ADR-002: RAG Pipeline Architecture

AGENTIC-WORKFLOWS-V2 | STATUS: PROPOSED | 2026-02-28

This Architecture Decision Record defines the design and implementation strategy for integrating a production-grade Retrieval-Augmented Generation (RAG) pipeline into the `agentic-workflows-v2` multi-agent workflow engine. The decision covers vector store selection, embedding strategy, retrieval architecture, resilience patterns, observability, and a phased implementation roadmap totaling ~6,500 lines of source code across 47 files.

Context & Problem Statement

The `agentic-workflows-v2` engine currently provides three built-in agents — **Coder**, **Architect**, and **Reviewer** — that generate responses entirely from the LLM's parametric knowledge. There is no mechanism for grounding agent responses in project-specific documentation, codebases, or domain knowledge. This creates a critical gap: agents cannot reason over proprietary context, evolving codebases, or specialized domain corpora.

RAG solves this by retrieving relevant chunks from an indexed knowledge base and injecting them into the LLM context window before generation. Anthropic's contextual retrieval research demonstrates a **67% reduction in retrieval failures** when combining contextual embeddings with BM25 and reranking — a compelling empirical foundation for this approach.

Engine Stability

452 tests green. DAG executor and LangGraph engine both stable and ready for new integration surfaces.

Clean Integration

Dual tool system (`BaseTool` + `@tool`) fully operational, providing clear attachment points for RAG tooling.

Portfolio Signal

Production-grade RAG demonstrates systems engineering depth and positions the engine for enterprise adoption.

Decision Drivers & Constraints

Core Decision Drivers

→ Codebase Consistency

Follow established patterns exactly — no one-off abstractions.

→ Extensibility

Every component swappable via Protocol abstractions.

→ Retrieval Quality

Three-stage hybrid retrieval is current state of the art.

→ Resilience

Cache → retry → circuit breaker → fallback at every I/O boundary.

→ Operational Simplicity

LanceDB embedded, LiteLLM unified API — minimal ops overhead.

Hard Constraints

Dual Engine Compatibility

Native DAG + LangGraph must both be supported without divergence.

Pydantic v2 Contracts

`ConfigDict(extra="forbid")` — strict schema enforcement throughout.

Immutability First

Frozen dataclasses; new objects only, no mutation in place.

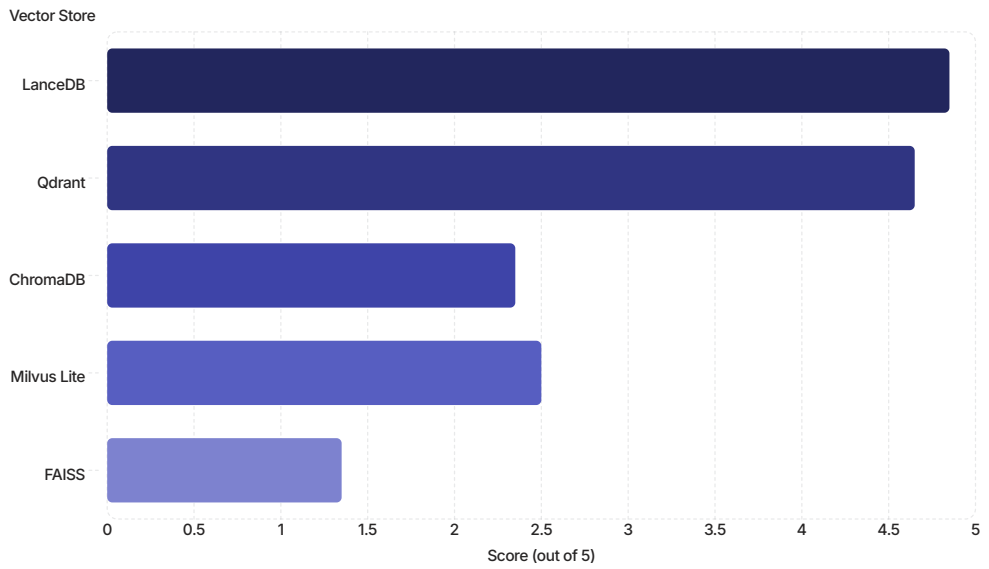
Async-First I/O

YAML-driven configuration, `CanonicalEvent` tracing via `TraceAdapter`.

Alternatives Evaluated

Six vector store candidates were scored across six criteria: async support, type safety, hybrid search, metadata filtering, persistence, and operational overhead. Embedding providers were evaluated on MTEB leaderboard performance, cost, and API unification.

Vector Store Scores



Why LanceDB Wins

Native Async

Full async/await support out of the box.

Pydantic-Native

Type-safe schema integration without adapters.

Built-in Hybrid

Tantivy BM25 + dense search without extra infrastructure.

Embedding Winner: Voyage 4 via LiteLLM at $0.06/1M tokens — tops MTEB retrieval leaderboard with MoE architecture, 32K context, and Matryoshka dims. Fallback chain: Voyage 4 → OpenAI → Nomic local.

Module Architecture: 16-Component Layout

RAG is placed as a peer module at `agentic_v2/rag/` with thin tool bridges into both engine surfaces. The 16-module layout enforces strict separation of concerns — each file targets 200–400 lines, each function under 50 lines.

Contracts & Config

`contracts.py` — Pydantic v2: `Document`, `Chunk`, `RetrievalResult`, `RAGResponse`
`config.py` — Frozen dataclasses: `EmbeddingConfig`, `ChunkingConfig`, `RAGConfig`

Ingestion & Indexing

`loaders.py`, `chunking.py`, `embeddings.py`, `ingestion.py`, `vectorstore.py` — Protocol-based loader, chunker, embedder, and LanceDB adapter.

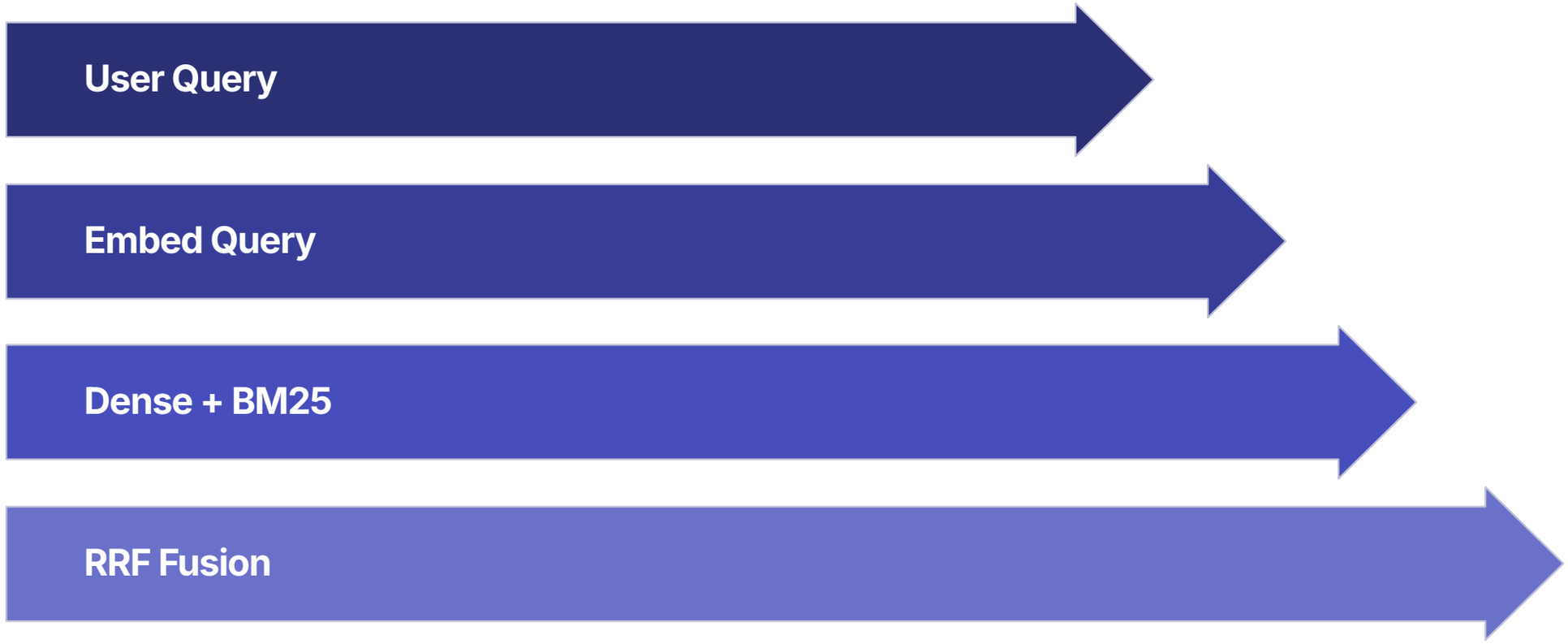
Retrieval & Grading

`retrieval.py`, `reranker.py`, `grading.py`, `context_assembly.py` — 3-stage pipeline, cross-encoder reranker, self-corrective grader, token-budgeted assembler.

Resilience & Ops

`resilience.py`, `tracing.py`, `evaluation.py`, `pipeline.py` — Circuit breaker, RAGTracer with CanonicalEvent emission, NDCG/Faithfulness eval, top-level orchestrator.

Query Data Flow



The query path is fully async end-to-end. Dense vector search and BM25 keyword search execute in parallel, then merge via Reciprocal Rank Fusion with `k=60`. The cross-encoder reranker applies a more expensive but precise scoring pass, followed by a self-corrective document grader that can loop back to retrieval if relevance thresholds are not met. The context assembler enforces a strict token budget before handing the final `RAGResponse` to the agent LLM. The 350ms retrieval latency overhead is effectively amortized by the 1–5s LLM inference time.

Resilience Architecture

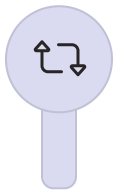
Every I/O boundary in the RAG pipeline is protected by a five-layer resilience stack defined in `resilience.py`. This ensures that transient failures in embedding providers, vector stores, or rerankers degrade gracefully rather than propagating as hard errors to the agent layer.

Layer 1 — Cache Check

Content-hash keyed LRU cache with TTL. Identical queries served instantly without any downstream I/O.

Layer 2 — Rate Limiter

`asyncio.Semaphore` per provider prevents burst overload and enforces per-provider concurrency limits.



Layer 3 — Retry Loop

Exponential backoff: 1s → 2s → 4s → 8s → 16s. Handles transient network and provider errors automatically.

Layer 4 — Circuit Breaker

Three-state machine: `CLOSED → OPEN → HALF_OPEN`. Prevents cascading failures when a provider is degraded.

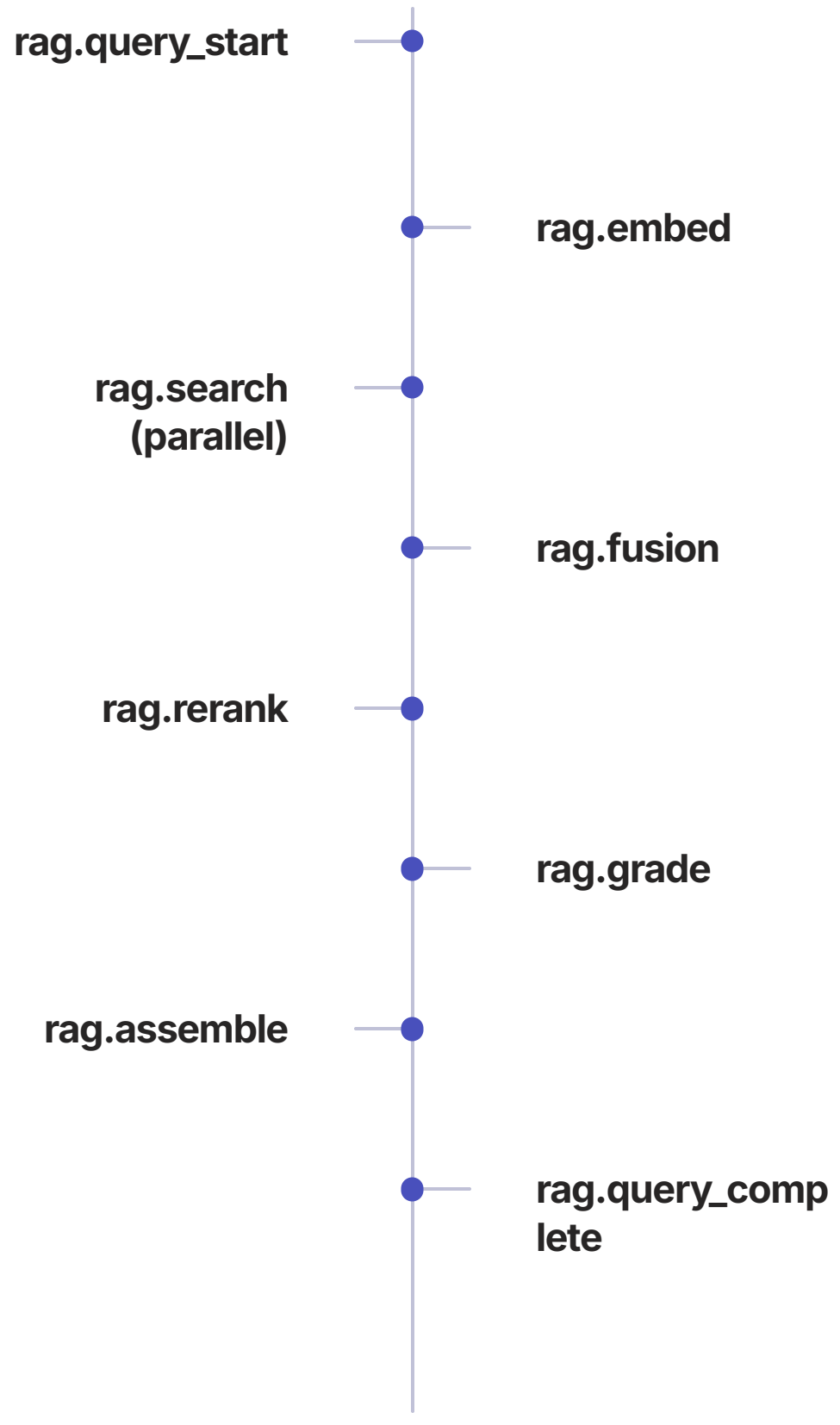
Layer 5 — Fallback Chain

Health-weighted provider selection: Voyage 4 → OpenAI → Nomic local. System remains operational even if primary provider is down.

Observability & Evaluation

CanonicalEvent Trace Chain

RAG-specific `CanonicalEvent` types flow through the existing `TraceAdapter` system, providing end-to-end visibility with zero new infrastructure:



Evaluation Rubric (YAML-Driven)

A golden dataset of 20 query-context-answer triples drives CI regression. Weighted scoring across four metrics with a hard faithfulness floor:

Metric	Weight	Floor
Faithfulness	0.30	0.70
Precision@k	0.25	—
Recall@k	0.25	—
Context Relevance	0.20	—

Faithfulness has a hard floor of 0.70 — any pipeline configuration scoring below this threshold fails CI regardless of other metric scores.

Consequences, Risks & Mitigations

Positive Outcomes

Knowledge-Grounded Agents

All three agents gain access to project-specific documentation, codebases, and domain knowledge at query time.

Composable Architecture

Protocol-based design means every component — embedder, chunker, reranker, store — is independently swappable.

Full Codebase Consistency

Frozen dataclasses, Pydantic v2, async-first, YAML config — zero deviation from existing engine patterns.

Risks & Mitigations

New Dependencies

`lancedb`, `litellm`, `sentence-transformers`/PyTorch added. **Mitigation:** Optional `[rag]` extras group with lazy loading — zero impact on non-RAG deployments.

Cold Start Latency

Initial pipeline warmup ~2–5s due to model loading. **Mitigation:** Lazy loading defers cost; explicit `warmup()` hook available for pre-warming.

Embedding Cost

~$0.30 per 10K documents at Voyage 4 pricing. **Mitigation:** Content-hash incremental indexing and LRU cache eliminate redundant embedding calls.

Implementation Roadmap: 9 Phases

The full implementation spans ~6,500 lines of source code and ~3,500 lines of tests across 47 files (33 source, 14 test). Phases 6, 7, and 8 execute in parallel to maximize throughput. All code adheres to the standards compliance matrix: 80%+ test coverage, TDD, no bare `Any`, custom `RAGError` hierarchy, and golden dataset regression in CI.

Phase 1 — 1

Foundation — contracts, config, error hierarchy, base Protocols

1 — **Phase 2**

Ingestion — document loaders, chunking strategies, ingestion pipeline

Phase 3 — 3

Indexing — LanceDB adapter, embedding provider, FallbackEmbedder

4 — **Phase 4**

Retrieval — 3-stage pipeline, RRF fusion, cross-encoder reranker, grader

Phase 5 — 5

Integration — RAGTool bridges for both native DAG and LangGraph engines

6 — **Phases 6–8**

Parallel: Resilience stack, CanonicalEvent tracing, evaluation harness

Phase 9 — 7

CLI tooling, workflow integration, documentation, and golden dataset finalization

6.5K	3.5K	16	80%+
Source Lines	**Test Lines**	**RAG Modules**	**Test Coverage**
Across 33 source files	Across 14 test files	In `agentic_v2/rag/`	Minimum required, TDD enforced