Context Engineering: Purpose-Built Data Pipelines for Agents

Why "just connect everything" fails, and how event sourcing fixes it

The Agile Monkeys

March 2026

Why Raw Data Ingestion Fails Agents

A common approach to enterprise agents is to connect them directly to Slack, email, calendar, ticketing systems, and document repositories — giving the agent access to everything and expecting value to emerge. An agent with access to organizational data is more useful than one without, but connecting raw sources hits a ceiling fast, and the failure modes can be subtle enough that organizations don't immediately recognize them.

Drew Breunig's context failure taxonomy, cited in LangChain's context engineering guide (2025), identifies four ways raw data ingestion fails agents:

Context poisoning. When an agent processes a hallucinated or incorrect piece of information and it enters persistent memory, every subsequent interaction is contaminated. In a raw data pipeline, there's no validation layer between source and agent — bad data flows straight through.

Context distraction. Agents have limited context windows. Flooding them with everything — every Slack message, every email thread, every ticket comment — means the signal is buried in noise. The agent spends its reasoning budget navigating irrelevant information instead of answering the question.

Context confusion. Enterprise data contains superfluous details that mislead agents. Meeting notes with casual side conversations, email threads where the decision was reversed three replies deep, ticket comments that contradict the ticket description. Without preprocessing, agents treat all information as equally authoritative.

Context clash. Different data sources contain contradictory information about the same topic — an updated spec in Confluence vs. an outdated discussion in Slack vs. a partially implemented ticket in Jira. Raw ingestion preserves all versions without resolution, leaving the agent to arbitrarily choose which to trust.

Beyond these, there's a fifth failure mode that's arguably the most dangerous: silent degradation. As one AWS community analysis (2025) put it, RAG failures are "a data engineering problem disguised as AI" — caused by "stale data, broken pipelines, schema drift, inconsistent backfills, and the absence of contracts between producers and consumers." The LLM "will happily synthesize an authoritative answer from outdated information." Queries keep returning results. Latency stays normal. Retrieval scores look reasonable. But the underlying embeddings are weeks old, and the agent is confidently citing information that has been superseded. Without explicit freshness mechanisms, no one notices.

Context Engineering as a Discipline

Andrej Karpathy reframed the problem with an analogy that has reshaped how the industry thinks about agent data: if LLMs function like CPUs, their context windows are RAM. Context engineering is the discipline of filling that RAM with exactly the right information at each step — not all information, not random information, but precisely the information the agent needs for the current task.

As Karpathy put it, context engineering is "the delicate art and science of filling the context window with just the right information for the next step."

This isn't just theoretical. Anthropic's engineering team demonstrated the impact empirically. When they built purpose-specific, Claude-optimized versions of Slack and Asana tools — with refined descriptions, semantic identifiers instead of UUIDs, and curated response formats — these tools measurably outperformed the generic human-written implementations. Refined tool descriptions contributed to leading results on SWE-bench.

One study cited in LangChain's context engineering guide (2025) quantified another dimension: applying RAG to tool descriptions — retrieving the most relevant tool descriptions for a given query rather than loading all of them — yielded a reported 3x improvement in tool selection accuracy in that specific evaluation. The agent wasn't smarter; it was given better context.

Cognition AI, the company behind the Devin coding agent, has described context engineering as "effectively the #1 job of engineers building AI agents" (Cognition AI, 2025). Not model selection. Not prompt tuning. Context engineering — designing the information environment in which agents operate.

Purpose-built data pipelines are context engineering infrastructure. They ensure agents receive the right information in the right format at the right time, rather than drowning in raw data and hoping the model figures it out.

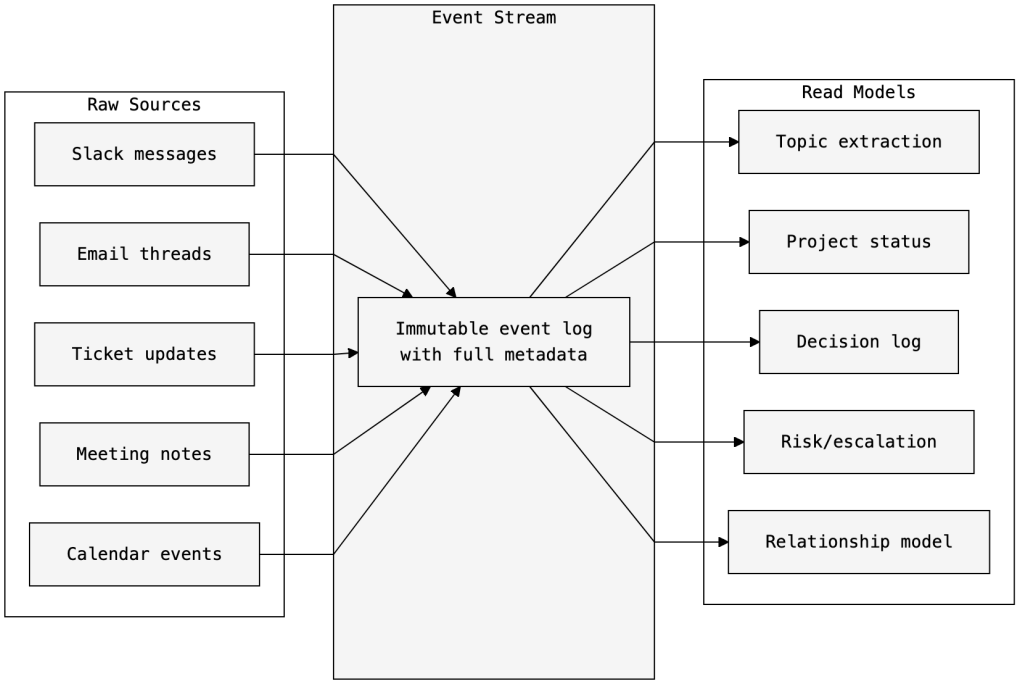
Event Sourcing + CQRS for Agent Data

Our proposed architectural pattern is to treat incoming data as an event source, combined with CQRS (Command Query Responsibility Segregation) — a

pattern borrowed from software engineering, where it is well-established in financial systems, e-commerce platforms, and distributed systems.

The Core Idea

Instead of storing the current state of enterprise data (the latest Slack messages, the current ticket status, the most recent document version), capture every change as an immutable event in a time-ordered log, with structured provenance identifiers that tie each event to its originating source (a specific Slack message ID, a particular Jira field change, a calendar event UID). Then build purpose-specific read models — materialized views optimized for specific query patterns — from that event stream, with each read model carrying correlation identifiers linking it back to the events that produced it.



Each read model is purpose-built for a specific agent consumption pattern. The topic extraction model clusters messages by subject across channels, maintaining topic threads even when conversations span multiple Slack channels and email threads. The project status model aggregates mentions of deadlines, blockers, completions, and milestones into a structured status view. The decision log identifies decisions — who made them, what context supported them, and what downstream actions followed. The risk/escalation model detects urgency signals, unresolved blockers, and escalation patterns.

Facts vs. Knowledge

This pattern introduces a crucial conceptual distinction, articulated well by Tracardi (2025) through an analogy to human memory: raw events are facts — immutable, timestamped records of what happened. Read models are knowledge — structured, interpreted views that can be rebuilt as understanding evolves.

The separation matters because knowledge needs to change even when facts don't. When a project's scope shifts, the historical Slack messages about the project (facts) don't change — but the project status summary (knowledge) must be regenerated to reflect the new context. When a decision is reversed, the original discussion (facts) remains, but the decision log (knowledge) must be updated.

The power of treating incoming data as an event stream is that you can attach different processors to the same events to generate different purpose-specific artifacts or trigger different actions. The same stream of Slack messages can feed a topic extractor, a decision logger, an urgency detector, and a compliance scanner simultaneously — each producing a different knowledge artifact optimized for a different agent consumption pattern, and each running independently without coordination. When a new use case emerges, you add a new processor to the existing stream rather than re-ingesting data. When a processor's logic improves, you replay historical events through the new logic to regenerate its projections.

In a traditional RAG pipeline, facts and knowledge are conflated. Slack messages are embedded and retrieved as-is, without the interpretive layer that separates what happened from what it means now. Event sourcing maintains the separation explicitly.

Provenance Through the Pipeline

Structured provenance is a first-class concern across the entire pipeline — not something bolted on for compliance. Every event carries a stable source identifier (the Slack message ID, the Jira ticket key, the calendar event UID) and, where applicable, a correlation ID that groups related events together. Read models inherit these references: a project status entry that mentions "three open blockers" must be able to enumerate the specific source events that support the claim.

This has three practical consequences:

> Derived artifacts are structured data, not unstructured summaries. A project status read model is not a blob of text — it's a structured record with explicit fields, each carrying references to the events that produced it. When an agent cites the status, the citation resolves to concrete source messages, not a similarity score against a vector.

Regeneration is targeted. When a source event is updated, corrected, or tombstoned (for GDPR erasure), the provenance chain identifies exactly which read models need to be regenerated. You don't rebuild everything — you rebuild only what depended on the changed facts.

Errors are traceable. When a downstream agent produces an incorrect output, you can walk the provenance chain backward from the agent's citation to the read model, from the read model to the source events, and from the source events to the raw data. Without provenance, LLM errors become undebuggable.

CDC-Driven Freshness

Silent degradation — agents confidently citing outdated information — is prevented through Change Data Capture (CDC). When source data changes, CDC triggers re-projection of affected read models. The project status model is regenerated when new tickets are created or existing ones updated. The decision log is refreshed when new discussions reference prior decisions.

This is not a periodic batch job ("re-index everything nightly"). It's a reactive pipeline that updates specific projections when specific source events occur. The result: read models stay fresh without the cost of regenerating everything, and staleness is detectable because every projection carries a last-updated timestamp.

The Slack Example in Full Detail

To make this concrete, consider how an organization's Slack data flows through this architecture:

Without event sourcing (the "connect everything" approach):

- All Slack messages are embedded and stored in a vector database
- An agent searching for "Project Alpha blockers" retrieves the 10 most similar messages
- Three of the retrieved messages are from two months ago and describe blockers that have been resolved
- One message is from a casual off-topic exchange that happens to mention "blocking" in a different context
- The agent synthesizes a response that includes resolved blockers as current issues
- Nobody notices because the response sounds confident and the retrieval scores are normal

With event sourcing:

- Slack messages are captured as immutable events: `{timestamp, author, channel, thread_id, content, reactions, edits}`

The topic extraction model clusters these events by subject, detecting when conversations shift topics and maintaining topic continuity across channels

The project status model processes topic clusters related to Project Alpha, identifying current blockers (not resolved ones) by tracking status changes over time

An agent searching for "Project Alpha blockers" queries the project status read model

The response contains only current, active blockers with their source conversations linked

CDC triggers re-projection when new messages arrive in relevant channels

The difference is not in the model's capability — it's in the quality of the information the model receives.

Replay and Reinterpretation

One of event sourcing's most powerful properties is replay. Because the event log is immutable and complete, read models can be rebuilt from scratch at any time — with different logic, different filters, or different aggregation strategies.

This matters for agents because interpretation requirements change. When a new compliance regulation takes effect, historical communications may need to be re-evaluated against the new standard. When organizational structure changes (teams merge, projects are reassigned), knowledge attributions must be updated. When a better summarization model becomes available, all summaries can be regenerated from the original events.

In a traditional RAG pipeline, re-interpretation requires re-processing and re-embedding the original documents. In an event-sourced system, it's a matter of running a new projection against the existing event stream.

Source-Specialist Knowledge Builders

Event sourcing provides the foundation: immutable event streams with purpose-built read models. But who builds those read models? The default approach is a generic pipeline — the same processing logic applied to Slack messages, Jira tickets, and Confluence pages. This works at small scale. It breaks as sources diverge in structure, semantics, and update patterns.

The alternative is source-specialist knowledge builders: dedicated processors, one per data source, each expert at extracting precisely the

information that consuming agents need from that specific source.

Why Specialization Matters

A Slack message and a Jira ticket both describe project status, but their structures are fundamentally different. A Slack message is conversational, threaded, often informal, with context distributed across reactions, replies, and channel membership. A Jira ticket is structured, stateful, with explicit fields for status, assignee, priority, and linked issues. A generic processor that treats both as "text to embed" loses the structural signals that make each source valuable.

A Slack knowledge builder understands threading (replies are context for the parent), reactions (a thumbs-up on a decision message is a signal of consensus), channel purpose (messages in #incidents carry different weight than #random), and conversational patterns (a question followed by an answer is a knowledge unit). A Jira knowledge builder understands state transitions (an issue moving from "In Progress" to "Blocked" is a risk signal), field relationships (priority + assignee + sprint = workload), and linking semantics (an issue blocking three others is a multiplied risk).

This is the same insight Anthropic validated empirically: purpose-built, Claude-optimized tool descriptions outperformed generic implementations because they encoded domain-specific knowledge about what matters. Source-specialist builders encode domain-specific knowledge about what matters in each data source.

Composable Knowledge Flows

The more powerful pattern emerges when one knowledge builder's output feeds another's input. A reporting agent's knowledge builders might work as follows:

- Source-level builders (Slack builder, Jira builder, Calendar builder) each process their source into structured, purpose-specific artifacts — topic summaries, blocker lists, decision logs, time allocations
- Team-level builders consume source-level artifacts to produce team-scoped views — project status rollups, team risk profiles, workload assessments
- Department-level builders consume team-level artifacts to produce department summaries — cross-team themes, aggregate risk, resource allocation patterns
- Organization-level builders consume department-level artifacts for executive views

Each level produces increasingly summarized knowledge. The artifacts at each level are available not just to the reporting agent, but to any agent with appropriate access — a risk-monitoring agent might consume the team-level risk profiles without needing the source-level Slack summaries.

This composable pattern is supported by production frameworks: CrewAI's task output forwarding chains agent outputs through sequential or hierarchical pipelines, and LangGraph's subgraph composition enables source-specialist subgraphs to be composed into larger knowledge flows.

Mandatory Guardrails

Composable knowledge flows introduce a critical risk: cascading error propagation. Each LLM-powered transformation is non-deterministic. If a Slack knowledge builder misinterprets a conversation, that error propagates through every downstream builder that consumes its output. Three layers deep, the error is untraceable to ground truth.

Three guardrails are non-negotiable:

Quality gates between stages. Each builder's output must pass validation before it can serve as input to the next stage. Validation can be deterministic (schema conformance, completeness checks) or LLM-assisted (consistency scoring against source data). The gate must be cheaper and more reliable than the builder itself — otherwise you've just added another layer of non-determinism.

Full data lineage. Every piece of knowledge produced by any builder must be traceable back to its source facts through every intermediate transformation. When a department summary says "three teams are blocked on the authentication refactor," you must be able to trace that claim back through team-level artifacts to the specific Slack messages and Jira tickets that support it. Without lineage, you cannot debug errors, audit claims, or satisfy compliance requirements.

Deterministic fallbacks. Not every transformation step needs an LLM. Data aggregation, filtering, deduplication, and format conversion should be deterministic code. Reserve the LLM for steps that genuinely require language understanding — topic extraction, sentiment analysis, summarization. Each deterministic step in the chain is a firebreak against error propagation.

The operational cost is real: maintaining N source specialists is more expensive than maintaining one generic pipeline. The tradeoff is worth it when the quality improvement directly impacts agent effectiveness — which, as the evidence shows, it consistently does. Start with your highest-value, most problematic data source (usually Slack or

equivalent), prove the quality improvement, then expand.

Purpose-Specific Tool Design

How agents access data matters as much as what data they access. Generic API wrappers — the default approach in most agent frameworks — tend to underperform purpose-built tools.

Few Thoughtful Tools Over Many Generic Ones

Anthropic's guidance is explicit: they recommend "building a few thoughtful tools targeting specific high-impact workflows" rather than wrapping every API endpoint. Generic wrappers expose the full complexity of an API — pagination parameters, filter syntax, authentication headers, response metadata — to an agent that doesn't need any of it and is confused by most of it.

A purpose-built Slack tool for an agent might expose three operations: `get_recent_decisions(project)`, `get_current_blockers(project)`, and `get_topic_summary(topic, timeframe)`. Each returns a structured, pre-processed response optimized for the agent's context window. Compare this to a generic Slack API wrapper that exposes `conversations.history`, `search.messages`, `reactions.get`, and 50 other endpoints, each returning raw JSON with pagination tokens, thread metadata, and user objects that the agent must parse.

Semantic Identifiers Over Technical IDs

Anthropic's guidance recommends replacing technical identifiers (UUIDs, MIME types, internal IDs) with semantically meaningful labels, noting this improves precision in retrieval tasks by reducing hallucinations (Anthropic Engineering, 2025). Instead of `user_id: 5f3a2b1c`, the tool returns `author: "Maria Chen, Engineering"`. Instead of `channel: C04NQJK2R`, it returns `channel: "#project-alpha-engineering"`.

This is not cosmetic. LLMs reason over text. When they encounter meaningful labels, they can cross-reference with other context ("Maria Chen mentioned this in the architecture review"). When they encounter UUIDs, they hallucinate associations.

One Agent, One Tool

A research paper on production-grade agentic workflows (arXiv:2512.08769, 2025) advocates a "one agent, one tool" design principle where each agent operates with a single tool for "predictable roles, simplified prompting,

and eliminated tool-selection noise." When an agent has one tool, there's no ambiguity about which tool to call. The agent's system prompt can be highly specific. Debugging is straightforward.

The paper also found that pure functions — deterministic operations executed without involving the LLM — are "deterministic, side-effect controlled, cheaper, faster, and fully testable." Not every operation needs to go through the LLM. Data formatting, filtering, pagination, and validation should be handled by deterministic code, with the LLM involved only in the reasoning steps that require it.

Observable Maturity Signals

Arcade.dev's catalog of 54 MCP tool design patterns (2025) introduces a practical framework for iterative tool improvement. The guide recommends monitoring for these signals:

- High retry rates may indicate poor tool descriptions — the agent is calling the tool with wrong parameters
- Repeated tool sequences suggest bundling opportunities — two tools that are always called together should be merged into one
- Partial failures signal the need for transaction boundaries — operations that should succeed or fail atomically are being split

These design heuristics create a feedback loop: deploy tools, monitor usage patterns, refine based on evidence, redeploy. Tool design is not a one-time exercise; it's an ongoing engineering discipline.

Counterarguments and Limitations

Event Sourcing Is Operationally Complex

Event sourcing is not free. The immutable event log grows continuously and requires storage management. Projections must be maintained, versioned, and rebuilt. Schema evolution in the event stream requires careful handling to avoid breaking existing projections. Organizations with no event sourcing experience will face a learning curve.

This complexity is real. Event sourcing has well-documented failure modes in traditional software engineering — event store performance considerations at scale, potentially long projection rebuild times for large event streams, and debugging difficulties when issues arise from the interaction between events and projection logic.

No Enterprise-Scale Evidence for Agent Data Specifically

Event sourcing and CQRS are proven patterns in financial systems (bank transaction logs), e-commerce (order processing), and distributed systems (event-driven microservices). But their application to agent data — Slack message processing, email summarization, meeting note structuring — has not been validated at enterprise scale. The architectural reasoning is sound (immutable events, purpose-built views, replay capability all map cleanly to agent data requirements), but we're honest that this is a novel application of established patterns, not a proven enterprise practice.

When Simpler Approaches Suffice

Not every data source needs event sourcing. Static document repositories (Confluence, Google Drive) that change infrequently may work fine with periodic re-indexing and basic chunking. The event sourcing approach adds the most value for high-velocity, conversational data sources — Slack, email, tickets — where context, temporality, and freshness matter.

The pragmatic recommendation: start with event sourcing for your highest-volume, most dynamic data source (usually Slack or equivalent). Prove the value of purpose-built read models on that single source. Then expand to other sources as the pipeline matures.

The MCP Complexity Question

A finding worth noting: research on production agentic workflows (arXiv:2512.08769) found that MCP introduces additional layers of abstraction that can reduce determinism, with the agent frequently making "ambiguous tool-selection decisions, inconsistently inferred invocation parameters, and occasionally failed with non-deterministic MCP responses." Generic protocol layers may underperform simpler, purpose-built integrations for specific use cases.

This doesn't invalidate MCP as a standard — it validates the argument that purpose-built tool design matters more than the protocol wrapping it. An MCP-compliant tool that is well-designed for agent consumption will outperform a poorly designed tool regardless of protocol.

Getting Started

For organizations ready to move beyond "connect everything":

Step 1: Pick one data source. Choose your highest-value, most problematic data source — the one where agents most often return stale, irrelevant, or contradictory information. Usually Slack or equivalent.

Step 2: Build one read model. Start with the most requested view — project status, decision log, or topic summary. Implement it as a projection from the event stream with CDC-driven freshness.

Step 3: Build a purpose-specific tool. Expose the read model through a single, well-designed tool with semantic identifiers and structured responses. Compare agent performance against the generic API wrapper.

Step 4: Measure the difference. Track retrieval relevance, agent response accuracy, token consumption, and user satisfaction. The quantitative improvement is the business case for expanding the pipeline.

Step 5: Expand incrementally. Add read models as you identify query patterns. Add data sources as you prove value. The event sourcing infrastructure scales horizontally; each new projection is independent.

The goal is not to build a perfect data architecture before deploying agents. It's to build the minimum viable pipeline that makes your agents meaningfully better — and then iterate based on evidence.

References

- Anthropic Engineering. "Writing Effective Tools for AI Agents." 2025. anthropic.com
- LangChain. "Context Engineering for Agents." 2025. blog.langchain.com
- Hoffman, K. "Event Sourcing: The Backbone of Agentic AI." Akka, 2025. akka.io
- AWS Builders Community. "RAG Is a Data Engineering Problem Disguised as AI." 2025. dev.to
- Spletzer, R. "Ask vs Act: Applying CQRS Principles to AI Agents." 2025. spletzer.com
- Tracardi. "Event Sourcing as the Backbone of AI Memory." 2025. tracardi.com
- Glean. "Knowledge Graphs as Agentic Engines." 2025. glean.com
- LlamaIndex. "Introducing Agentic Document Workflows." 2025. llamaindex.ai
- AWS Prescriptive Guidance. "Event-Driven Architecture for Serverless AI." 2025. docs.aws.amazon.com
- arXiv:2512.08769. "A Practical Guide for Designing, Developing, and Deploying Production-Grade Agentic AI Workflows." 2025. arxiv.org
- Arcade.dev. "54 Patterns for Building Better MCP Tools." 2025. arcade.dev
- Karpathy, A. "Context Engineering." 2025. x.com

- CrewAI. "Multi-Agent Knowledge Pipelines." 2025. docs.crewai.com
- LangChain. "LangGraph: Multi-Actor Applications with LLMs." 2025. langchain-ai.github.io/langgraph
- Anthropic. "Building Effective Agents." December 2024. anthropic.com