View-Update):** Messages → Update → Model → View
* **MVI (Model-View-Intent):** Intents → Model → View

All share: events/actions flow into pure functions that produce new state; views render from state; no direct mutation.

**Relationship to RCM:**

* **Same conceptual backbone:** Events → deterministic reduce → state → projection. RCM extends this to durable, multi-session, multi-consumer memory.
* **UI patterns are local:** State lives in browser/app memory; ephemeral across sessions; single consumer (the UI).
* **RCM is durable and shared:** Frames persist beyond sessions; many consumers (agents, services, dashboards) subscribe; lineage and governance apply.

**Co-existence pattern:**

Server (RCM):

Sources → Views → Frames → Subscriptions

Client (Redux/MVU):

Subscribe to frames → Dispatch actions → Reducers → UI renders

Local state for transient UI concerns (hover, focus, form draft)

**When to use Redux/MVU without RCM:** You're building a single-page app with no backend persistence or cross-session memory; all state is local and ephemeral.

**When to add RCM:** You need durable context that survives sessions, serves multiple clients, and requires explainability (lineage, replay). RCM becomes the "server-side Redux" feeding many client-side Reduxes.

### **5.1.6 Blackboard Systems**

**What it is:** A blackboard architecture has a shared "blackboard" where independent specialists read partial information, contribute hypotheses or solutions, and iterate toward a goal. Common in AI planning and collaborative problem-solving.

**Relationship to RCM:**

* **Shared memory role:** Both provide a common substrate for knowledge. Blackboard = workspace for collaborative reasoning; RCM = versioned, reactive memory for distributed systems.
* **RCM adds:** Explicit versioning, lineage, time semantics (watermarks, windows), delivery guarantees, and governance. Blackboard systems often lack these.
* **Blackboard adds:** Opportunistic reasoning, conflict resolution heuristics, goal-driven control. RCM is reactive (source changes trigger updates), not goal-driven.

**Co-existence pattern:** Use RCM frames as the blackboard's knowledge units. Specialists subscribe to frames, reason, and post new frames. The blackboard controller uses frame lineage to explain decisions.

**When to use Blackboard without RCM:** You're building an AI planning system within a single process; durability and distributed delivery aren't concerns.

**When to add RCM:** Your blackboard needs to persist across sessions, serve distributed specialists, provide audit trails, and control resource usage (budgets, quotas).

### **5.1.7 Knowledge Graphs**

**What it is:** A graph-structured knowledge base (RDF, Property Graph) with rich semantics: entities, relationships, attributes, ontologies. Supports traversals, SPARQL/Cypher queries, inference.

**Relationship to RCM:**

* **Different layers:** Knowledge Graphs are a *storage and query substrate*; RCM governs the *reactive lifecycle* of contexts derived from (or feeding into) the graph.
* **RCM can compose over KGs:** A Context View queries the graph, materializes frames with provenance pointing to graph nodes/edges, and delivers to subscribers.
* **Frames can feed KGs:** RCM frames (entities, facts, summaries) can be indexed into a graph for discovery and traversal.

**Co-existence pattern:**

Knowledge Graph (persistent, queryable)

↕

RCM Context Views (reactive queries over graph)

↓

Frames (versioned slices with lineage to graph entities)

↓

Agents, Dashboards (subscribe for fresh graph-derived contexts)

**When to use KG without RCM:** You need semantic search, inference, and traversals but don't need reactive updates or multi-consumer delivery with governance.

**When to add RCM:** You want fresh, versioned slices of graph knowledge delivered reactively to agents/services; you need lineage (which graph nodes contributed to this frame); you want cost control and access policies on graph-derived contexts.

### 

### **5.1.8 Rule Engines (Rete, Drools, Production Systems)**

**What it is:** Rule engines evaluate "if-then" rules over a working memory of facts. They optimize pattern matching (Rete algorithm) and fire rules when conditions match.

**Relationship to RCM:**

* **Frames as working memory:** RCM frames can feed a rule engine's fact base. Frame updates trigger rule re-evaluation.
* **Rules as view logic:** Simple rules (filters, thresholds) can be expressed as RCM view plans. Complex rules (multi-hop inference) fit better in a dedicated engine.
* **Different strengths:** Rule engines excel at inference and complex condition matching; RCM excels at reactive composition, versioning, delivery, and governance.

**Co-existence pattern:**

RCM Frames (facts: user context, recent events, policies)

↓

Rule Engine Working Memory

↓

Rules Fire (produce actions, alerts, decisions)

↓

Actions as new Source Signals → RCM recomputes affected views

**When to use Rule Engine without RCM:** You have complex business logic with many interdependent rules; all facts fit in memory; single-process execution; no need for durable versioning or multi-consumer delivery.

**When to add RCM:** You need to feed rules with fresh, versioned contexts from multiple sources; decisions must be auditable (replay which frames led to which rule firings); rule outputs need governed delivery to downstream systems.

### **5.1.9 Caching Layers (Redis, Memcached, CDN)**

**What it is:** Key-value stores optimized for fast reads, often used to cache results of expensive queries or API calls. Data is ephemeral (can be evicted) and typically lacks structure beyond key-value.

**Relationship to RCM:**

* **Frames can be cached:** Implementations MAY cache frames in Redis/Memcached for low-latency access, but caching is an optimization, not the pattern.
* **RCM is not a cache:** Caches answer "what did I fetch recently?" RCM answers "what do I currently know, with lineage and versioning?"
* **Caches lack:** Reactive updates (RCM recomputes on source change), lineage (RCM records inputs and plan), versioning (RCM provides monotone versions for replay), governance (RCM provides extension points).

**Co-existence pattern:**

RCM Frame Store (durable, versioned, governed)

↓

Cache Layer (ephemeral, fast, key-value)

↓

Consumers (fast reads from cache; fallback to frame store on miss)

**When to use Cache without RCM:** You need fast lookups of simple values (user sessions, rate-limit counters); no need for composition, lineage, or versioning.

**When to add RCM:** You need composite contexts that update reactively, carry lineage for compliance, version for replay, and deliver to many consumers with governance. Cache becomes a read-through layer atop RCM frames.

## **5.2 Decision Heuristics**

This section provides rules of thumb for when to adopt RCM, when to use adjacent patterns, and when to combine them.

### **5.2.1 When to Adopt RCM**

**Choose RCM when:**

* ✅ You need **composite context** from multiple changing sources (not just a single cache or query)
* ✅ **Freshness matters** and you want reactive updates (not hourly batch jobs)
* ✅ **Explainability is required** (compliance, trust, debugging) via lineage and replay
* ✅ **Multiple consumers** (agents, services, dashboards, analytics) need the same context
* ✅ **Cost control** is critical (recompute on change, not on traffic; budgets and quotas)
* ✅ **Governance** (policy, retention, redaction, fairness) must apply uniformly

**Indicators RCM applies:**

* Users or regulators ask "why did the system recommend/decide X?"
* You're building LLM agents that need rich, fresh context across sessions
* You have multiple teams duplicating similar joins and summaries
* Context assembly latency and cost are growing with traffic
* Compliance requires audit trails showing "what we knew when"

### **5.2.2 When RCM May Be Overkill**

**Don't adopt RCM if:**

* ❌ Context is simple (single-source query; no joins or windows)
* ❌ Freshness isn't critical (daily batch updates are fine)
* ❌ Only one consumer exists (no need for fan-out or subscription abstraction)
* ❌ Explainability isn't required (no audits, no compliance, no user-facing explanations)
* ❌ You're just caching API responses (use a cache; RCM is heavier)

**Alternatives to consider:**

* Simple caching (Redis, Memcached) for key-value lookups
* Scheduled ETL for historical reporting
* Direct database queries for low-traffic, low-complexity contexts
* In-process state management (Redux, MVU) for client-side UI state

### **5.2.3 Combining Patterns: Decision Matrix**

|  |  |  |  |
| --- | --- | --- | --- |
| **Your Need** | **Base Pattern** | **Add RCM?** | **Why?** |
| Authoritative event history | Event Sourcing | Yes | RCM builds governed read models atop the event log |
| Read/write separation | CQRS | Yes | RCM implements the read side with reactive updates and delivery |
| Fast precomputed queries | Materialized Views | Yes | RCM standardizes envelopes, lineage, delivery, governance |
| Backpressure for streams | Reactive Streams | Optional | Use RS for transport; RCM for memory lifecycle |
| UI state management | Redux/MVU | Optional | Keep UI state local; use RCM for durable, shared server-side context |
| Collaborative reasoning | Blackboard | Yes | RCM frames become blackboard facts with versioning and lineage |
| Semantic knowledge | Knowledge Graph | Yes | RCM delivers fresh graph slices reactively with provenance |
| Complex business rules | Rule Engine | Yes | RCM frames feed working memory; rule outputs trigger new views |
| Fast key-value lookups | Caching | No (but can co-exist) | Cache RCM frames for speed; RCM handles lifecycle |

## **5.3 Co-Existence Patterns**

This section shows how RCM composes with adjacent patterns in practice.

### **5.3.1 RCM + Event Sourcing (Ledger + Living Memory)**

**Pattern:** Event Sourcing provides the authoritative ledger; RCM builds living, composite memory for operations.

**Architecture:**

Commands → Aggregates → Event Store (append-only log)

↓

RCM Source Signal

↓

Context Views (events ⨝ reference data)

↓

Frames (versioned read models)

↓

Subscriptions (deliver to consumers)

↓

Agents, Dashboards, Search Indices, Analytics

**Benefits:**

* Event Store remains source of truth for "what happened"
* RCM provides "what we know now" with freshness SLOs
* Agents and dashboards consume frames (fast, governed) not raw events (slow, complex joins)
* Audits replay both: events for authority, frames for decisions

**Example:** E-commerce order management

* Events: OrderPlaced, PaymentProcessed, ItemsShipped, OrderCancelled
* RCM View: order\_status = join(order\_events, inventory, shipping\_status)
* Frames: Versioned order context updated reactively; served to customer support agents and dashboard widgets

### **5.3.2 RCM + CQRS (Governed Read Side)**

**Pattern:** CQRS separates writes from reads; RCM implements the read side with reactivity and governance.

**Architecture:**

Write Side:

Commands → Command Handlers → Domain Model → Events → Event Store

Read Side (RCM):

Event Store → RCM Context Views → Frames → Subscriptions → Query Services

**Benefits:**

* Clear separation: writes optimize for consistency and validation; reads optimize for query patterns
* RCM read models update reactively (no polling or manual refresh)
* Multiple read models (different views) serve different consumers from the same events
* Governance (budgets, policies) applies uniformly across all read models

**Example:** Banking application

* Write side: account commands produce transaction events
* RCM read side:
  + View 1: account\_balance (current balance, recent transactions) → agents, mobile app
  + View 2: fraud\_signals (anomaly scores, velocity checks) → fraud detection service
  + View 3: reporting\_snapshot (daily aggregates) → compliance dashboards

### 

### **5.3.3 RCM + Search/Vector Indices (Discoverable Memory)**

**Pattern:** RCM frames feed search and vector indices; search results link back to frame IDs for provenance.

**Architecture:**

RCM Context Views → Frames (entities, facts, summaries)

↓

Indexing Pipeline (curated, redacted)

↓

Search/Vector Index (Elasticsearch, Pinecone, etc.)

↓

Apps, Agents (search for context)

↓

Lookup Frame by ID (for full lineage, replay)

**Benefits:**

* Search provides discovery ("find relevant contexts for query X")
* RCM provides authoritative detail (frame body + lineage)
* Governance applies before indexing (redaction, classification, TTL)
* Index updates reactively as frames change (no stale search results)

**Example:** Knowledge management for support agents

* RCM Views: extract entities, facts, summaries from docs and tickets
* Frames: versioned knowledge units with lineage to source docs
* Index: embeddings + keywords for semantic search
* Agent workflow: search index → retrieve frame IDs → fetch full frames for context assembly

### **5.3.4 RCM + Rule Engines (Explainable Inference)**

**Pattern:** RCM frames populate the rule engine's working memory; rule firings produce new events that trigger view recomputation.

**Architecture:**

RCM Frames (facts: user context, policies, recent events)

↓

Rule Engine Working Memory

↓

Rules Fire (produce decisions, alerts, recommendations)

↓

Decisions as new Source Signals

↓

RCM recomputes affected views (e.g., "user\_risk\_assessment")

↓

New Frames (include rule firing lineage)

↓

Agents, Services (consume updated context)

**Benefits:**

* Rules operate on fresh, versioned facts (RCM frames)
* Rule firings are auditable (frames record which facts led to which rules)
* Decisions feed back into context (closed loop)
* Governance wraps both: frames and rule outputs

**Example:** Fraud detection

* RCM View: transaction\_context = user\_profile ⨝ recent\_txns ⨝ velocity\_stats
* Rule Engine: evaluates fraud rules (velocity thresholds, geo-anomalies, spending patterns)
* Rule fires: produces fraud\_alert event
* RCM recomputes: user\_risk\_assessment view includes the alert
* Frames: delivered to fraud analysts and automated block/review workflows

### **5.3.5 RCM + Edge Protocols (MCP, A2A)**

**Pattern:** RCM governs internal context lifecycle; edge protocols expose/consume context at boundaries.

**Architecture:**

Internal (RCM):

Sources → Views → Frames → Subscriptions → Agents, Services

Edge (MCP Gateway):

External LLM Apps ← MCP Context Requests ← RCM Frames (curated, redacted)

External LLM Apps → MCP Context Pushes → RCM Source Signals (stamped with policy)

Edge (A2A Gateway):

Partner Agents ← A2A Delegation Requests ← RCM Frames (with budget envelopes)

Partner Agents → A2A Results → RCM Source Signals (recorded for lineage)

**Benefits:**

* RCM provides consistent, governed internal context
* MCP/A2A provide interoperability at edges without diluting internal guarantees
* Frames crossing boundaries carry lineage; external systems can verify integrity
* Governance (rate limits, budgets, classification) enforces at gateways

**Example:** Multi-agent collaboration

* Internal agent: uses RCM frames for up-to-date user context
* External partner agent (via A2A): requests context; gateway fetches relevant frames, applies redaction, returns via A2A protocol
* Partner agent processes, returns results
* Results ingested as new Source Signal; RCM recomputes views that depend on partner outputs
* Full audit trail: frames record which partner agents contributed what data

## **5.4 Anti-Patterns and Common Confusions**

This section warns against misunderstandings that lead teams astray.

### **5.4.1 "It's Just Caching"**

**Confusion:** "RCM is a fancy cache. We already have Redis."

**Correction:**

* Caches store results of past fetches; RCM computes composite contexts reactively
* Caches lack lineage (no provenance); RCM frames cite inputs and transforms
* Caches don't version (no replay); RCM frames have monotone versions
* Caches don't govern (no policies or budgets); RCM provides extension points

**When caching suffices:** You're caching simple, single-source queries with no need for freshness guarantees, explainability, or multi-consumer coordination.

**When to adopt RCM:** Context is composite, must stay fresh, requires auditable lineage, and serves multiple consumers with governance.

### **5.4.2 "Just Use Materialized Views"**

**Confusion:** "Our database has materialized views. Why do we need RCM?"

**Correction:**

* Database MVs are single-system optimizations; RCM is a cross-system pattern
* DB MVs lack standardized envelopes (no portable lineage or versioning)
* DB MVs don't provide delivery contracts (at-least-once, per-key ordered fan-out)
* DB MVs don't integrate governance (no budgets, policies, classification)

**When DB MVs suffice:** You're optimizing queries within one database; no cross-team reuse; no compliance/governance requirements.

**When to adopt RCM:** You need to share materialized contexts across many consumers (agents, services, dashboards, analytics), provide lineage for compliance, and govern uniformly.

### **5.4.3 "Our UI State Management Covers This"**

**Confusion:** "We use Redux for state. That's the same thing."

**Correction:**

* Redux is ephemeral, in-process UI state; RCM is durable, cross-session, multi-consumer memory
* Redux state lives in one browser/app; RCM frames are shared across agents, services, dashboards
* Redux doesn't handle time semantics (watermarks, windows); RCM does
* Redux doesn't provide governance, budgets, or retention policies; RCM does

**When Redux suffices:** You're building a single-page app with no backend persistence; all state is local and transient.

**When to adopt RCM:** You need durable context that survives sessions, serves multiple consumers, includes lineage for explainability, and requires governance.

### 

### **5.4.4 "We Log Everything—Good Enough"**

**Confusion:** "We send all events to a log (Kafka, S3). Isn't that our memory?"

**Correction:**

* Logs are raw inputs; RCM provides *composite, versioned contexts*
* Logs don't define "what the system knows now"; they define "what happened"
* Logs don't have delivery contracts (no at-least-once, per-key ordered guarantees to downstream consumers)
* Logs don't include governance (no budgets, classification, redaction)

**When logs suffice:** You're collecting data for future analysis; no need for real-time composition or multi-consumer delivery.

**When to adopt RCM:** You need to compose logs into contexts reactively, serve them with freshness SLOs, version them for replay, and govern access/retention.

### **5.4.5 "Too Complex for Our Needs"**

**Confusion:** "RCM sounds like overkill. We'll just build something simple."

**Pitfalls:**

* "Simple" solutions grow organically into fragile, undocumented messes
* Each team builds their own "simple" version; systems drift
* No shared vocabulary leads to integration nightmares
* Explainability and governance bolted on later are expensive retrofits

**When simplicity is right:** You have one team, one use case, no compliance requirements, and no plans to scale consumers or sources.

**When RCM applies:** You anticipate growth (more teams, more consumers, more sources), need compliance/explainability, or want to avoid technical debt from ad-hoc memory plumbing.

## **5.5 Edge Protocols and Pattern Fit**

RCM operates internally; edge protocols enable interoperability with external systems. This section clarifies how they relate.

### 

### **5.5.1 Model Context Protocol (MCP)**

**What it is:** An open, vendor-neutral protocol for exposing and consuming context and tools between LLM applications and external systems. MCP defines JSON-RPC messages for context requests, tool invocations, and resource discovery.

**Relationship to RCM:**

* **RCM is internal lifecycle; MCP is edge protocol:** RCM governs how context forms, versions, persists internally. MCP governs how context crosses boundaries to/from external LLM apps.
* **Use both:** Run RCM inside your system for governed, versioned frames. Expose curated frames via MCP gateway at edges.
* **Complementary concerns:** MCP standardizes request/response format; RCM standardizes memory lifecycle and governance.

**Integration pattern:**

Internal RCM:

Sources → Views → Frames (versioned, governed)

MCP Gateway:

External LLM App → MCP Context Request → Gateway fetches relevant frames → applies redaction → returns via MCP

External LLM App → MCP Tool Invocation → Gateway logs as Source Signal → RCM recomputes affected views

**Benefits:**

* External systems get fresh context without knowing RCM internals
* Governance (policy, budgets, classification) enforces at gateway
* Lineage preserved: MCP responses can include frame IDs for provenance
* RCM frames can ingest MCP contexts from external systems (with policy stamps)

**When to use MCP without RCM:** You're building a simple integration between two specific systems; no need for multi-consumer memory or governance.

**When to combine:** You have rich, governed internal context (RCM) and need to interoperate with external LLM apps (MCP at edges).

### **5.5.2 Agent-to-Agent (A2A) Protocols**

**What it is:** Emerging standards for multi-agent collaboration: how agents discover capabilities, delegate tasks, negotiate terms, and exchange results. Historical precedent: FIPA-ACL (Agent Communication Language) defined structured messages and performatives for agent communication.

**Relationship to RCM:**

* **RCM provides working memory; A2A provides delegation protocol:** Agents use RCM frames as their internal context; they use A2A messages to request help from other agents.
* **Frames in A2A payloads:** Delegation requests can reference frame IDs for provenance; results can be recorded as new frames with lineage.
* **Governance travels with delegation:** A2A messages can carry budget envelopes (how much the delegating agent is willing to spend) and capability contracts.

**Integration pattern:**

Agent A (Internal RCM):

Uses RCM frame "user\_context\_v42" to assess situation

Determines delegation is needed

Agent A → A2A Delegation Request → Agent B (external or federated domain)

Payload: {task, budget\_envelope, context\_reference: "frameId\_xyz"}

Agent B executes task using its own RCM context

Agent B → A2A Result Response → Agent A

Result: {outcome, cost\_consumed, provenance: "based on frameId\_xyz"}

Agent A's RCM:

Records result as new Source Signal

Recomputes affected views

New frames include lineage: "derived from user\_context\_v42 + Agent B delegation result"

**Benefits:**

* **Explainable delegation chains:** Frames record which agent delegations contributed to decisions
* **Cost control:** Budget envelopes prevent runaway multi-agent spend
* **Cross-agent provenance:** Frame references enable auditing ("Agent B's recommendation used our frame X")
* **Uniform governance:** RCM policies apply to both internal work and delegation responses

**When to use A2A without RCM:** You're building a simple two-agent handoff with stateless interactions; no need for versioned, governed memory or complex context.

**When to combine:** You have multi-agent systems where agents maintain rich internal contexts (RCM), delegate specialized tasks to partners (A2A), and need complete audit trails showing who contributed what knowledge.

### **5.5.3 RCM as Foundation for Edge Interoperability**

**Key insight:** RCM provides the *internal substrate* that makes edge protocols effective and trustworthy.

**Without RCM (ad-hoc approach):**

* Edge protocols expose inconsistent, unversioned contexts
* No lineage to explain what was shared or why
* No governance to control exposure (classification, budgets, redaction)
* Each integration reinvents context assembly
* Trust and auditability are afterthoughts

**With RCM:**

* Edge protocols expose well-formed, versioned, governed frames
* Lineage travels with frames (even across organizational boundaries)
* Governance enforces uniformly (budgets, classification, redaction at gateways)
* Internal improvements (better views, optimized coalescing) automatically benefit all edge integrations
* Consumers can verify frame integrity via hashes and provenance

**Architectural principle:**

┌─────────────────────────────────────────┐

│ External Systems │

│ (LLM Apps, Partner Agents, etc.) │

└──────────────┬──────────────────────────┘

│ MCP / A2A / REST / GraphQL

│

┌────────▼────────┐

│ Edge Gateways │

│ - Policy enforcement

│ - Budget tracking

│ - Redaction

│ - Rate limiting

│ - Protocol translation

└────────┬────────┘

│

┌──────────▼───────────┐

│ Internal RCM Core │

│ - Views

│ - Frames

│ - Subscriptions

│ - Extension points

└──────────┬───────────┘

│

┌──────▼──────┐

│ Sources │

│ (Events, DBs, APIs, Graphs)

└─────────────┘

**Design maxim:** "Keep RCM semantics inside; speak edge protocols at boundaries; enforce governance at gateways."

**Gateway responsibilities:**

1. **Protocol translation:** Convert between RCM frames and edge protocol formats (MCP context payloads, A2A messages, REST JSON)
2. **Policy enforcement:** Check classification, access control, allowed use cases
3. **Redaction:** Apply field-level masking before egress
4. **Budget tracking:** Charge for frame access; enforce quotas
5. **Rate limiting:** Protect internal systems from external traffic spikes
6. **Provenance preservation:** Include frame IDs and lineage summaries in responses
7. **Integrity:** Optionally sign frames for verification

**Example: MCP Gateway Implementation Pattern**

External LLM requests context via MCP:

1. Gateway receives MCP context request

2. Gateway authenticates and authorizes requester

3. Gateway translates MCP query to RCM frame lookup (contextId, key, version?)

4. Gateway fetches relevant frames from internal RCM

5. Gateway applies redaction per classification policy

6. Gateway records budget charge (egress bytes, frame count)

7. Gateway constructs MCP response with curated frame content + frame IDs

8. Gateway returns to external LLM

External LLM can later reference frame IDs in its responses for provenance

## **5.6 Quick Selector (Rule of Thumb)**

This decision tree helps architects choose patterns quickly.

### **Primary Decision Tree**

**Question 1: Do you need a complete, authoritative event history?**

* **Yes** → Start with **Event Sourcing** for write side
  + Need reactive read models? → Add **RCM** for read side
  + Simple queries suffice? → Use direct projections
* **No** → Continue to Question 2

**Question 2: Do you need reactive, composite contexts from multiple sources?**

* **No** (single source, simple query) → Use direct database queries or **simple caching**
* **Yes** → Continue to Question 3

**Question 3: Must contexts be explainable with lineage and replay?**

* **No** → Consider **simple materialized views** or **caching**
* **Yes** → Continue to Question 4

**Question 4: Do multiple heterogeneous consumers need the same contexts?**

* **No** (single consumer, e.g., one dashboard) → Point solution may suffice
* **Yes** → Continue to Question 5

**Question 5: Are freshness SLOs critical (contexts must stay current reactively)?**

* **No** (batch updates acceptable) → **ETL pipelines** may suffice
* **Yes** → **RCM applies**; continue to Question 6

**Question 6: Do you need governance (budgets, policies, classification, retention)?**

* **No** → RCM Core semantics (§§4.2-4.6) may be sufficient
* **Yes** → RCM with **extension points** (§4.7) for governance

### **Situational Selectors**

**If your primary problem is...**

|  |  |  |
| --- | --- | --- |
| **Problem** | **First Choice** | **Add RCM If...** |
| Fast key-value lookups | **Caching** (Redis, Memcached) | Cached values need lineage, versioning, reactive updates |
| Complex business rules | **Rule Engine** (Drools, etc.) | Rules need fresh composite contexts; decisions need audit trails |
| Semantic search | **Knowledge Graph** or **Vector Index** | Need reactive updates to indexed content with provenance |
| UI state management | **Redux/MVU** for local state | Need durable, cross-session, multi-consumer state |
| Read/write separation | **CQRS** | Read models need reactivity, versioning, governance |
| Authoritative ledger | **Event Sourcing** | Need governed read models for multiple consumers |
| Agent context for LLMs | **RCM** | (Primary use case) |
| Multi-agent collaboration | **A2A protocol** | Agents need rich internal memory with explainability |
| Stream processing | **Kafka Streams / Flink** | Need standardized envelopes, delivery contracts, governance |

### 

### **Quick Pattern Compatibility Check**

**Can I use RCM with X?**

* ✅ **Event Sourcing:** Yes, RCM reads events and builds governed views
* ✅ **CQRS:** Yes, RCM implements the read side
* ✅ **Materialized Views:** Yes, RCM standardizes and extends them
* ✅ **Reactive Streams:** Yes, use for delivery transport
* ✅ **Redux/MVU:** Yes, RCM for server state, Redux for UI state
* ✅ **Knowledge Graphs:** Yes, RCM composes views over graph queries
* ✅ **Rule Engines:** Yes, frames feed working memory
* ✅ **Search Indices:** Yes, frames feed indices with provenance
* ✅ **MCP/A2A:** Yes, expose frames via edge protocols
* ⚠️ **Simple Caching:** Can co-exist (cache frames) but RCM is not a cache replacement

## **5.7 Summary: RCM's Place in the Ecosystem**

RCM does not replace adjacent patterns; it **complements and composes** with them.

### **What RCM Is**

**RCM is a pattern for:**

* Reactive composition of sources into versioned, governed contexts
* Deterministic materialization with lineage and time semantics
* Idempotent delivery to many consumers with ordering guarantees
* Extension points for governance (policy, budgets, security)

### **Where RCM Sits**

**RCM sits between:**

* **Upstream (inputs):** Event Sourcing, CQRS write models, databases, APIs, knowledge graphs
* **Downstream (outputs):** Agents, services, dashboards, analytics, search indices

**RCM works alongside:**

* **Reactive Streams** (for flow control and backpressure)
* **Redux/MVU/MVI** (for client-side UI state)
* **Knowledge Graphs** (for semantic storage and queries)
* **Rule Engines** (for inference and complex logic)
* **Search/Vector Indices** (for discovery and retrieval)
* **MCP / A2A** (for edge interoperability)

### **What RCM Is Not**

**RCM is not:**

* ❌ A database, cache, or storage engine (though it uses them)
* ❌ A UI framework (though it feeds them)
* ❌ A rule engine (though it integrates with them)
* ❌ An event log (though it consumes them)
* ❌ A replacement for Event Sourcing, CQRS, or streaming engines
* ❌ A product or framework (it's a pattern)

### **The Value Proposition**

**RCM applies when you need:**

* **Fresh** context that updates reactively on source changes
* **Explainable** context with lineage and replay capability
* **Composite** context from multiple changing sources
* **Shared** context serving multiple heterogeneous consumers
* **Governed** context with policies, budgets, and security controls
* **Auditable** context for compliance and trust

**The litmus test:**

*"Do we need to answer 'what does the system know about X right now, why, and since when?' for multiple consumers, with freshness SLOs and audit trails?"*

* **Yes** → RCM likely applies
* **No** → Simpler patterns (caching, queries, ETL) may suffice

### **Integration Philosophy**

**Best practices for RCM adoption:**

1. **Start internal:** Implement RCM for core contexts (agent memory, operational dashboards)
2. **Layer governance:** Add extension points as operational needs emerge (budgets, policies)
3. **Expose selectively:** Use edge gateways (MCP, A2A) for controlled external access
4. **Compose incrementally:** Integrate with existing Event Sourcing, CQRS, search, rules
5. **Measure value:** Track freshness SLOs, cost reduction, audit coverage, consumer satisfaction

**Common adoption path:**

Phase 1: Core RCM (Weeks 1-8)

- Implement 2-3 critical views

- Emit frames with envelopes

- Switch 1-2 consumers to subscriptions

- Measure freshness, cost, adoption

Phase 2: Governance (Weeks 8-16)

- Add extension points (policy, budgets)

- Implement coalescing and brownout

- Establish freshness SLOs

- Add compaction and TTL policies

Phase 3: Ecosystem (Weeks 16-24)

- Feed search indices from frames

- Integrate with rule engines

- Expose via MCP/A2A at edges

- Dual-run plan changes with confidence

Phase 4: Optimization (Ongoing)

- Tune coalescing windows

- Optimize keys and partitioning

- Implement adaptive TTLs

- Extend to additional contexts

### **Final Positioning**

**Think of RCM as:**

* **REST** is to web APIs
* **Reactive Streams** is to backpressure
* **CQRS** is to read/write separation
* **RCM** is to **context lifecycle**

It defines how composite knowledge forms, versions, persists, and flows—with time, lineage, and governance as first-class concerns.

**When architects understand this positioning, they can:**

* Recognize when RCM solves their problem (not just adds complexity)
* Integrate RCM cleanly with existing patterns (not replace them)
* Adopt incrementally (not require a full rewrite)
* Justify the investment (not chase shiny objects)

**The pattern succeeds when teams say:**

*"We used to rebuild context per-request with no lineage. Now we compose once, deliver to many, replay for audits, and control costs—all with the same substrate."*

# **6. Conclusion**

## **6.1 What RCM Standardizes**

Reactive Composite Memory (RCM) standardizes the **lifecycle and governance of composite context** in intelligent systems. It defines:

**Core semantics:**

* **Portable materialization envelope** (§4.3): Every frame carries identity, version, time window, provenance, determinism keys, and lifecycle metadata in a standardized header
* **Time and ordering** (§4.4): Event-time semantics with watermarks, window closure rules, late-data handling, and per-key delivery ordering
* **Determinism and replay** (§4.5): Plans are referentially transparent; frames have stable idempotency keys; replay by version or time interval is required
* **Delivery contract** (§4.6): At-least-once delivery with per-key ordering, retry semantics, dead-letter queues, and idempotent consumption
* **Extension points** (§4.7): Well-defined hooks for admission control, resource management, security, and observability with required telemetry

**What this enables:**

* **Interoperability:** Multiple implementations can claim conformance and produce/consume compatible frames
* **Explainability:** Every frame cites its sources, transform, and time bounds—auditable by construction
* **Reusability:** One set of frames serves agents, services, dashboards, analytics without duplication
* **Cost control:** Recomputation happens on change (data velocity) not traffic (request volume); budgets bound spend
* **Governance:** Policies (access, retention, classification) apply uniformly via extension points

**What RCM does not standardize:**

* ❌ Implementation technology (Kafka, Flink, custom runtimes all valid)
* ❌ View definition languages (SQL, dataflow DSLs, code—all acceptable if deterministic)
* ❌ Specific governance policies (rate limits, budget models—implementation-defined)
* ❌ Storage backends (SQL, NoSQL, object stores—choice is yours)
* ❌ Programming languages or frameworks

**The pattern is portable:** You can implement RCM on any stack that provides streams, deterministic computation, storage, and delivery primitives.

## **6.2 Conformance Summary**

**To claim RCM conformance, an implementation MUST:**

1. ✓ Implement all normative requirements in sections 4.2 through 4.6
2. ✓ Pass the test vectors specified in Annex A.5
3. ✓ Emit the required telemetry events and metrics defined in Annex A.2
4. ✓ Produce a conformance report in the format specified in Annex A.6

**Conformance class:**

RCM defines **one conformance class** covering reactive composition, versioned frames, time semantics, deterministic materialization, and idempotent delivery. Extension points (§4.7) are **optional but recommended**; if provided, they must emit required telemetry.

**Verification:**

Conformance is demonstrated by:

* Executing the test harness (Annex A.1) in a documented environment
* Passing all required test vectors (Annex A.5)
* Meeting declared SLO targets for freshness, delivery latency, and stability
* Documenting telemetry mappings (if event/metric names differ from spec)

**Conformance report:**

The report (JSON format per Annex A.6) includes:

* Implementation name, version, and contact
* Environment description (nodes, region, hardware)
* Declared SLOs (freshness, latency, DLQ rate, fairness if applicable)
* Test results (pass/fail per vector)
* SLO attainment (percentage of windows meeting targets)
* Telemetry schema mapping
* Extension points documentation (if implemented)

**Public conformance reports** build trust and enable buyers to compare implementations objectively.

## **6.3 Adoption Path**

Adopting RCM does not require a big-bang rewrite. The pattern supports **incremental adoption** alongside existing systems.

### 

### **6.3.1 Pilot Phase (Weeks 1-4)**

**Goal:** Prove value with minimal scope.

**Steps:**

1. **Identify 1-2 high-value contexts** that are painful today (e.g., support agent context, incident hub, personalization tier)
2. **Define Context Views** declaratively (sources, joins, windows)
3. **Implement minimal RCM runtime:**
   * Materializer with event-time and watermarks
   * Frame Store with versioned storage and TTL
   * Subscriptions bus (at-least-once, per-key ordered)
4. **Emit frames with complete envelopes** (headers per §4.3)
5. **Switch one consumer** to subscribe (agent, dashboard panel, or service)
6. **Shadow existing system** (emit frames but don't cut over critical paths yet)

**Success criteria:**

* Frames emit reactively on source changes
* Freshness lag is measurable (watermark\_lag\_ms)
* Consumer receives frames in order
* Lineage is present (inputs, planHash visible)

**Deliverable:** Working prototype serving 1 consumer with 1-2 views; measurable freshness.

### **6.3.2 Production Hardening (Weeks 5-8)**

**Goal:** Make the pilot production-ready.

**Steps:**

1. **Add coalescing** (tens of ms) to absorb microbursts
2. **Implement compaction:** Periodic snapshots + delta folding
3. **Add retry and DLQ** for delivery failures
4. **Establish freshness SLO** (e.g., p95 watermark\_lag\_ms < 2s)
5. **Implement brownout mode** (widen coalescing under backlog)
6. **Add extension hooks** (policy checks, basic rate gates, budget tracking if needed)
7. **Build replay harness** with 1-2 golden vectors

**Success criteria:**

* SLO met in ≥95% of windows
* DLQ handles failures gracefully
* Replay produces identical idempotencyKeys
* Telemetry dashboards show freshness, latency, cost

**Deliverable:** Production-grade RCM serving 1 consumer with governance basics.

### **6.3.3 Expansion (Weeks 9-16)**

**Goal:** Broaden adoption across teams and use cases.

**Steps:**

1. **Add 2-3 more consumers** to the same frames (dashboard, analytics, search indexer)
2. **Define 3-5 additional views** (hierarchical: build on existing views)
3. **Cut over critical consumers** (agents, operational dashboards) from ad-hoc assembly to RCM subscriptions
4. **Feed search/vector indices** from frames (with redaction)
5. **Integrate with existing systems:**
   * Event Sourcing: RCM reads event log
   * CQRS: RCM implements read side
   * Knowledge Graph: RCM composes over graph queries
6. **Document extension patterns** (governance chains, budget models) for your org
7. **Run dual-run tests** for plan changes (compare planHash, validate diffs)

**Success criteria:**

* Multiple consumers share frames (no duplicate pipelines)
* Cost reduced (recompute on change, not per consumer)
* Freshness SLOs hold under increased load
* Auditors can replay frames for compliance checks

**Deliverable:** RCM serving 5+ consumers across 5+ views; measurable cost savings and audit coverage.

### 

### **6.3.4 Ecosystem Maturity (Weeks 16+)**

**Goal:** Establish RCM as the canonical memory plane.

**Steps:**

1. **Federate across domains** (if multi-team): Each domain runs RCM; gateways enforce policy at boundaries
2. **Expose at edges** (MCP, A2A, REST): Controlled external access via gateways with redaction and budgets
3. **Optimize continuously:**
   * Tune coalescing windows per view
   * Optimize keys for natural partitioning
   * Implement adaptive TTLs (heat-based retention)
4. **Contribute conformance vectors** to open test suite (if adopting community spec)
5. **Mentor other teams** in RCM adoption (share learnings, anti-patterns)
6. **Measure business impact:**
   * Faster feature velocity (new views ship declaratively)
   * Reduced incidents (auditable context for RCA)
   * Compliance wins (regulator reviews lineage, TTL policies)

**Success criteria:**

* RCM is default pattern for new memory features
* Cross-team reuse is common (shared views)
* Cost per frame is tracked and predictable
* Conformance report published (internal or public)

**Deliverable:** Mature RCM ecosystem with federated domains, edge interop, and organizational buy-in.

### 

### **6.3.5 Adoption Anti-Patterns to Avoid**

**Don't:**

* ❌ **Boil the ocean:** Start with 20 views; overwhelm the team; fail to deliver value
* ❌ **Skip governance:** Ignore extension points; face cost surprises and policy violations later
* ❌ **Ignore SLOs:** Run without freshness targets; discover staleness in production
* ❌ **Neglect replay:** Skip golden vectors; lose confidence in determinism when it matters
* ❌ **Force retrofit:** Mandate RCM for all contexts immediately; create resentment and technical debt

**Do:**

* ✅ **Start small:** 1-2 views, 1 consumer, prove value
* ✅ **Measure always:** Freshness, cost, adoption metrics from day one
* ✅ **Layer governance:** Start with Core (§§4.2-4.6), add extensions (§4.7) as needs emerge
* ✅ **Build trust:** Demonstrate replay works; show lineage to stakeholders; meet SLOs consistently
* ✅ **Adopt voluntarily:** Let teams choose RCM when it solves their pain; don't mandate top-down

## **6.4 Interoperability and Ecosystem**

RCM's value multiplies when multiple implementations exist and interoperate.

### **6.4.1 Conformance Enables Choice**

**For buyers:**

* Compare implementations objectively via conformance reports
* Switch vendors without re-architecting (frames are portable)
* Mix implementations (e.g., Kafka-based for one domain, custom runtime for another)

**For implementers:**

* Differentiate on performance, UX, ecosystem integrations
* Claim standards compliance (credibility signal)
* Contribute to shared test suite (reduce conformance burden)

### 

### **6.4.2 Open Specification Development**

This specification is developed as a **Community Specification** under open governance:

* **Repository:** github.com/critical-insight/rcm-spec (hypothetical; adjust to actual repo)
* **License:** CC-BY-4.0 (spec text), Apache-2.0 (test code)
* **Participation:** Open to all via GitHub issues, pull requests, discussions
* **Governance:** Maintainer(s) review contributions; rough consensus for changes; versioned releases

**How to participate:**

1. **Feedback:** Open issues for clarifications, ambiguities, suggestions
2. **Test vectors:** Contribute golden replay vectors (Annex A.5) covering edge cases
3. **Crosswalks:** Document mappings to other patterns, frameworks (expand Annex B)
4. **Implementations:** Build conformant systems; publish conformance reports
5. **Adoption stories:** Share scenarios, metrics, lessons learned

### **6.4.3 Ecosystem Building**

**Current state (v1.0):**

* Specification published
* Conformance test kit available (Annex A defines requirements; implementation needed)
* Reference implementations in progress (CognOS, Kafka-based)

**Near-term goals (6-12 months):**

* 3+ independent implementations claim conformance
* Public conformance reports from 2+ vendors
* Community test vectors cover 90% of edge cases
* Crosswalks to major AI frameworks (LangChain, Semantic Kernel, LlamaIndex)

**Long-term vision (12-24 months):**

* RCM adopted in enterprise AI platforms (Databricks, Snowflake, AWS, Azure, GCP)
* Standards bodies (OASIS, CNCF, Linux Foundation) recognize RCM
* Edge protocols (MCP, A2A) reference RCM semantics
* Academic papers cite RCM for AI memory architectures

### 

### **6.4.4 Compatibility with Existing Standards**

RCM is designed to **complement** (not compete with) existing standards:

* **W3C PROV:** RCM frames can carry PROV-DM provenance in extensions; inputs field is PROV-compatible
* **Reactive Streams:** RCM subscriptions can use Reactive Streams transport for backpressure
* **OpenTelemetry:** RCM telemetry (Annex A.2) maps cleanly to OTel events and metrics
* **CloudEvents:** RCM frames can be wrapped in CloudEvents envelopes for transport interop
* **JSON Schema / Avro / Protobuf:** Frame bodies and headers can use any schema; implementations document their choice

**Design principle:** RCM specifies *semantics*, not *encodings*. Implementations choose serialization formats that fit their ecosystems.

## **6.5 How to Participate**

### **6.5.1 For Architects and Decision-Makers**

**Evaluate RCM:**

1. Read Chapters 1-3 (problem, picture, overview)
2. Read Chapter 5 (comparative guide) to understand fit
3. Review Annex A (conformance) to assess rigor
4. Check for existing implementations in your ecosystem

**Adopt RCM:**

1. Follow adoption path (§6.3): pilot → harden → expand
2. Measure value: freshness SLOs, cost reduction, audit coverage
3. Share learnings: blog posts, conference talks, GitHub discussions

**Influence the spec:**

1. Open issues for ambiguities or gaps
2. Propose refinements via pull requests
3. Contribute real-world scenarios to Chapter 10 (if accepted)

### 

### **6.5.2 For Implementers**

**Build conformant systems:**

1. Implement normative requirements (§§4.2-4.6)
2. Run conformance tests (Annex A.5)
3. Emit required telemetry (Annex A.2)
4. Produce conformance report (Annex A.6)
5. Publish report (GitHub, website, or vendor docs)

**Contribute to test suite:**

1. Add golden replay vectors for edge cases
2. Document test harness improvements
3. Share performance benchmarks

**Integrate with ecosystem:**

1. Build bridges to adjacent patterns (Event Sourcing, CQRS, search, rules)
2. Implement edge protocol gateways (MCP, A2A)
3. Document integration patterns (blog, docs, talks)

### **6.5.3 For Researchers**

**Extend the theory:**

1. Formalize temporal correctness proofs for window closure and late-data handling
2. Explore adaptive algorithms (watermark advancement, coalescing, TTL optimization)
3. Model fairness and starvation bounds for weighted fair queueing with aging
4. Develop cost models for RCM vs. per-request assembly (amortization analysis)
5. Study memory consolidation (cognitive analogies from Chapter 8)

**Publish findings:**

1. Cite this specification (DOI or persistent URL, once assigned)
2. Share datasets and benchmarks (open science)
3. Propose spec extensions via academic-to-industry feedback loops

### 

### **6.5.4 For Standards Bodies**

**If RCM gains traction and multiple implementations exist, we invite standards organizations to:**

* Adopt RCM as a work item (OASIS TC, CNCF project, etc.)
* Provide governance and IP protections
* Fast-track to ISO/IEC if enterprise adoption warrants
* Co-develop conformance badging programs

**Current status:** Community Specification (open governance, no formal SDO affiliation yet).

**Path to standardization:** Demonstrate 3+ independent implementations, public conformance reports, and cross-vendor interoperability; then approach OASIS or CNCF.

## **6.6 Acknowledgments**

This specification draws on decades of distributed systems research and practice:

**Stream processing and time semantics:**

* Tyler Akidau and the Dataflow Model team (Google/Apache Beam) for watermarks, windows, and event-time semantics
* The Apache Flink, Kafka Streams, and Pulsar communities for streaming best practices

**Reactive architectures:**

* The Reactive Streams working group for backpressure protocols
* Evan Czaplicki (Elm), Dan Abramov (Redux), and André Staltz (Cycle.js) for unidirectional flow patterns

**Event-driven patterns:**

* Martin Fowler and Greg Young for Event Sourcing and CQRS
* Pat Helland for work on immutability and append-only architectures

**Cognitive science:**

* William James for stream of consciousness
* Richard Atkinson and Richard Shiffrin for the modal model of memory
* Alan Baddeley and Graham Hitch for working memory theory
* Endel Tulving for episodic and semantic memory distinctions

**AI and knowledge representation:**

* Marvin Minsky for frames and knowledge structures
* The blackboard model community (Nii, Erman, et al.)
* Knowledge graph researchers (Hogan et al.)

**Interoperability and edge protocols:**

* Anthropic for Model Context Protocol (MCP)
* The agent communication community (FIPA-ACL and successors)

**Special thanks:**

* Early adopters and implementers who tested these ideas in production
* Reviewers who provided feedback on drafts (to be named as they contribute)
* The open-source and open-standards communities for making portable patterns possible

## **6.7 Closing Thoughts**

Reactive Composite Memory is not a revolutionary idea—it's a **methodical synthesis** of proven patterns (materialized views, event-time processing, unidirectional flow, provenance) applied to the urgent problem of context management in intelligent systems.

**The pattern succeeds when:**

* Teams stop rebuilding context per-request and start composing reactively
* Decisions become auditable because frames carry lineage by construction
* Costs become predictable because recomputation follows change, not traffic
* Governance becomes uniform because extension points apply policies consistently
* Interoperability becomes real because frames are portable across implementations

**The pattern fails when:**

* Adopted for the wrong reasons (chasing trends, not solving pain)
* Over-engineered (implementing all of Chapter 4 when caching would suffice)
* Under-governed (skipping extension points, facing cost surprises later)
* Isolated (not integrating with existing Event Sourcing, CQRS, search, rules)

**Our hope:**

That RCM becomes a **shared vocabulary** for architects, a **reusable substrate** for implementers, and a **trustworthy foundation** for intelligent systems that remember reliably, explain credibly, and scale economically.

**Your move:**

Read. Evaluate. Pilot. Share learnings. Contribute. Build the ecosystem.

Welcome to Reactive Composite Memory.

# **Annex A (Normative): Conformance and Testing**

## **A.1 Test Harness Model**

### **A.1.1 Purpose and Scope**

This annex defines the test procedures, required telemetry, SLO measurement rules, and reporting format necessary to demonstrate RCM conformance. All implementations claiming RCM conformance MUST execute these tests and produce results in the specified format.

### **A.1.2 Test Environment Requirements**

**Isolation:**

* Tests MUST run in an isolated environment (dedicated test cluster or containers)
* Tests MUST NOT interfere with production workloads
* Test data MUST be synthetic (generated or anonymized)
* Random seeds MUST be fixed for reproducibility

**Documentation:**

* Environment specifications MUST be documented: node count, CPU/memory per node, network topology, storage type
* Software versions MUST be recorded: runtime, libraries, operating system
* Test execution date and duration MUST be recorded

**Repeatability:**

* Tests MUST be repeatable: running the same test twice produces the same results (within measurement tolerance for timing metrics)
* Non-determinism sources (wall-clock, random) MUST be controlled or eliminated

### 

### **A.1.3 Required Test Suites**

Implementations MUST execute the following test suites and report pass/fail for each:

**Suite 1: Determinism and Idempotency**

* **DV-1:** Deterministic View Evaluation (§A.5.1)
* **ID-1:** Idempotent Delivery (§A.5.2)

**Suite 2: Time and Windows**

* **TM-1:** Watermark Monotonicity (§A.5.3)
* **TM-2:** Window Closure Timing (§A.5.4)
* **TM-3:** Late Data Handling (§A.5.5)

**Suite 3: Ordering and Delivery**

* **OR-1:** Ordering Under Retry (§A.5.6)
* **OR-2:** Per-Key Isolation (§A.5.7)

**Suite 4: Replay**

* **RP-1:** Replay by Version (§A.5.8)
* **RP-2:** Replay by Time Interval (§A.5.9)

**Suite 5: Extension Points (Optional, if implemented)**

* **EX-1:** Admission Decision Telemetry (§A.5.10)
* **EX-2:** Budget Lease and Commit (§A.5.11)
* **EX-3:** Redaction Application (§A.5.12)

## **A.2 Required Events and Metrics**

### **A.2.1 Event Semantics**

Implementations MUST emit structured events documenting key lifecycle transitions. Event names shown here are **normative for semantics**; exact spelling MAY vary if the mapping is documented in the conformance report.

**Event naming convention:**

* Use dot-notation: category.action (e.g., frame.materialized)
* Keep names lowercase with underscores for multi-word terms
* Include required labels as key-value pairs

**Delivery:**

* Events MUST be emitted to a durable, queryable log (stdout, structured logging system, telemetry backend)
* Events MUST be timestamped (RFC 3339 or Unix timestamp with millisecond precision)
* Events MUST NOT contain PII in labels or free-text fields

### **A.2.2 Required Events (Core)**

**frame.materialized**

Emitted when a frame is successfully computed and stored.

Labels:

* contextId (string): Context view identifier
* key (string): Partition/correlation key
* version (integer): Frame version
* type (enum: "snapshot" | "delta"): Frame type
* window.start (RFC 3339): Event-time window start
* window.end (RFC 3339): Event-time window end
* watermarkAt (RFC 3339): Watermark at emission
* planHash (string): Transform identity hash
* idempotencyKey (string): Stable deduplication key
* ttl (ISO 8601 duration): Time-to-live
* frameSizeBytes (integer): Total frame size (headers + body)

**view.window.closed**

Emitted when a window transitions from Accepting to Closing state.

Labels:

* contextId (string)
* key (string)
* window.start (RFC 3339)
* window.end (RFC 3339)
* watermark (RFC 3339): Watermark that triggered closure
* policy (string): Window policy name (e.g., "tumbling-1m", "session-5m-gap")

**delivery.attempt**

Emitted for each delivery attempt to a subscriber.

Labels:

* contextId (string)
* key (string)
* version (integer)
* frameId (string)
* subscriberId (string): Subscriber identifier
* attemptNumber (integer): 1 for first attempt, increments on retry

**delivery.retry**

Emitted when a delivery fails and will be retried.

Labels:

* contextId (string)
* key (string)
* version (integer)
* subscriberId (string)
* attemptNumber (integer)
* reason (string): Failure reason (exception class, status code)
* nextRetryDelayMs (integer): Backoff delay before next attempt

**delivery.dlq**

Emitted when a frame is routed to the dead-letter queue after exhausting retries.

Labels:

* contextId (string)
* key (string)
* version (integer)
* frameId (string)
* subscriberId (string)
* reason (string): Terminal failure reason
* totalAttempts (integer): Number of delivery attempts made

**duplicate.detected**

Emitted when duplicate work is detected via idempotencyKey matching.

Labels:

* contextId (string)
* key (string)
* idempotencyKey (string)
* stage (enum: "materialization" | "delivery"): Where duplicate was detected

### **A.2.3 Required Events (Extensions, if implemented)**

**admission.decision**

Emitted when an extension hook makes an admit/deny/defer decision.

Labels:

* stage (enum: "pre-materialize" | "pre-publish" | "delivery"): Hook location
* contextId (string)
* key (string)
* verdict (enum: "admit" | "deny" | "defer"): Decision outcome
* reasonCode (string): Implementation-defined reason (e.g., "rate\_limit\_exceeded", "policy\_denied", "budget\_unavailable")

**budget.lease**

Emitted when budget units are reserved.

Labels:

* contextId (string)
* key (string)
* leaseId (string): Unique lease identifier
* units.timeMs (integer, optional): Time budget in milliseconds
* units.moneyCents (integer, optional): Cost budget in cents
* units.tokens (integer, optional): Token budget (e.g., LLM tokens)
* units.netBytes (integer, optional): Network egress budget in bytes
* units.rows (integer, optional): Row/record processing budget
* ttlMs (integer): Lease expiration time

**budget.commit**

Emitted when actual budget usage is recorded after successful completion.

Labels:

* leaseId (string)
* actual.timeMs (integer, optional): Actual time consumed
* actual.moneyCents (integer, optional): Actual cost
* actual.tokens (integer, optional): Actual tokens consumed
* actual.netBytes (integer, optional): Actual network egress
* actual.rows (integer, optional): Actual rows processed

**budget.cancel**

Emitted when a lease is released without completion (failure, denial, timeout).

Labels:

* leaseId (string)
* reason (string): Why lease was cancelled

**security.classification**

Emitted when a frame is classified.

Labels:

* contextId (string)
* key (string)
* frameId (string)
* classification (string): Implementation-defined taxonomy (e.g., "public", "internal", "confidential", "restricted")

**security.redaction**

Emitted when redaction is applied to a frame.

Labels:

* contextId (string)
* key (string)
* frameId (string)
* pathsRedacted (array of strings): JSON pointers or field paths that were masked/removed
* redactionCount (integer): Number of fields redacted

**human.approval.requested**

Emitted when a frame requires human approval before exposure.

Labels:

* contextId (string)
* key (string)
* frameId (string)
* approvalId (string): Unique approval request identifier
* ttlMs (integer): Time window for human response

**human.approval.granted** / **human.approval.denied**

Emitted when human makes approval decision.

Labels:

* approvalId (string)
* contextId (string)
* key (string)
* frameId (string)
* approver (string): Identity of approver (non-PII identifier)
* reason (string, optional): Explanation for decision

### 

### **A.2.4 Required Metrics (Core)**

Implementations MUST maintain the following metrics. Metric names are **normative for semantics**; exact spelling MAY vary if documented.

**Naming conventions:**

* Use snake\_case with units as suffixes: metric\_name\_unit
* Common units: ms (milliseconds), bytes, count, ratio (0-1)
* Use histogram for latency and size distributions; gauge for current state; counter for cumulative counts

**Metric Types:**

* **Histogram:** Records distribution of values (e.g., latency); MUST support percentile queries (p50, p95, p99)
* **Gauge:** Records current value (e.g., queue depth)
* **Counter:** Records cumulative count (monotonically increasing)

**Core Metrics:**

**watermark\_lag\_ms** (histogram)

Lag between current watermark and maximum observed event-time, per (contextId, key).

Calculation: emit\_watermark - max(event\_time\_seen\_in\_window)

Labels: contextId, key

**recompute\_latency\_ms** (histogram)

Time to evaluate view plan and produce frame body, measured from plan invocation to completion.

Labels: contextId

**delivery\_latency\_ms** (histogram)

End-to-end time from frame emission to subscriber acknowledgment.

Calculation: subscriber\_ack\_time - frame\_createdAt

Labels: contextId, subscriberId

**frame\_size\_bytes** (histogram)

Total size of frame (headers + body) in bytes.

Labels: contextId, type (snapshot|delta)

**queue\_depth** (gauge)

Number of frames pending delivery per subscriber, sampled periodically.

Labels: subscriberId

**queue\_wait\_ms** (histogram)

Time a frame spends in delivery queue before being sent to subscriber.

Labels: subscriberId, priority (if priority scheduling implemented)

**concurrency\_utilization** (gauge)

Ratio of in-flight materializations to maximum concurrency limit, sampled periodically.

Calculation: active\_materializations / max\_concurrency

Labels: none (global) or contextId (per-view)

**delivery\_retry\_count** (counter)

Cumulative count of delivery retries.

Labels: contextId, subscriberId, reason

**delivery\_dlq\_count** (counter)

Cumulative count of frames routed to DLQ.

Labels: contextId, subscriberId, reason

**duplicate\_dedup\_count** (counter)

Cumulative count of duplicate work detected and skipped.

Labels: contextId, stage (materialization|delivery)

### 

### **A.2.5 Required Metrics (Extensions, if implemented)**

**admission\_decision\_count** (counter)

Count of admission decisions by verdict.

Labels: stage, verdict, reasonCode

**budget\_lease\_count** (counter)

Count of budget leases requested.

Labels: contextId, unit (timeMs|moneyCents|tokens|netBytes|rows)

**budget\_commit\_count** (counter)

Count of budget commits (successful work).

Labels: contextId, unit

**budget\_timeMs\_committed** / **budget\_moneyCents\_committed** / **budget\_tokens\_committed** / **budget\_netBytes\_committed** / **budget\_rows\_committed** (counter)

Cumulative budget units consumed.

Labels: contextId, key (optional, for per-key attribution)

**security\_redaction\_count** (counter)

Count of redaction applications.

Labels: contextId, classification

**policy\_denied\_count** (counter)

Count of policy denials.

Labels: stage, reasonCode

## 

## **A.3 Measurement Rules and SLO Accounting**

### **A.3.1 Measurement Windows**

**Requirement:** Implementations MUST compute SLO metrics over sliding windows.

**Window parameters:**

* **Window length:** MUST be declared in the conformance report (recommended: 60 minutes)
* **Step width:** MUST be ≤ 1 minute (how often windows advance)
* **Example:** 60-minute window advancing every 1 minute produces 60 overlapping windows per hour

**Purpose:** Sliding windows smooth out transient spikes and provide statistically meaningful SLO evaluation.

### **A.3.2 Percentile Estimation**

**Requirement:** Implementations MUST estimate percentiles (p50, p95, p99) using a deterministic algorithm.

**Acceptable methods:**

* **CKMS (Cormode-Korn-Muthukrishnan-Srivastava):** Quantile sketch with error bounds
* **HDR Histogram:** Fixed-precision histogram with configurable resolution
* **T-Digest:** Mergeable quantile sketch

**Not acceptable:**

* Reservoir sampling (non-deterministic without fixed seed)
* Exact sorting (doesn't scale)

**Error tolerance:**

* p95/p99 estimates MUST be within ±2% of true value (validated via synthetic workloads with known distributions)

### 

### **A.3.3 Freshness SLO Calculation**

**Metric:** watermark\_lag\_ms per (contextId, key)

**SLO statement:** "p95 of watermark\_lag\_ms MUST be ≤ TARGET in ≥ X% of measurement windows"

**Calculation procedure:**

1. For each measurement window W:  
   * Collect all watermark\_lag\_ms samples in W
   * Compute p95 of those samples
   * Compare to declared TARGET
   * Record: window\_meets\_slo = (p95 <= TARGET)
2. Over all windows in the test period:  
   * Compute: attainment\_ratio = count(window\_meets\_slo=true) / total\_windows
   * SLO is met if: attainment\_ratio >= X

**Conformance requirement:**

* For p95 target: attainment\_ratio MUST be ≥ 0.95 (95% of windows)
* For p99 target: attainment\_ratio MUST be ≥ 0.90 (90% of windows)

**Example:**

* Target: p95 watermark\_lag\_ms ≤ 2000ms
* Test period: 8 hours (480 windows of 60 minutes, stepping every 1 minute)
* Result: 462 windows meet target
* Attainment: 462/480 = 96.25% → **PASS** (≥ 95%)

### **A.3.4 Delivery Latency SLO Calculation**

**Metric:** delivery\_latency\_ms per (contextId, subscriberId)

**SLO statement:** "p95 of delivery\_latency\_ms MUST be ≤ TARGET in ≥ 95% of measurement windows"

**Calculation procedure:** Same as freshness (§A.3.3), applied to delivery\_latency\_ms samples.

### 

### **A.3.5 Stability SLO Calculation**

**Metrics:**

* delivery\_retry\_count
* delivery\_dlq\_count

**SLO statements:**

* "Retry rate MUST be ≤ TARGET per 10,000 deliveries in ≥ 99% of windows"
* "DLQ rate MUST be ≤ TARGET per 10,000 deliveries in ≥ 99% of windows"

**Calculation procedure:**

1. For each measurement window W:  
   * Sum: retry\_count = Σ delivery\_retry\_count in W
   * Sum: delivery\_attempts = Σ delivery\_attempt events in W
   * Compute: retry\_rate = (retry\_count / delivery\_attempts) \* 10000
   * Compare to TARGET
   * Record: window\_meets\_slo = (retry\_rate <= TARGET)
2. Compute attainment\_ratio over all windows
3. SLO is met if attainment\_ratio ≥ 0.99

**Same procedure for DLQ rate.**

### **A.3.6 Fairness SLO Calculation (Extensions, if implemented)**

**Requirement:** Implementations claiming fairness MUST demonstrate weighted share allocation.

**Setup:**

* Configure N tenants/priorities with declared weights: {w1, w2, ..., wN}
* Saturate system with equal upstream demand from all tenants
* Measure delivered frames per tenant over measurement windows

**Calculation procedure:**

1. For each measurement window W:  
   * Count: delivered[i] = frames delivered to tenant i in W
   * Compute: total\_delivered = Σ delivered[i]
   * Compute observed share: observed[i] = delivered[i] / total\_delivered
   * Compute expected share: expected[i] = w[i] / Σ w[j]
   * Compute deviation: deviation[i] = |observed[i] - expected[i]|
   * Record: max\_deviation = max(deviation[i])
2. Over all windows:  
   * SLO is met if: max\_deviation <= declared\_bound in ≥ 95% of windows

**Declared bound:**

* Implementations MUST declare their fairness bound (e.g., ±15%)
* Bound MUST NOT exceed ±20% unless justified and documented

**Aging requirement:**

* Long-queued work MUST progress (no permanent starvation)
* Test: Introduce one tenant with weight 0.01 among tenants with weight 1.0
* Measure: time until minority tenant receives first frame
* Requirement: MUST receive frame within 10x average frame emission interval

### **A.3.7 Budget SLO Calculation (Extensions, if implemented)**

**Metrics:**

* budget.lease, budget.commit, budget.cancel events
* budget\_\*\_committed counters

**Requirements:**

1. **No double-spend:** For each leaseId, exactly one of budget.commit or budget.cancel MUST be observed within lease TTL
2. **Leak detection:** Sum of committed + cancelled MUST equal lease count (within ±1% tolerance for timing skew)
3. **Attribution accuracy:** Per-context budget totals MUST be accurate within ±5%

**Test procedure:**

1. Run workload with budget tracking enabled
2. Collect all budget events
3. Group by leaseId; verify one-and-only-one commit or cancel per lease
4. Sum committed units per context; compare to expected (calculated from frame sizes, plan costs)
5. Report any leaks (leases with no commit or cancel) as failures

## **A.4 SLO Obligations**

### **A.4.1 Core SLOs (All Implementations)**

Implementations MUST declare SLO targets for the following and demonstrate attainment per §A.3.

**Freshness:**

* Metric: watermark\_lag\_ms
* Targets: p95 and p99 in milliseconds
* Attainment: p95 target met in ≥95% of windows; p99 target met in ≥90% of windows

**Delivery Latency:**

* Metric: delivery\_latency\_ms
* Target: p95 in milliseconds
* Attainment: p95 target met in ≥95% of windows

**Stability (DLQ Rate):**

* Metric: delivery\_dlq\_count / delivery\_attempts \* 10000
* Target: Maximum DLQ rate per 10,000 deliveries
* Attainment: Target met in ≥99% of windows

**Ordering:**

* Requirement: Zero violations of per-key ordering in test suite
* Test: OR-1, OR-2 (§A.5.6, §A.5.7) MUST pass

### 

### **A.4.2 Extension SLOs (If Implemented)**

**Fairness (if weighted scheduling implemented):**

* Metric: max\_deviation from expected weighted share
* Target: Declared fairness bound (e.g., ±15%)
* Attainment: Target met in ≥95% of windows

**Budget (if budget tracking implemented):**

* Requirement: Zero double-spends; ≤1% lease leaks; ≤5% attribution error
* Test: EX-2 (§A.5.11) MUST pass

**Redaction (if security extensions implemented):**

* Requirement: 100% redaction compliance where classification dictates
* Test: EX-3 (§A.5.12) MUST pass

### **A.4.3 Reference SLO Profiles (Optional Badges)**

Implementations MAY claim optional reference profiles if they meet all profile requirements in the declared environment.

**Profile R0 (Batch-Friendly):**

* Freshness p95: ≤ 60,000ms (60 seconds)
* Delivery p95: ≤ 2,000ms (2 seconds)
* DLQ rate: ≤ 5 per 10,000 deliveries
* Fairness deviation: ≤ ±20% (if applicable)
* Use case: Analytical workloads, batch reporting

**Profile R1 (Regional-Interactive):**

* Freshness p95: ≤ 2,000ms (2 seconds)
* Delivery p95: ≤ 500ms
* DLQ rate: ≤ 1 per 10,000 deliveries
* Fairness deviation: ≤ ±15%
* Use case: Operational dashboards, agent context

**Profile R2 (Low-Latency):**

* Freshness p95: ≤ 500ms
* Delivery p95: ≤ 150ms
* DLQ rate: ≤ 0.5 per 10,000 deliveries
* Fairness deviation: ≤ ±10%
* Use case: Real-time decision systems, high-frequency trading

**Note:** Profiles are aspirational; implementations claim profiles only if all requirements are met in representative workloads.

## **A.5 Required Test Vectors**

This section specifies the test vectors that implementations MUST pass to claim conformance.

### **A.5.1 DV-1: Deterministic View Evaluation**

**Purpose:** Verify that identical inputs produce identical outputs.

**Setup:**

* Define a Context View with at least one join and one aggregation
* Prepare fixed input dataset: 100 events across 3 sources
* Fix all parameters (window size, keys, plan configuration)
* Compute planHash

**Procedure:**

1. Evaluate the view plan over the inputs (first run)
2. Record: frame1.body, frame1.idempotencyKey
3. Evaluate the view plan over the same inputs (second run)
4. Record: frame2.body, frame2.idempotencyKey
5. Repeat N=5 total times

**Pass Criteria:**

* All frame bodies MUST be byte-identical
* All idempotencyKeys MUST be identical
* No variation across runs

**Failure modes:**

* Non-deterministic operators (random, wall-clock, unordered hash maps)
* Unstable serialization
* planHash doesn't capture all dependencies

### 

### **A.5.2 ID-1: Idempotent Delivery**

**Purpose:** Verify that duplicate deliveries produce effect-once semantics.

**Setup:**

* Emit one frame to a test subscriber
* Inject duplicate deliveries: deliver the same frame M=5 times (same frameId, same idempotencyKey)
* Subscriber counts effects (e.g., writes to a database, increments a counter)

**Procedure:**

1. Subscriber receives frame (first delivery)
2. Subscriber processes and applies effect (effect count = 1)
3. Subscriber receives frame again (duplicate #1)
4. Subscriber deduplicates via idempotencyKey; no new effect (effect count still = 1)
5. Repeat for M-1 more duplicates

**Pass Criteria:**

* Effect applied exactly once (effect count = 1) despite M deliveries
* Subscriber acknowledges all deliveries (no errors)

**Failure modes:**

* Subscriber doesn't deduplicate (effect count = M)
* Subscriber crashes on duplicate (errors logged)

### **A.5.3 TM-1: Watermark Monotonicity**

**Purpose:** Verify that watermarks never decrease for a given (contextId, key).

**Setup:**

* Define a Context View over an event stream
* Inject 100 events with out-of-order timestamps (random shuffle)
* Track watermark emissions

**Procedure:**

1. Feed events one by one to the Materializer
2. After each event, record current watermark for the key
3. Build sequence: [wm\_0, wm\_1, ..., wm\_100]

**Pass Criteria:**

* For all i: wm\_{i+1} >= wm\_i (monotonically non-decreasing)
* No watermark reversals

**Failure modes:**

* Watermark decreases due to late event processing
* Incorrect watermark computation

### **A.5.4 TM-2: Window Closure Timing**

**Purpose:** Verify that frames are not emitted before window closure.

**Setup:**

* Define tumbling windows of 60 seconds
* Inject events spanning 3 windows: [0-60s), [60-120s), [120-180s)
* Watermark advances manually (controlled for test)

**Procedure:**

1. Inject all events for window [0-60s) with event-times in that range
2. Advance watermark to 59s → no frame emitted (window not closed)
3. Advance watermark to 60s → frame for [0-60s) MUST emit (window closed)
4. Repeat for subsequent windows

**Pass Criteria:**

* Frames emitted only after watermark >= window.end
* No early emissions

**Failure modes:**

* Frame emitted before watermark passes window end (processing-time emission)
* Window never closes (watermark stuck)

### 

### **A.5.5 TM-3: Late Data Handling**

**Purpose:** Verify that late data is handled per declared policy.

**Setup:**

* Define lateness policy: accept up to 10s late; emit corrective delta
* Define tumbling windows of 60s
* Inject on-time events for window [0-60s)
* Advance watermark to 70s (window closed, frame v1 emitted)
* Inject late event with timestamp 55s (5s late, within bound)

**Procedure:**

1. Window [0-60s) closes at watermark=60s; frame v1 emits
2. Late event (timestamp=55s) arrives at watermark=70s
3. Materializer recomputes window [0-60s) with late data included
4. Materializer emits corrective frame v2 (same contextId, key, window; incremented version; type=delta)

**Pass Criteria:**

* Frame v2 emitted with version = v1.version + 1
* Frame v2 has same window as v1
* Frame v2 body reflects late data inclusion
* late\_data.accepted telemetry emitted

**Failure modes:**

* Late data ignored (no corrective frame)
* Late data causes error/crash
* Frame v2 has different window (incorrect)

### **A.5.6 OR-1: Ordering Under Retry**

**Purpose:** Verify that per-key ordering is preserved despite retries.

**Setup:**

* Emit 3 frames for same (contextId, key): F1 (v1), F2 (v2), F3 (v3)
* Inject delivery failure for F2 (simulated transient error)
* Retry F2 after F3 is already sent

**Procedure:**

1. Subscriber receives F1 → acks
2. Subscriber receives F2 → fails (injected error)
3. F2 queued for retry
4. F3 sent → **MUST be blocked until F2 succeeds** (per-key ordering)
5. F2 retried → succeeds → acks
6. F3 delivered → acks

**Pass Criteria:**

* Subscriber observes order: F1, F2, F3 (never F1, F3, F2)
* Telemetry shows delivery.retry for F2

**Failure modes:**

* Out-of-order delivery (F3 before F2)
* F3 never delivered (deadlock)

### **A.5.7 OR-2: Per-Key Isolation**

**Purpose:** Verify that frames for different keys can be delivered concurrently without order constraints.

**Setup:**

* Emit frames for two keys: key=A (frames A1, A2), key=B (frames B1, B2)
* Introduce artificial delay in processing B1

**Procedure:**

1. A1, A2, B1, B2 emitted in that logical order
2. B1 delivery delayed (simulated slow subscriber for key=B)
3. A1, A2 delivered successfully to subscriber

**Pass Criteria:**

* A1, A2 delivered without waiting for B1 (keys are independent)
* B1, B2 eventually delivered in order (per-key guarantee holds)

**Failure modes:**

* A2 blocked waiting for B1 (incorrect global ordering)
* B2 delivered before B1 (per-key ordering violated)

### 

### **A.5.8 RP-1: Replay by Version**

**Purpose:** Verify that frames can be replayed by (contextId, key, version) with identical idempotencyKey.

**Setup:**

* Emit frame F1 with (contextId=C, key=K, version=10)
* Record: original\_idempotencyKey, original\_body
* Wait for TTL to pass (frame remains in store for test; or use long TTL)

**Procedure:**

1. Request replay: replay(contextId=C, key=K, version=10)
2. Receive replayed frame F1'
3. Compare: F1'.idempotencyKey vs original\_idempotencyKey
4. Compare: F1'.body vs original\_body

**Pass Criteria:**

* F1'.idempotencyKey == original\_idempotencyKey
* F1'.body == original\_body (byte-identical or semantically equivalent per documented rules)
* F1'.frameId MAY differ (new emission event)

**Failure modes:**

* Replay returns different idempotencyKey (non-determinism)
* Replay returns different body (incorrect storage or recomputation)
* Replay fails (frame not found despite being within TTL)

### **A.5.9 RP-2: Replay by Time Interval**

**Purpose:** Verify that frames can be replayed by time range with correct ordering.

**Setup:**

* Emit 5 frames for (contextId=C, key=K) with windows:
  + F1: [0-60s)
  + F2: [60-120s)
  + F3: [120-180s)
  + F4: [180-240s)
  + F5: [240-300s)

**Procedure:**

1. Request replay: replay(contextId=C, key=K, startTime=60s, endTime=240s)
2. Expect: F2, F3, F4 (windows ending in [60s, 240s))
3. Verify order: F2.window.end < F3.window.end < F4.window.end

**Pass Criteria:**

* Replay returns exactly {F2, F3, F4}
* Order preserved: by (window.end, version)
* Completeness: no frames missing; no extra frames

**Failure modes:**

* Missing frames (e.g., F3 not returned)
* Extra frames (e.g., F1 or F5 included)
* Out-of-order (e.g., F4 before F3)

### **A.5.10 EX-1: Admission Decision Telemetry**

**Purpose:** Verify that extension hooks emit required telemetry for admission decisions.

**Setup:**

* Implement pre-materialize hook with policy that denies 25% of requests (e.g., rate limit)
* Configure hook to emit admission.decision events
* Generate 100 materialization requests

**Procedure:**

1. Send 100 requests for frame materialization
2. Hook evaluates each request; approximately 25 should be denied
3. Collect all admission.decision events
4. Verify event structure and labels

**Pass Criteria:**

* All 100 requests generate admission.decision events
* Events include required labels: stage, contextId, key, verdict, reasonCode
* verdict values are valid: "admit", "deny", or "defer"
* Denied requests have meaningful reasonCode (e.g., "rate\_limit\_exceeded")
* Approximately 75 admitted, 25 denied (within statistical variance ±10%)
* No materialization proceeds for denied requests (verify no frame.materialized events for denied keys)

**Failure modes:**

* Missing telemetry events
* Events missing required labels
* Denied requests still produce frames (hook not enforced)
* reasonCode is empty or meaningless (e.g., "error" with no detail)

### **A.5.11 EX-2: Budget Lease and Commit**

**Purpose:** Verify budget lease/commit/cancel semantics with no double-spend.

**Setup:**

* Implement budget tracking with time and token budgets
* Configure materialization to lease {timeMs: 100, tokens: 50}
* Run 50 materializations: 45 succeed, 5 fail (simulated errors)

**Procedure:**

1. For each materialization attempt:
   * Emit budget.lease before evaluation
   * On success: emit budget.commit with actual usage
   * On failure: emit budget.cancel with reason
2. Collect all budget events
3. Group by leaseId

**Pass Criteria:**

* Each leaseId has exactly one budget.lease event
* Each leaseId has exactly one of: budget.commit OR budget.cancel (never both)
* No orphaned leases (lease with neither commit nor cancel within TTL)
* Total committed units = sum of actual usage from successful materializations
* Leak rate: ≤1% of leases unaccounted
* No double-commits (same leaseId committed twice)

**Failure modes:**

* Lease without commit or cancel (budget leak)
* Both commit and cancel for same lease (double accounting)
* Commit after cancel (state machine violation)
* Committed units don't match actual work

### 

### **A.5.12 EX-3: Redaction Application**

**Purpose:** Verify that redaction is applied per classification policy before frame publication.

**Setup:**

* Define classification policy: frames marked "confidential" MUST redact fields ["$.user.email", "$.user.ssn"]
* Create test view that produces frames with sensitive fields
* Configure 10 frames: 5 classified "internal" (no redaction), 5 classified "confidential" (redaction required)

**Procedure:**

1. Materialize all 10 frames
2. For each frame:
   * Check extensions.classification field
   * If classification="confidential", verify security.redaction event emitted
   * Fetch frame body from store
   * Verify redacted fields are masked/removed
3. For "internal" frames, verify no redaction applied

**Pass Criteria:**

* 100% of "confidential" frames have redaction applied (5/5)
* 0% of "internal" frames have redaction applied (0/5)
* security.redaction events emitted for all confidential frames
* Redacted fields are absent or replaced with mask value (e.g., "[REDACTED]", null)
* Original values NOT present in stored frame bodies

**Failure modes:**

* Confidential frame stored without redaction (PII leak)
* Redaction applied to wrong fields
* Redaction event emitted but frame body still contains original values
* All frames redacted (policy not applied selectively)

## 

## **A.6 Conformance Report Format**

### **A.6.1 Report Structure**

Implementations MUST produce a conformance report in **JSON format** with the structure defined below.

**File naming:** rcm-conformance-{implementation-name}-{version}.json

**Top-level schema:**

{

"rcmVersion": "string (e.g., '1.0')",

"reportVersion": "string (e.g., '1.0')",

"generatedAt": "RFC 3339 timestamp",

"implementation": {

"name": "string",

"version": "semver",

"vendor": "string",

"contactEmail": "email",

"websiteUrl": "url",

"repositoryUrl": "url (optional)",

"conformanceClass": "RCM"

},

"environment": {

"region": "string (e.g., 'us-west-2', 'on-premises')",

"nodeCount": "integer",

"nodeSpec": "string (e.g., '4 vCPU, 16GB RAM')",

"storageType": "string (e.g., 'SSD', 'S3')",

"networkTopology": "string (e.g., 'single-AZ', 'multi-AZ')",

"notes": "string (optional)"

},

"declaredSLOs": {

"windowMinutes": "integer (e.g., 60)",

"stepMinutes": "integer (e.g., 1)",

"freshnessMsP95": "integer",

"freshnessMsP99": "integer",

"deliveryMsP95": "integer",

"dlqRatePer10k": "float",

"fairnessDeviation": "float (optional, if fairness implemented)",

"profilesClaimed": ["array of strings (e.g., ['R1'])"]

},

"testResults": {

"executionDate": "RFC 3339 timestamp",

"durationMinutes": "integer",

"tests": [

{

"id": "string (e.g., 'DV-1')",

"name": "string (e.g., 'Deterministic View Evaluation')",

"pass": "boolean",

"notes": "string (optional details, error messages)"

}

]

},

"sloResults": {

"freshness": {

"p95Ms": "integer (measured)",

"p99Ms": "integer (measured)",

"attainedWindowsP95": "float (0-1, e.g., 0.97)",

"attainedWindowsP99": "float (0-1)"

},

"delivery": {

"p95Ms": "integer",

"attainedWindowsP95": "float"

},

"dlqRatePer10k": "float",

"fairnessDeviation": "float (optional)"

},

"telemetry": {

"eventsSchemaVersion": "string (e.g., '1.0')",

"metricsSchemaVersion": "string (e.g., '1.0')",

"mappings": {

"eventNameMappings": {

"frame.materialized": "implementation-specific-name (if different)"

},

"metricNameMappings": {

"watermark\_lag\_ms": "implementation-specific-name (if different)"

}

},

"exportFormat": "string (e.g., 'OpenTelemetry', 'Prometheus', 'CloudWatch')"

},

"extensionPoints": {

"implemented": "boolean",

"hooksProvided": ["array of strings (e.g., ['pre-materialize', 'pre-publish'])"],

"policies": "string (description of governance policies)",

"documentationUrl": "url (optional)"

},

"artifacts": {

"rawDataUrl": "url (optional, link to detailed test data)",

"dashboardUrl": "url (optional, link to telemetry dashboard)"

}

}

### **A.6.2 Required Fields**

The following fields are **REQUIRED** in all conformance reports:

* rcmVersion, reportVersion, generatedAt
* implementation.\* (all subfields)
* environment.\* (all subfields except notes)
* declaredSLOs.\* (all subfields except optional ones)
* testResults.\* (all subfields)
* sloResults.\* (all required metrics per conformance class)
* telemetry.eventsSchemaVersion, telemetry.metricsSchemaVersion

### 

### **A.6.3 Optional Fields**

The following fields are **OPTIONAL** but recommended:

* environment.notes
* declaredSLOs.fairnessDeviation (only if fairness implemented)
* declaredSLOs.profilesClaimed
* telemetry.mappings (only if names differ from spec)
* extensionPoints.\* (only if extensions implemented)
* artifacts.\*

### **A.6.4 Report Validation**

Implementations SHOULD validate conformance reports against the JSON schema before publication.

**Schema validation requirements:**

* All required fields present
* Types match (integers, floats, booleans, strings, arrays, objects)
* Enum values valid (e.g., conformanceClass must be "RCM")
* SLO attainment ratios in range [0, 1]

**Publishing recommendations:**

* Publish report to implementation website or public repository
* Include in release artifacts (e.g., GitHub releases)
* Update report for each new version or significant environment change
* Maintain historical reports for transparency

## **A.7 Producer and Subscriber Reference Checks**

### **A.7.1 Producer Checks**

Implementations MUST verify the following properties for all frames produced:

**Envelope completeness:**

* All required header fields present (§4.3.1)
* frameId globally unique (no collisions)
* version monotonically increasing per (contextId, key)
* idempotencyKey stable for identical work
* inputs sufficient for replay (ranges + hashes present)

**Immutability:**

* Frames never mutated after publication
* Corrections emit new frames with incremented version

**Provenance integrity:**

* inputs hashes match actual source content (spot-check via sampling)
* planHash changes only when plan changes
* watermarkAt matches actual watermark state at emission time

**Validation procedure:**

1. Sample 1000 frames from test run
2. For each frame:
   * Verify all required fields present and valid
   * Check version sequence (no gaps, no duplicates)
   * Recompute idempotencyKey; verify matches header
   * Fetch source content; verify input hashes
3. Pass if 100% of samples pass all checks

### **A.7.2 Subscriber Checks**

Implementations MUST verify the following properties for all subscriptions:

**Ordering guarantee:**

* Per-key order preserved: frames for same (contextId, key) delivered by (window.end, version)
* Test: OR-1, OR-2 (§A.5.6, §A.5.7)

**At-least-once delivery:**

* Every emitted frame delivered to every subscriber (may have duplicates)
* Test: inject 100 frames; verify all 100 reach subscriber (allow duplicates)

**Idempotent consumption:**

* Subscribers deduplicate using idempotencyKey
* Test: ID-1 (§A.5.2)

**Retry and DLQ:**

* Failed deliveries retried with backoff
* Terminal failures routed to DLQ with full context
* Test: inject transient failures; verify retries succeed; inject permanent failures; verify DLQ routing

**Validation procedure:**

1. Run subscription tests (OR-1, OR-2, ID-1)
2. Inject controlled failures (transient and terminal)
3. Verify delivery telemetry (attempts, retries, DLQ)
4. Pass if all tests pass and telemetry complete

## **A.8 Notes on Portability**

### **A.8.1 Name Mapping**

Implementations MAY use different names for events and metrics if:

1. The semantic meaning is preserved
2. The mapping is documented in telemetry.mappings section of conformance report
3. The alternative names are clearly equivalent (e.g., frame.published ≈ frame.materialized)

**Example mapping:**

"eventNameMappings": {

"frame.materialized": "frame\_emitted",

"delivery.attempt": "subscriber.receive\_attempt"

}

### **A.8.2 Environment Variability**

Conformance is valid **for the declared environment**. Materially different deployments (different hardware, network, storage) MUST re-run tests or clearly document deviations.

**Material changes that require re-testing:**

* Different cloud provider or region
* 2x or more difference in node count or specs
* Different storage backend (e.g., SSD → HDD, local → network)
* Different network topology (single-AZ → multi-AZ)

**Non-material changes:**

* Minor version bumps (patches, bug fixes) within same runtime
* Configuration tuning (buffer sizes, timeouts) that don't affect semantics
* Telemetry backend changes (Prometheus → CloudWatch)

### **A.8.3 Test Data Portability**

Implementations SHOULD contribute golden test vectors (inputs, expected outputs) to a shared repository for community use.

**Test vector format:**

{

"vectorId": "DV-1-basic-join",

"testCase": "DV-1",

"description": "Basic deterministic join over two sources",

"inputs": {

"source1": [ /\* array of events \*/ ],

"source2": [ /\* array of events \*/ ]

},

"viewSpec": { /\* declarative view definition \*/ },

"window": {"start": "...", "end": "..."},

"expectedIdempotencyKey": "sha256:abc123...",

"expectedBodyHash": "sha256:def456..."

}

**Contributing vectors:**

1. Create vectors for edge cases (late data, empty windows, cross-source joins)
2. Include both positive cases (should pass) and negative cases (should fail or handle gracefully)
3. Submit via pull request to conformance test suite repository
4. Vectors reviewed by maintainers; accepted vectors become part of canonical suite

### **A.8.4 Version Compatibility**

**Specification versioning:**

* Spec uses semantic versioning: MAJOR.MINOR.PATCH
* **MAJOR:** Breaking changes to normative requirements
* **MINOR:** Additive changes (new optional features, clarifications)
* **PATCH:** Corrections, typos, non-semantic improvements

**Implementation compatibility:**

* Implementations conformant to v1.0 MUST remain conformant to v1.x (minor updates)
* Implementations MAY adopt v2.0 features but MUST continue supporting v1.0 semantics for backward compatibility
* Conformance reports SHOULD specify highest spec version claimed

**Interoperability:**

* Frames produced by v1.0 implementation MUST be consumable by v1.x implementations
* Frame envelope extensions (implementation-specific fields) MUST be ignored by consumers that don't understand them
* Cross-version replay: v1.x implementation SHOULD be able to replay frames produced by v1.0 implementation if within TTL

# 

# **Annex B (Informative): References and Crosswalks**

## **B.1 Normative References**

The following documents are **normatively referenced** in this specification. Implementations claiming conformance MUST adhere to the semantics defined in these references where cited.

**RFC 2119:** Key words for use in RFCs to Indicate Requirement Levels  
 https://www.rfc-editor.org/rfc/rfc2119  
 Status: Best Current Practice  
 Cited in: §1.7, throughout Chapter 4

**RFC 8174:** Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words  
 https://www.rfc-editor.org/rfc/rfc8174  
 Status: Best Current Practice  
 Cited in: §1.7

**RFC 3339:** Date and Time on the Internet: Timestamps  
 https://www.rfc-editor.org/rfc/rfc3339  
 Status: Proposed Standard  
 Cited in: §4.3 (timestamp format for headers)

**ISO 8601:** Date and time format (for durations, intervals)  
 https://www.iso.org/iso-8601-date-and-time-format.html  
 Cited in: §4.3 (TTL duration format)

**Reactive Streams Specification v1.0.4** https://www.reactive-streams.org/  
 Status: Community specification  
 Cited in: §4.6 (delivery semantics, backpressure)

**W3C PROV-DM:** PROV Data Model  
 https://www.w3.org/TR/prov-dm/  
 Status: W3C Recommendation  
 Cited in: §4.3 (provenance concepts), Annex C (glossary)

## 

## **B.2 Informative References**

The following documents provide context and background but are **not normatively required** for conformance.

### **B.2.1 Stream Processing and Time Semantics**

**The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing** Tyler Akidau, Robert Bradshaw, Craig Chambers, et al.  
 VLDB 2015  
 https://research.google/pubs/pub43864/  
 Key concepts: Event-time vs processing-time, watermarks, windowing, triggers

**Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing** Tyler Akidau, Slava Chernyak, Reuven Lax  
 O'Reilly Media, 2018  
 ISBN: 978-1491983874

**Streaming 101 / 102** (O'Reilly Radar articles)  
 Tyler Akidau, 2015  
 https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101/  
 https://www.oreilly.com/radar/the-world-beyond-batch-streaming-102/

### **B.2.2 Event-Driven Patterns**

**Event Sourcing** Martin Fowler, 2005  
 https://martinfowler.com/eaaDev/EventSourcing.html

**CQRS (Command Query Responsibility Segregation)** Martin Fowler, 2011  
 https://martinfowler.com/bliki/CQRS.html

**CQRS Documents** Greg Young  
 https://cqrs.files.wordpress.com/2010/11/cqrs\_documents.pdf

**Immutability Changes Everything** Pat Helland  
 CIDR 2015  
 http://cidrdb.org/cidr2015/Papers/CIDR15\_Paper16.pdf

### 

### **B.2.3 Reactive Architectures and UI Patterns**

**The Elm Architecture** Evan Czaplicki  
 https://guide.elm-lang.org/architecture/  
 Model-View-Update pattern for UI state

**Redux Documentation** Dan Abramov and the Redux team  
 https://redux.js.org/understanding/thinking-in-redux/three-principles  
 Three principles: Single source of truth, State is read-only, Changes are made with pure functions

**Unidirectional User Interface Architectures** André Staltz, 2015  
 https://staltz.com/unidirectional-user-interface-architectures.html  
 Survey of Flux, Redux, Elm, Cycle.js

**Reactive Streams JVM Specification** https://github.com/reactive-streams/reactive-streams-jvm  
 Reference implementation and TCK

### **B.2.4 Cognitive Science and Memory Models**

**The Principles of Psychology** William James, 1890  
 Chapter IX: The Stream of Thought  
 http://psychclassics.yorku.ca/James/Principles/prin9.htm

**Human Memory: A Proposed System and its Control Processes** Richard Atkinson, Richard Shiffrin, 1968  
 Psychology of Learning and Motivation, Vol 2  
 Modal model: sensory → short-term → long-term

**Working Memory** Alan Baddeley, Graham Hitch, 1974  
 Psychology of Learning and Motivation, Vol 8  
 Working memory model with central executive

**Episodic and Semantic Memory** Endel Tulving, 1972  
 Organization of Memory  
 https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4320042/

### 

### **B.2.5 Knowledge Representation**

**A Framework for Representing Knowledge** Marvin Minsky, 1974  
 MIT AI Lab Memo 306  
 https://courses.media.mit.edu/2004spring/mas966/Minsky%201974%20Framework%20for%20Knowledge.pdf

**The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures** H. Penny Nii, 1986  
 AI Magazine, Volume 7, Number 2  
 https://aaai.org/ojs/index.php/aimagazine/article/view/537

**Knowledge Graphs** Aidan Hogan, Eva Blomqvist, Michael Cochez, et al., 2021  
 ACM Computing Surveys, Vol 54, No 4  
 https://dl.acm.org/doi/10.1145/3447772

### **B.2.6 Edge Protocols and Interoperability**

**Model Context Protocol (MCP)** Anthropic, 2024  
 https://modelcontextprotocol.io/  
 https://spec.modelcontextprotocol.io/  
 Open protocol for LLM-application context exchange

**FIPA Agent Communication Language (ACL)** Foundation for Intelligent Physical Agents  
 http://www.fipa.org/repository/aclspecs.html  
 Historical agent messaging standard

### **B.2.7 Telemetry and Observability**

**OpenTelemetry Specification** https://opentelemetry.io/docs/specs/otel/  
 Vendor-neutral observability: traces, metrics, logs

**OpenTelemetry Semantic Conventions** https://opentelemetry.io/docs/specs/semconv/  
 Standard names and meanings for telemetry

**Prometheus Metric Types** https://prometheus.io/docs/concepts/metric\_types/  
 Counter, Gauge, Histogram, Summary

## **B.3 Crosswalk: RCM Terms to Adjacent Patterns**

This table maps RCM concepts to equivalent or analogous concepts in related patterns and systems.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **RCM Concept** | **Event Sourcing** | **CQRS** | **Materialized Views (DB/Streaming)** | **Reactive Streams** | **Elm/MVU/Redux** | **Blackboard** | **Knowledge Graphs** |
| **Source Signal** | Event log; every state change recorded | Command side produces changes | Change streams/CDC as inputs | Publisher emits items | Actions/Intents/Messages | Knowledge sources post hypotheses | Triples/edges flowing into graph |
| **Context View** | Projection built from events | Read side (query models) | View definition / streaming transform | Processing stage | update(model, msg) function | Specialists reading/writing board | SPARQL/Cypher queries |
| **Frame** | Snapshot derived from event projection | Read model instance (persisted/versioned) | Materialized view table / snapshot | Element/sequence item | Model state after reduce | Blackboard state at a time | Versioned graph slice |
| **Materializer** | Projection executor | Read model builder | View maintenance engine | Processor/Operator | Reducer/Update function | Blackboard controller | Query executor |
| **Subscription** | Projected stream to consumers | Read side feeds UIs/APIs | Change notifications; view update streams | Subscriber with backpressure | View re-render on state change | Knowledge sources react to changes | Downstream consumers of graph deltas |
| **Time & Watermarks** | Out-of-order replay handling | Not specified | Windowing/watermarks in stream engines | Backpressure & async sequencing only | N/A (UI time is immediate) | N/A | Temporal validity via reification |
| **Governance** | Orthogonal concern | Orthogonal concern | Orthogonal concern | Orthogonal concern | Orthogonal concern | Rarely formalized | Access control & provenance layers |
| **Lineage/Provenance** | Event IDs & positions | Not standardized | Not standardized | Not specified | Not specified | Not standardized | RDF provenance graphs (PROV-O) |

**Key observations:**

* **Event Sourcing:** RCM often consumes ES logs; frames are derived read models, not authoritative events
* **CQRS:** RCM implements governed read side; CQRS is principle, RCM is pattern
* **Materialized Views:** RCM standardizes what DB/streaming MVs do ad-hoc (envelope, lineage, delivery)
* **Reactive Streams:** Flow control (backpressure) is complementary; RCM adds memory lifecycle
* **UI Patterns:** Same unidirectional backbone; RCM extends to durable, multi-consumer, governed memory
* **Blackboard:** Similar shared-memory role; RCM adds versioning, lineage, delivery contracts
* **Knowledge Graphs:** Different layer; KGs are storage substrate; RCM governs reactive context lifecycle

## **B.4 Crosswalk: RCM vs. AI Frameworks**

This table positions RCM relative to popular AI/LLM frameworks.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Capability** | **LangChain** | **Semantic Kernel** | **MCP** | **RCM** |
| **Short-term chat memory** | ConversationMemory helpers persist recent turns | Long-term memory via Kernel Memory / vector stores | Not specified (edge protocol only) | Hot-tier frames (short TTL); reactive updates |
| **Retrieval / Indexing** | Retrievers wrap search; link to originals | Vector store abstractions; Kernel Memory for RAG | Protocol to fetch resources/tools | RCM feeds indices with governed frames (lineage + TTL) |
| **Orchestration** | Chains/agents & LangGraph; long-term memory agents | Agents & connectors; orchestrates skills | Client/server roles; message schema/versioning | Agent workflows subscribe to frames; governance orders admission |
| **Interoperability** | Framework-level (Python/JS APIs) | Framework-level (.NET APIs) | Open protocol (JSON-RPC) for cross-tool context | Pattern (internal); exposes via MCP/A2A at edges |
| **Governance** | Application-level (callbacks, rate limits) | Application-level (filters, connectors) | Not specified | Extension points (policy, budgets, security) built into pattern |
| **Lineage** | Via callbacks and metadata (non-standard) | Via connectors (non-standard) | Not specified | Frames carry provenance (inputs, planHash) by design |
| **Position** | Framework for building LLM apps | Framework for building LLM apps | Edge protocol for context exchange | Pattern for context lifecycle (internal substrate) |

**Integration patterns:**

* **LangChain + RCM:** LangChain retrieves context from RCM frames (via API or subscription); LangChain chain outputs feed back as RCM source signals
* **Semantic Kernel + RCM:** SK memory connectors read/write RCM frames; SK agents subscribe to frame updates for fresh context
* **MCP + RCM:** RCM exposes curated frames via MCP gateway at edges; MCP contexts from external systems ingested as RCM source signals
* **All three + RCM:** Use frameworks for agent logic and tool orchestration; use RCM for governed, versioned, auditable context substrate

**Why RCM complements (not competes):**

* Frameworks provide developer ergonomics and tool ecosystems
* RCM provides production semantics (versioning, lineage, governance, delivery)
* Frameworks can consume RCM frames for trustworthy, cost-controlled context
* RCM can ingest framework outputs (agent decisions, tool results) as source signals

## **B.5 Bibliography**

**Books:**

* Kleppmann, Martin. *Designing Data-Intensive Applications*. O'Reilly Media, 2017.
* Evans, Eric. *Domain-Driven Design*. Addison-Wesley, 2003.
* Vernon, Vaughn. *Implementing Domain-Driven Design*. Addison-Wesley, 2013.
* Akidau, Tyler; Chernyak, Slava; Lax, Reuven. *Streaming Systems*. O'Reilly Media, 2018.

**Papers:**

* Akidau et al. "The Dataflow Model." VLDB 2015.
* Helland, Pat. "Immutability Changes Everything." CIDR 2015.
* Lakshman, Avinash; Malik, Prashant. "Cassandra: A Decentralized Structured Storage System." ACM SIGOPS 2010.
* Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing." NSDI 2012.

**Specifications:**

* Reactive Streams: https://www.reactive-streams.org/
* OpenTelemetry: https://opentelemetry.io/docs/specs/otel/
* W3C PROV: https://www.w3.org/TR/prov-overview/
* CloudEvents: https://cloudevents.io/

**Community Resources:**

* RCM Specification Repository: https://github.com/critical-insight/rcm-spec (adjust to actual)
* RCM Conformance Test Kit: https://github.com/critical-insight/rcm-conformance (adjust to actual)
* RCM Discussion Forum: GitHub Discussions or mailing list (TBD)

# 

# **Annex C (Informative): Glossary**

This glossary defines key terms used throughout the specification. Terms are organized alphabetically with references to authoritative sources where applicable.

## **A**

**Acknowledgment (Ack):** A signal from a subscriber to the subscriptions bus confirming successful processing of a frame. Delivery semantics rely on acks for at-least-once guarantees and retry logic.

**Admission Control:** The process of deciding whether to admit, deny, or defer a unit of work based on policy, capacity, or resource availability. In RCM, admission control is implemented via extension points (§4.7).

**Agent:** A software component that acts autonomously to achieve goals, often using AI/ML techniques. In RCM context, agents subscribe to frames for fresh context and emit events that trigger view recomputation.

**At-Least-Once Delivery:** A delivery guarantee where each message/frame is delivered to subscribers one or more times. Subscribers must implement idempotent processing to handle duplicates. Specified in §4.6.1.

## **B**

**Backpressure:** A flow control mechanism where consumers signal capacity to producers, preventing overload. RCM subscriptions MAY use Reactive Streams-style backpressure (§4.6.6).  
 Reference: https://www.reactive-streams.org/

**Blackboard Architecture:** A pattern where independent specialists collaborate via a shared "blackboard" workspace. RCM frames can serve as blackboard knowledge units with versioning and lineage (§5.1.6).  
 Reference: Nii, AI Magazine 1986

**Brownout:** A controlled degradation mode where the system widens coalescing windows, emits coarse frames, or reduces features to preserve core SLOs under load. Specified in §4.4.5 and Chapter 5.

**Budget:** A multi-dimensional allowance (time, money, tokens, network, rows) that bounds resource consumption for a unit of work. Managed via leases (§4.7, Annex A.2.3).

## **C**

**Change Data Capture (CDC):** A technique for streaming row-level changes from databases as events. Common RCM source signal type.  
 Reference: Debezium documentation

**Classification:** A security label applied to frames indicating sensitivity level (e.g., public, internal, confidential, restricted). Drives redaction and access control policies (§4.7.3).

**Coalescing:** Delaying frame emission for a small, bounded duration to absorb multiple rapid source changes into a single frame, reducing version churn while adding bounded latency (§4.4.5).

**Conformance:** The property of an implementation satisfying all normative requirements in Chapter 4 and passing test vectors in Annex A. RCM defines one conformance class (

§4.1, §6.2).

**Conformance Report:** A structured JSON document (Annex A.6) that demonstrates an implementation's conformance by documenting test results, SLO attainment, environment, and telemetry mappings.

**Consumer:** Any agent, service, UI component, or analytics pipeline that subscribes to and processes frames. Consumers MUST implement idempotent processing (§4.2.6).

**Context View:** A declarative specification of how to compose sources (and/or other views) into a derived context. Views are deterministic plans with no mutable state (§4.2.2).

**CQRS (Command Query Responsibility Segregation):** An architectural pattern separating write models (commands) from read models (queries). RCM implements the read side with reactive semantics (§5.1.2).  
 Reference: Fowler, https://martinfowler.com/bliki/CQRS.html

## **D**

**Dead-Letter Queue (DLQ):** A quarantine destination for frames that repeatedly fail delivery after exhausting retries. DLQ entries include full context for debugging and remediation (§4.6.4).

**Delta Frame:** A frame representing an incremental change relative to prior state. Consumers fold deltas to reconstruct full state. Contrasts with snapshot frames (§4.2.4).

**Determinism:** The property that identical inputs and parameters produce identical outputs. RCM views MUST be deterministic for replay and audit (§4.5.1).

**Differential Privacy:** A mathematical framework for quantifying privacy risk in data releases by adding calibrated noise. May be applied to RCM aggregate frames (Chapter 12).  
 Reference: NIST SP 800-188

**Duplicate Detection:** The process of identifying repeated work or delivery via idempotencyKey matching. Enables safe retries and at-least-once delivery without duplicate effects (§4.5.3).

## **E**

**Event Sourcing:** A pattern where all state changes are persisted as an append-only event log. RCM commonly consumes event-sourced streams as source signals (§5.1.1).  
 Reference: Fowler, https://martinfowler.com/eaaDev/EventSourcing.html

**Event Time:** The timestamp reflecting when an event occurred in the source domain, independent of processing time. RCM uses event-time for windows and watermarks (§4.4.1).  
 Reference: Akidau et al., Dataflow Model, VLDB 2015

**Extension Point:** A well-defined hook where implementations inject cross-cutting behaviors (admission control, budgets, security) without coupling to core reactive semantics (§4.7).

## **F**

**Fairness:** The property that resources are allocated proportionally to configured weights, with aging to prevent starvation. Measured via weighted share deviation (Annex A.3.6).

**Frame:** An immutable materialization of a Context View at a specific point or interval in event-time, carrying headers for identity, version, lineage, time, and governance metadata (§4.2.4, §4.3).

**Frame ID:** A globally unique identifier for a specific frame instance (ULID, UUID). May change on replay but must remain unique (§4.3.1).

**Frame Store:** Durable storage for versioned frames with TTL and lineage. Supports queries by (contextId, key, version) and time intervals (§4.2.4).

**Freshness:** The degree to which context reflects recent source changes. Measured via watermark\_lag\_ms (§A.3.3).

## **G**

**Gate:** A rate-limiting mechanism that caps request or emission rates before work is admitted. Part of the governance ordering (§4.7).  
 Reference: RFC 2697 (token bucket)

**Governance:** Cross-cutting policies for access control, resource management, security, and observability. Implemented via extension points in RCM (§4.7).

**Golden Replay:** A test using pre-recorded input vectors with expected outputs to verify determinism and idempotency (§4.5.5, Annex A.5).

## **H**

**HITL (Human-In-The-Loop):** A gate requiring human approval before high-risk actions or exposures. Optional extension point in RCM (§4.7.3).

**Histogram:** A metric type recording distribution of values (e.g., latency) with support for percentile queries (p50, p95, p99). Required for RCM telemetry (Annex A.2.4).

## 

## **I**

**Idempotency:** The property that repeating the same operation has the same effect once. RCM requires idempotent plans (materialization) and idempotent consumers (delivery) (§4.5).

**Idempotency Key:** A stable hash derived from inputs, plan, window, and parameters that uniquely identifies a unit of work. Used for deduplication (§4.3.1, §4.5.3).

**Immutability:** The property that data cannot be changed after creation. RCM frames MUST be immutable; corrections emit new frames with incremented versions (§4.3.3).

## **K**

**Key:** A partition or correlation identifier (e.g., tenant ID, user ID, asset ID) that scopes ordering and delivery guarantees. Per-key ordering is a core RCM requirement (§4.4.7).

**Knowledge Graph:** A graph-structured knowledge base (RDF, Property Graph) with rich semantics. RCM can compose views over graph queries and emit frames with graph provenance (§5.1.7).  
 Reference: Hogan et al., ACM CSUR 2021

## **L**

**Late Data:** Events arriving with event-time timestamps earlier than the current watermark. Handled per declared lateness policy (accept, emit corrective delta, or drop) (§4.4.6).

**Lease:** A short-TTL reservation of budget units taken before execution and settled (committed or cancelled) after completion. Prevents double-spend (§4.7.3).

**Lineage:** Machine-readable evidence of which sources and transforms produced a frame. Recorded in the inputs and planHash fields (§4.3.1, §4.3.5).  
 Reference: W3C PROV-DM

## 

## **M**

**Materialized View:** A precomputed query result stored for fast access. RCM standardizes MV semantics with envelopes, lineage, and delivery contracts (§5.1.3).

**Materializer:** The executor responsible for applying time semantics, evaluating view plans, and emitting frames with complete envelopes (§4.2.3).

**MCP (Model Context Protocol):** An open protocol for exposing and consuming context between LLM applications and external systems. RCM exposes frames via MCP at edges (§5.5.1).  
 Reference: https://modelcontextprotocol.io/

**Monotonicity:** The property of always increasing (or non-decreasing). Watermarks and versions MUST be monotonic in RCM (§4.4.2, §4.3.4).

## **O**

**OpenTelemetry (OTel):** A vendor-neutral framework for telemetry (traces, metrics, logs). RCM telemetry aligns with OTel conventions (Annex A.2).  
 Reference: https://opentelemetry.io/

**Ordering:** The guarantee that frames for a given (contextId, key) are delivered in order by (window.end, version). Cross-key ordering is not required (§4.4.7).

## **P**

**Per-Key Ordering:** The delivery guarantee that frames with the same (contextId, key) arrive in order, even when cross-key deliveries are concurrent or out-of-order (§4.6.2).

**Plan:** The compiled, deterministic transform graph for a Context View, including operator versions and configuration. Identified by planHash (§4.5.1, §4.5.2).

**Plan Hash:** A cryptographic hash (e.g., SHA-256) of a view's compiled plan that changes if and only if the effective transform changes. Enables safe dual-run and replay (§4.5.2).

**Policy:** Declarative rules for access control, retention, classification, and consent. Applied via extension points in RCM governance (§4.7).

**Processing Time:** The wall-clock time when a system observes or processes an event. Contrasts with event-time (§4.4.1).

**Provenance:** Information about the origins and history of data. RCM frames carry provenance via inputs (source ranges + hashes) and planHash (transform identity) (§4.3.5).  
 Reference: W3C PROV-DM, https://www.w3.org/TR/prov-dm/

## **R**

**Reactive Composite Memory (RCM):** A design pattern for managing context in intelligent systems as governed, reactive dataflow: sources compose into versioned frames that update when dependencies change, with lineage and delivery guarantees.

**Reactive Streams:** A specification for asynchronous stream processing with non-blocking backpressure. RCM subscriptions MAY use Reactive Streams protocols (§4.6.6, §5.1.4).  
 Reference: https://www.reactive-streams.org/

**Redaction:** Removal or masking of sensitive fields before frame storage or egress. Applied per classification policy via security extension points (§4.7.3).

**Replay:** The ability to re-emit previously materialized frames by version or time interval for audit, debugging, or rebuilding state. Replayed frames MUST have identical idempotencyKeys (§4.5.4).

**Retry:** Automatic re-attempt of failed delivery with exponential backoff. RCM requires bounded retries with DLQ routing for terminal failures (§4.6.4).

## **S**

**Session Window:** A window type where boundaries are determined by gaps in data (periods of inactivity). Closes when inactivity exceeds a threshold (§4.4.3).

**Sliding Window:** A window type where fixed-size intervals overlap (e.g., 5-minute window sliding every 1 minute). Produces more frequent but overlapping frames than tumbling windows (§4.4.3).

**Snapshot Frame:** A frame representing complete state for a view, key, and window. Consumers can process snapshots independently without prior frames (§4.2.4).

**Source Signal:** A typed, timestamped change feed (events, CDC, file arrivals) with monotone positions that drives reactive recomputation (§4.2.1).

**Subscription:** A delivery relationship where consumers receive frames with at-least-once, per-key ordered guarantees. Subscriptions enable fan-out to many heterogeneous consumers (§4.2.5).

## **T**

**Time-To-Live (TTL):** A retention policy specifying how long a frame remains accessible before deletion. RCM defaults to short, non-zero TTLs for auditability (§4.3.1).

**Tumbling Window:** A window type with fixed-size, non-overlapping intervals (e.g., every 1 minute). Each event belongs to exactly one window (§4.4.3).

## **V**

**Version:** A monotonically increasing counter per (contextId, key) that increments by 1 for each new frame (snapshots and deltas share the sequence). Enables deterministic replay (§4.3.1, §4.3.4).

**View:** See Context View.

## **W**

**Watermark:** A threshold in event-time indicating "no events with timestamps earlier than this are expected" for a given (contextId, key). Watermarks MUST advance monotonically and trigger window closure (§4.4.2).  
 Reference: Akidau et al., Dataflow Model

**Weighted Fair Queueing (WFQ):** A scheduling algorithm that allocates service proportionally to configured weights. Used with aging to prevent starvation in RCM fairness implementations (Annex A.3.6).  
 Reference: https://en.wikipedia.org/wiki/Weighted\_fair\_queueing

**Window:** An event-time interval [start, end) that defines the scope of data included in a frame. Windows can be tumbling, sliding, or session-based (§4.4.3).

**Window Closure:** The moment a window transitions from Accepting to Closing state because the watermark has passed the window's end. Frames MUST NOT emit before closure (§4.4.4).

# 

# **Annex D (Informative): Extension Patterns**

This annex documents common patterns for implementing RCM extension points (§4.7). These patterns are **informative**: implementations MAY adopt, adapt, or ignore them. The normative requirement is that extension hooks emit required telemetry; the specific policies are implementation-defined.

## **D.1 Common Extension Point Patterns**

Extension points allow implementations to inject cross-cutting behaviors without coupling to core reactive semantics. This section surveys proven patterns from distributed systems, cloud platforms, and enterprise architectures.

### **D.1.1 When to Use Extension Points**

**Use extensions for:**

* ✅ Cross-cutting concerns (policy, security, cost) that apply uniformly
* ✅ Operational controls (admission, capacity, fairness) needed under load
* ✅ Compliance requirements (retention, redaction, audit trails)
* ✅ Multi-tenancy (isolation, quotas, chargeback)

**Don't use extensions for:**

* ❌ Business logic (belongs in view definitions)
* ❌ Data transformations (belongs in materializer)
* ❌ Consumer-specific logic (belongs in subscribers)

### **D.1.2 Design Principles**

**Orthogonality:** Extensions should not alter frame content or ordering; they observe, delay, deny, or charge but do not mutate.

**Composability:** Multiple extensions at the same hook should compose cleanly (e.g., policy check + rate gate + budget).

**Explainability:** Every decision (admit, deny, defer) MUST emit telemetry with reasonCode.

**Fail-safe defaults:** Denial is safe; admission requires explicit approval.

## 

## **D.2 Admission Control Patterns**

Admission control decides whether to accept, reject, or defer a unit of work before execution.

### **D.2.1 Simple Rate Limiting (Token Bucket)**

**Pattern:** Limit emission rate per context or tenant using token bucket algorithm.

**Implementation:**

* Each bucket has capacity C tokens, refill rate R tokens/second
* Materialization request consumes 1 token
* If bucket empty, request denied with reasonCode="rate\_limit\_exceeded"

**Configuration:**

admission:

type: token\_bucket

capacity: 100

refillRatePerSecond: 10

scope: per\_tenant # or per\_context, global

**Telemetry:**

admission.decision:

stage: pre-materialize

verdict: deny

reasonCode: rate\_limit\_exceeded

**When to use:** Protect downstream systems from traffic spikes; enforce fair use quotas.

### **D.2.2 Policy-as-Code (OPA, Cedar)**

**Pattern:** Evaluate declarative policies before admission using external policy engine.

**Implementation:**

* Before materialization or delivery, call policy engine with context: {contextId, key, requestor, classification, timestamp}
* Policy returns: {allow: boolean, reason: string}
* If denied, emit telemetry and skip work

**Example policy (OPA Rego):**

allow {

input.classification == "public"

}

allow {

input.classification == "internal"

input.requestor.role == "employee"

}

deny\_reason = "insufficient\_clearance" {

not allow

}

**Telemetry:**

admission.decision:

stage: pre-publish

verdict: deny

reasonCode: insufficient\_clearance

**When to use:** Complex, organization-wide policies; compliance requirements; need for audit trails on policy decisions.

**References:**

* Open Policy Agent (OPA): https://www.openpolicyagent.org/
* Cedar: https://www.cedarpolicy.com/

### **D.2.3 Concurrency Limiting**

**Pattern:** Cap in-flight materializations to prevent resource exhaustion.

**Implementation:**

* Maintain counter: active\_materializations
* On request: if active < max\_concurrency, admit and increment; else defer
* On completion: decrement counter
* Deferred requests wait in queue (queue-after-concurrency principle)

**Configuration:**

concurrency:

max\_materializations: 50

max\_deliveries: 200

**Telemetry:**

admission.decision:

stage: pre-materialize

verdict: defer

reasonCode: concurrency\_limit\_reached

concurrency\_utilization: 1.0 # gauge metric

**When to use:** Prevent overload during bursts; bound memory/CPU usage; ensure system stability.

## **D.3 Resource Management Patterns**

Resource management tracks and bounds consumption of compute, memory, network, tokens, and cost.

### **D.3.1 Multi-Dimensional Budgets**

**Pattern:** Track multiple resource dimensions (time, money, tokens, network, rows) with per-tenant or per-context budgets.

**Implementation:**

* Define budget vector: {timeMs, moneyCents, tokens, netBytes, rows}
* Before work: lease = tryLease(required\_units, ttlMs)
* After work: commit(lease, actual\_units) or cancel(lease, reason)
* Enforce limits: if any dimension exceeded, deny with reasonCode="budget\_exhausted\_{dimension}"

**Configuration:**

budgets:

tenants:

acme:

monthly:

timeMs: 3600000 # 1 hour

moneyCents: 10000 # $100

tokens: 1000000 # 1M tokens

netBytes: 10737418240 # 10GB

**Telemetry:**

budget.lease:

leaseId: uuid

units: {timeMs: 100, tokens: 500}

ttlMs: 30000

budget.commit:

leaseId: uuid

actual: {timeMs: 85, tokens: 450}

**When to use:** Multi-tenant SaaS with chargeback; cost control for LLM inference; prevent runaway spend.

### **D.3.2 Adaptive Budgets**

**Pattern:** Adjust budgets dynamically based on usage patterns, priority, or external signals (e.g., grid carbon intensity).

**Implementation:**

* Start with base budget
* Monitor: usage rate, time-of-day, tenant priority, carbon intensity
* Adjust: increase budget for high-priority during business hours; decrease during high-carbon periods

**Example logic:**

def adjustBudget(tenant, basebudget):

priority = tenant.priority # 1-5

hour = currentHour() # 0-23

carbon = getCarbonIntensity() # gCO2/kWh

multiplier = 1.0

if priority >= 4 and 9 <= hour <= 17:

multiplier \*= 1.5 # boost high-priority during business hours

if carbon > 400:

multiplier \*= 0.7 # reduce during high-carbon periods

return baseBudget \* multiplier

**When to use:** Dynamic workload management; sustainability goals; priority-based fairness.

### **D.3.3 Lease-Based Budgets with TTL**

**Pattern:** Reserve budget with short TTL to prevent indefinite holds; release on timeout or cancel.

**Implementation:**

* tryLease(units, ttlMs) → {granted: bool, leaseId}
* Lease expires after ttlMs; unreleased leases auto-cancel
* On success: commit(leaseId, actualUnits); on failure: cancel(leaseId, reason)
* Detect leaks: leases neither committed nor cancelled within TTL

**Benefits:**

* Prevents budget lock from hung workers
* Enables accurate accounting even with failures
* Bounded lease lifetime simplifies reasoning

**When to use:** Distributed systems with worker failures; need for accurate cost attribution despite retries.

## 

## **D.4 Security and Privacy Patterns**

Security extensions enforce classification, redaction, and access control.

### **D.4.1 Classification Taxonomy**

**Pattern:** Label frames with sensitivity levels; enforce policy based on labels.

**Common taxonomies:**

* **US Government:** Unclassified, Confidential, Secret, Top Secret
* **Enterprise:** Public, Internal, Confidential, Restricted
* **Healthcare (HIPAA):** PHI, De-identified, Limited Data Set
* **Finance:** Public, Internal, Material Non-Public Information (MNPI)

**Implementation:**

* Classifier examines frame body (keywords, patterns, ML model)
* Assigns label: classification = "confidential"
* Stores in frame header: extensions.classification
* Policy engine enforces: access control, retention, redaction per label

**Configuration:**

classification:

rules:

- pattern: "SSN:\\d{3}-\\d{2}-\\d{4}"

label: restricted

- pattern: "INTERNAL ONLY"

label: internal

default: public

**When to use:** Compliance (GDPR, HIPAA, SOC 2); multi-tenant SaaS; government/defense.

### 

### **D.4.2 Field-Level Redaction**

**Pattern:** Remove or mask specific fields before frame storage or egress.

**Implementation:**

* Policy defines redaction paths per classification: {classification: "restricted", redact: ["$.user.ssn", "$.user.email"]}
* Before publish, apply redactions: replace with null, "[REDACTED]", or hash
* Record redacted paths in extensions.redactedPaths
* Emit telemetry: security.redaction.count

**Example:**

// Before redaction:

{

"user": {

"name": "Alice",

"email": "alice@example.com",

"ssn": "123-45-6789"

}

}

// After redaction (classification=restricted):

{

"user": {

"name": "Alice",

"email": "[REDACTED]",

"ssn": "[REDACTED]"

}

}

**When to use:** PII protection; GDPR Article 5(1)(c) data minimization; export controls.

### 

### **D.4.3 Human-in-the-Loop Approval**

**Pattern:** Require human approval before exposing high-risk frames.

**Implementation:**

* Pre-publish hook detects high-risk condition (classification=restricted, cost>threshold, anomaly detected)
* Emit human.approval.requested with approval ID and TTL
* Block frame publication until: approval granted, approval denied, or TTL expires
* On timeout, route to DLQ or auto-deny per policy

**Workflow:**

1. Frame materialized, classified as "restricted"

2. Hook: requestApproval(frameId, ttl=15m)

3. Notify approver (Slack, email, dashboard)

4. Approver reviews frame metadata (not full body for privacy)

5. Approver grants or denies

6. System emits human.approval.granted or human.approval.denied

7. Frame published or blocked

**When to use:** High-stakes decisions (financial trades, medical diagnoses); regulatory requirements; trust-building.

## 

## **D.5 Observability Patterns**

Observability extensions emit telemetry for lineage tracking, cost attribution, and health monitoring.

### **D.5.1 Distributed Tracing Integration**

**Pattern:** Propagate trace context through RCM pipeline; link frames to upstream and downstream traces.

**Implementation:**

* On frame materialization, create span: materialize\_frame
* Record: contextId, key, version, planHash, inputs
* Propagate trace context to deliveries
* Subscribers create child spans: process\_frame
* Result: end-to-end trace from source change → materialization → delivery → consumption

**OpenTelemetry example:**

with tracer.start\_as\_current\_span("materialize\_frame") as span:

span.set\_attribute("rcm.contextId", contextId)

span.set\_attribute("rcm.key", key)

span.set\_attribute("rcm.version", version)

frame = evaluatePlan(inputs, plan)

publishFrame(frame)

**When to use:** Debugging latency issues; understanding causal chains; compliance audits.

**Reference:** OpenTelemetry Tracing: https://opentelemetry.io/docs/concepts/signals/traces/

### 

### **D.5.2 Cost Attribution and Chargeback**

**Pattern:** Attribute resource costs to tenants, contexts, or business units for chargeback.

**Implementation:**

* On budget commit, record: {tenantId, contextId, key, version, units, timestamp}
* Aggregate by tenant/month: Σ units.moneyCents
* Generate invoices or chargeback reports
* Dashboard: cost per tenant, cost per context, cost trends

**Metrics:**

budget.moneyCents\_committed (counter, labels: tenantId, contextId)

cost\_per\_frame\_avg (gauge, calculated: total\_cost / total\_frames)

**When to use:** Multi-tenant SaaS; internal IT chargeback; FinOps initiatives.

### **D.5.3 Lineage Graph Construction**

**Pattern:** Build a graph of frame dependencies for provenance visualization.

**Implementation:**

* Parse inputs field from each frame
* Create graph edges: frame\_v2 -[DERIVED\_FROM]-> source\_range, frame\_v2 -[SUPERSEDES]-> frame\_v1
* Store in graph database (Neo4j, TigerGraph) or as RDF triples
* Query: "Show all frames contributing to decision X" or "Trace data lineage for user Y"

**Graph schema:**

Nodes: Frame, Source, View

Edges: DERIVED\_FROM (Frame -> Source/Frame), SUPERSEDES (Frame -> Frame), DEFINED\_BY (Frame -> View)

Properties: frameId, contextId, key, version, timestamp, planHash

**When to use:** Regulatory compliance (GDPR Article 15); trust and explainability; debugging data quality issues.

## **D.6 Example: Multi-Stage Governance Chain**

This example combines multiple extension patterns into a coordinated governance chain.

**Design:** Policy → Gate → Concurrency → Priority → Budget (with aging)

### **D.6.1 Flow**

1. Materialization request arrives

2. Stage: Policy

- Check: Is this contextId allowed for this tenant?

- Check: Is classification permitted?

- Verdict: admit | deny (reasonCode="policy\_denied")

3. Stage: Gate (if admitted)

- Check: Token bucket has tokens?

- Verdict: admit | deny (reasonCode="rate\_limit\_exceeded")

4. Stage: Concurrency (if admitted)

- Check: active\_materializations < max?

- If yes: admit, increment counter

- If no: defer, add to queue (with priority)

5. Stage: Priority (if queued)

- Assign priority: 1 (low) to 5 (high) based on tenant tier, context type

- Apply aging: priority increases over time queued

- Schedule: WFQ with aging ensures fairness

6. Stage: Budget (when scheduled)

- tryLease(required\_units, ttlMs)

- If granted: proceed to materialize

- If denied: deny (reasonCode="budget\_exhausted")

7. Materialize

- Evaluate plan, produce frame

- Measure actual usage

8. Post-Materialize

- commit(lease, actual\_units)

- decrement concurrency counter

- emit frame.materialized

### 

### **D.6.2 Configuration**

governance:

policy:

engine: opa

policyUrl: https://policy-server/v1/data/rcm/allow

gate:

type: token\_bucket

capacity: 100

refillRate: 10

scope: per\_tenant

concurrency:

max\_materializations: 50

queue\_discipline: wfq\_with\_aging

aging\_halflife\_seconds: 60

priority:

default: 3

rules:

- condition: tenant.tier == "premium"

priority: 5

- condition: contextId.startsWith("incident\_")

priority: 4

budgets:

dimensions: [timeMs, moneyCents, tokens]

lease\_ttl\_ms: 30000

tracking: per\_tenant\_monthly

### **D.6.3 Telemetry Example**

For a single materialization attempt (tenant=acme, contextId=support\_context, key=user\_123):

[

{

"event": "admission.decision",

"stage": "policy",

"verdict": "admit",

"reasonCode": null

},

{

"event": "admission.decision",

"stage": "gate",

"verdict": "admit",

"reasonCode": null

},

{

"event": "admission.decision",

"stage": "concurrency",

"verdict": "defer",

"reasonCode": "concurrency\_limit\_reached"

},

{

"event": "queue.enqueued",

"priority": 3,

"queueDepth": 12

},

// ... time passes, aging increases effective priority ...

{

"event": "queue.dequeued",

"effectivePriority": 3.8,

"waitMs": 1200

},

{

"event": "budget.lease",

"leaseId": "lease\_xyz",

"units": {"timeMs": 100, "moneyCents": 5, "tokens": 500}

},

{

"event": "frame.materialized",

"frameId": "frame\_abc",

"version": 42,

"durationMs": 85

},

{

"event": "budget.commit",

"leaseId": "lease\_xyz",

"actual": {"timeMs": 85, "moneyCents": 4, "tokens": 450}

}

]

## 

## **D.7 Example: Simple Rate Limiting**

For teams that need only basic rate limiting without full governance:

**Implementation:**

class SimpleRateLimiter:

def \_\_init\_\_(self, capacity, refillRate):

self.bucket = TokenBucket(capacity, refillRate)

def onBeforeMaterialize(self, contextId, key):

if self.bucket.tryConsume(1):

emit("admission.decision", stage="gate", verdict="admit")

return {"admit": True}

else:

emit("admission.decision", stage="gate", verdict="deny", reasonCode="rate\_limit\_exceeded")

return {"admit": False, "reason": "rate\_limit\_exceeded"}

**Usage:**

* Attach to pre-materialize hook
* Denials skip materialization; no budget charged
* Simple, stateless, low overhead

**When to use:** Proof-of-concept; single-tenant systems; protect downstream from bursts.

## 

## **D.8 Example: Policy-as-Code Integration (OPA)**

**Policy file (Rego):**

package rcm

import future.keywords.if

import future.keywords.in

default allow = false

# Allow public data for anyone

allow if {

input.classification == "public"

}

# Allow internal data for employees

allow if {

input.classification == "internal"

input.requestor.role in ["employee", "contractor"]

}

# Allow restricted data for authorized roles

allow if {

input.classification == "restricted"

input.requestor.clearance >= 3

}

# Deny reason

deny\_reason = reason if {

not allow

reason := sprintf("insufficient\_clearance: needs clearance %d, has %d",

[required\_clearance(input.classification),

input.requestor.clearance])

}

required\_clearance(c) = 1 if c == "public"

required\_clearance(c) = 2 if c == "internal"

required\_clearance(c) = 3 if c == "restricted"

**Integration:**

def onBeforePublish(frameHeaders, frameBody):

classification = frameHeaders.extensions.get("classification", "public")

requestor = getCurrentRequestor()

decision = opaClient.evaluate("rcm/allow", {

"classification": classification,

"requestor": {

"role": requestor.role,

"clearance": requestor.clearance

}

})

if decision["allow"]:

emit("admission.decision", stage="pre-publish", verdict="admit")

return {"admit": True}

else:

reason = decision.get("deny\_reason", "policy\_denied")

emit("admission.decision", stage="pre-publish", verdict="deny", reasonCode=reason)

return {"admit": False, "reason": reason}

**When to use:** Organization-wide policies; separation of policy from code; need for policy versioning and audit.

## 

## **D.9 Summary: Choosing Extension Patterns**

**Decision matrix:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Need** | **Pattern** | **Complexity** | **When to Use** |
| Basic rate limiting | Token bucket | Low | Protect from bursts; simple quotas |
| Complex policies | Policy-as-Code (OPA) | Medium | Compliance; multi-condition rules |
| Concurrency control | Semaphore + queue | Low | Prevent overload; bound resources |
| Cost tracking | Multi-dimensional budgets | Medium-High | Multi-tenant SaaS; chargeback |
| PII protection | Classification + redaction | Medium | GDPR; HIPAA; export control |
| High-stakes decisions | HITL approval | Medium | Financial; medical; trust-building |
| Full governance | Multi-stage chain | High | Enterprise; regulated industries |

**Adoption path:**

1. **Start simple:** Rate limiting or concurrency control only
2. **Add observability:** Emit telemetry; build dashboards
3. **Layer governance:** Add policy checks, budgets as needs emerge
4. **Iterate:** Tune based on production metrics and incidents

**Anti-patterns to avoid:**

* ❌ Building all governance upfront (YAGNI violation)
* ❌ Governance without telemetry (can't verify or debug)
* ❌ Policy logic in materialization code (couples concerns)
* ❌ Extensions that mutate frame content (violates determinism)

# 

# **Appendix I (Non-Normative): Implementation Guidance**

This appendix provides brief pointers to implementation concerns. Detailed implementation strategies are covered in separate playbooks (vendor-agnostic and platform-specific guides).

## **I.1 Minimal Runtime Primitives**

RCM can be implemented with a small set of primitives:

* **Streams:** Append-only change feeds (Kafka, Kinesis, Pulsar, NATS)
* **Deterministic compute:** Pure functions, dataflow engines (Flink, Spark Streaming, custom)
* **Storage:** Versioned key-value or document store (Postgres, DynamoDB, MongoDB, S3)
* **Delivery:** At-least-once message bus (Kafka, RabbitMQ, SQS+SNS, gRPC streams)
* **Time:** Event-time tracking, watermarks (built into Flink/Beam, or custom)

## **I.2 Technology Mapping Examples**

**Kafka-based:**

* Sources: Kafka topics with CDC connectors
* Materializer: Kafka Streams or Flink consumer
* Frame Store: Kafka compacted topic + S3 for long-term
* Subscriptions: Kafka consumer groups

**Cloud-native:**

* Sources: Kinesis, EventBridge, Cloud Pub/Sub
* Materializer: Lambda, Cloud Functions, Cloud Run
* Frame Store: DynamoDB, Firestore, BigQuery
* Subscriptions: SNS+SQS, Pub/Sub subscriptions

**Actor-based:**

* Sources: Orleans virtual streams, Akka Streams
* Materializer: Stateful actors per (contextId, key)
* Frame Store: Actor state + event journal
* Subscriptions: Actor pub-sub or external bus

## 

## **I.3 Further Resources**

Detailed implementation guidance is available in:

* **RCM Implementation Playbook** (vendor-agnostic): Architecture patterns, technology mappings, deployment topologies
* **Platform-Specific Guides**: CognOS, Kafka/Flink, AWS, Azure, GCP
* **Community Recipes**: GitHub discussions, blog posts, conference talks

# 

# **Appendix II (Non-Normative): Cognitive Analogies**

This appendix briefly summarizes the cognitive science analogies from the original document (Chapter 8) to provide intuitions for RCM design choices.

## **II.1 William James: Stream of Consciousness**

James distinguished **substantive** (resting) and **transitive** (in-flight) parts of thought. In RCM: frames are substantive resting places; reactive recomputation is the transitive flight between states.

**Design takeaway:** Make each frame a coherent, inspectable state; keep transitions (windowing, watermarks) explicit and observable.

## **II.2 Atkinson-Shiffrin: Modal Model**

The modal model splits memory into sensory buffer → short-term → long-term stores. In RCM: coalescing buffers → hot frames (short TTL) → warm/cold frames (longer TTL with compaction).

**Design takeaway:** Separate latency concerns (hot tier) from forensic concerns (warm/cold); use deltas for hot, snapshots for warm/cold.

## **II.3 Baddeley-Hitch: Working Memory**

Working memory has limited capacity and a central executive. In RCM: agents act as the executive, drawing from hot frames (working memory); coalescing and capacity limits model attention constraints.

**Design takeaway:** Expect pressure at hot tier; provide summary views as reusable chunks; model attention as governance (gates, priorities, budgets).

## **II.4 Tulving: Episodic vs. Semantic Memory**

Episodic memory is time-stamped and eventful; semantic memory is fact-like. In RCM: versioned frame series = episodic; stable summary views = semantic.

**Design takeaway:** Keep episodic trails for audit; surface semantic projections for day-to-day consumers.

## **II.5 Forgetting and Consolidation**

Biological memory consolidates (short-term → long-term) and forgets. In RCM: short default TTLs with periodic snapshots; selective retention based on tags or access patterns.

**Design takeaway:** Default to short TTLs; extend only with justification; schedule consolidation to cap replay times.

# 

# **Appendix III (Non-Normative): Version History and Change Log**

## **Version 1.0 (January 2025)**

**Status:** Initial Community Specification Release

**Major components:**

* Normative semantics (Chapter 4): envelope, time/ordering, determinism, delivery, extension points
* Conformance test suite (Annex A): test vectors, telemetry requirements, SLO measurement
* Comparative guide (Chapter 5): positioning vs. Event Sourcing, CQRS, caching, UI patterns
* Extension patterns (Annex D): admission control, budgets, security, observability

**Design decisions:**

* Single conformance class (dropped Core/Governed split based on feedback)
* Extension points are optional but recommended; telemetry required if implemented
* Governance policies are implementation-defined; pattern specifies hooks and telemetry semantics

**Known limitations:**

* Conformance test kit implementation in progress (reference available Q1 2025)
* Cross-implementation interoperability not yet demonstrated (target: Q2 2025)
* Federation semantics deferred to future version (outlined in Chapter 12)

**Contributors:**

* Dan Vanderboom (Critical Insight Inc.) - Primary author
* [Additional contributors to be listed as they participate]

**Acknowledgments:**

* Early reviewers and implementers (to be named)
* Research communities: stream processing, event-driven architectures, cognitive science
* Standards bodies: W3C (PROV), Reactive Streams, OpenTelemetry

## **About This Specification**

**Title:** Reactive Composite Memory (RCM): A Pattern for Governed, Versioned Context in Intelligent Systems

**Version:** 1.0  
**Date:** January 2025  
**Status:** Community Specification Draft

**Maintainers:**

* Dan Vanderboom (Critical Insight Inc.)

**Repository:** github.com/critical-insight/rcm-spec  
**Discussion:** GitHub Discussions  
**Test Kit:** github.com/critical-insight/rcm-conformance

**License:**

* Specification text: Creative Commons Attribution 4.0 (CC-BY-4.0)
* Code examples and test suite: Apache License 2.0

**How to Cite:**

Vanderboom, Dan. "Reactive Composite Memory: A Pattern for Governed,

Versioned Context in Intelligent Systems." Version 1.0. Critical Insight Inc.,

January 2025. https://github.com/critical-insight/rcm-spec

**Contact:**

* Email: rcm-spec@criticalinsight.com
* Website: https://rcm-pattern.org (TBD)