Architectural decision records for agentic-workflows-v2

Three critical architectural concerns face the `agentic-workflows-v2` monorepo: a dual execution engine that risks behavioral divergence, a circuit-breaker design exposed to cascade failures across LLM providers, and a supervisor state machine governing multi-round deep research with composite quality gating. Each concern carries production-reliability implications. The analysis below produces a formal ADR for each, grounded in workflow orchestration literature, distributed systems resilience patterns, and formal state machine theory.

ADR-001: Dual execution engine — LangGraph Pregel vs. native Kahn's DAG

Context

The repository maintains two active execution engines serving overlapping purposes. The **LangGraph engine** (`agentic_v2/langchain/`) uses `StateGraph` compiled to a `CompiledGraph` extending `Pregel` — Google's Bulk Synchronous Parallel model adapted by LangChain. It provides integrated checkpointing at every superstep, Langchain channel-based state management with typed reducers, streaming, human-in-the-loop interrupts, LangChain and time-travel debugging. GitHub The **native DAG executor** (`agentic_v2/engine/dag_executor.py`) implements Kahn's topological sort algorithm (Kahn, 1962) with dynamic parallel scheduling via `asyncio`, `StepExecutor`, `ExecutionContext`, and `StepStateManager`. The orchestrator agent's `execute_as_dag` method explicitly constructs and dispatches workflows through this second engine.

This dual-engine situation creates three risks: **behavioral divergence** (identical YAML workflows producing different outputs through different engines), **maintenance burden** (two codepaths for execution, error handling, and state management), and **feature asymmetry** (LangGraph provides ~9 integrated capabilities — checkpointing, streaming, human-in-the-loop, time travel, subgraph support — that the native engine would need to reimplement independently).

Analysis of options

Option A — Consolidate to LangGraph only. Temporal.io's architecture offers the strongest precedent here: a single, opinionated execution engine with strict determinism requirements produces the highest reliability. Temporal's production users report "production issues falling from once-a-week to near-zero." swyx LangGraph's Pregel execution model provides automatic checkpointing at every superstep, fault-tolerant resume (pending writes from successful nodes are preserved when others fail), Langchain PyPI and streaming langchain — all capabilities that would require significant engineering to replicate. However, LangGraph's BSP scheduling is **conservative with parallelism**: even when a node's dependencies are satisfied, it waits until the entire current superstep completes before scheduling the next batch. For pure DAG workflows with deep parallelism, this leaves performance on the table.

Option B — Consolidate to native DAG only. Kahn's algorithm delivers **optimal wavefront parallelism** — every node whose in-degree reaches zero is immediately schedulable, without waiting for unrelated nodes to complete. The algorithm runs in Θ(n + m) time. Wikipedia However, this path requires reimplementing checkpointing, state persistence, streaming, human-in-the-loop, time travel, conditional routing, and subgraph support. Prefect's evolution is instructive: they deliberately moved *away* from DAG-only execution toward richer execution models, finding that "if Python can write it, Orion can run it" Prefect was a more powerful abstraction than strict DAG constraints.

Option C — Keep both with a conformance layer. Apache Airflow provides direct precedent: since version 2.10.0, Airflow supports **multiple simultaneous executors** via comma-separated configuration, with per-task or per-DAG executor routing. This validates the pattern of maintaining parallel execution backends. However, verifying behavioral equivalence between engines is fundamentally hard — Rice's theorem establishes that proving behavioral equivalence is undecidable in general, Valerio-terragni and differential testing (running both engines on identical inputs and comparing outputs) is practical but incomplete. The conformance layer itself becomes a maintenance burden and potential failure point.

Option D — Abstract behind a common `ExecutionEngine` interface. This approach creates a facade that dispatches to either backend based on workflow characteristics, following the Strangler Fig migration pattern. Martin Fowler's updated guidance (2024) explicitly endorses transitional architecture: "People often balk at the necessity of building transitional architecture to allow the new and legacy system to coexist, code that will go away once the modernization is complete. While this may appear to be a waste, the reduced risk and earlier value from the gradual approach outweigh its costs." martinfowler The interface would expose `execute(workflow_def) → ExecutionResult` with engine selection based on workflow capabilities (cycles → LangGraph, pure DAG with max parallelism → Kahn's).

Tradeoff matrix

Criterion	A: LangGraph only	B: Native DAG only	C: Both + conformance	D: Common interface
Maintenance cost	Low (single engine)	Low (single engine)	High (two engines + test layer)	Medium (interface + two engines)
Feature completeness	High (built-in)	Low (must reimplement ~9 features)	High (use each engine's strengths)	High (delegate to appropriate engine)
Scheduling optimality	Moderate (BSP conservatism)	Optimal (wavefront parallelism)	Optimal (route by need)	Optimal (route by need)
Behavioral consistency	Guaranteed (one engine)	Guaranteed (one engine)	Risk of divergence	Risk of divergence (mitigated by routing)
Migration risk	Medium (must migrate all DAG workflows)	High (must reimplement LangGraph features)	Low (no migration needed)	Low (incremental migration)

Criterion	A: LangGraph only	B: Native DAG only	C: Both + conformance	D: Common interface
Vendor coupling	High (LangGraph API)	None	Medium	Low (abstracted)
Production precedent	Temporal.io	—	Airflow 2.10+	Strangler Fig pattern

Decision

Adopt Option D (common interface) as transitional architecture, converging toward Option A (LangGraph) over 2–3 quarters. The `ExecutionEngine` protocol should define `async execute(workflow: WorkflowDefinition, context: ExecutionContext) → ExecutionResult` with implementations `LangGraphEngine` and `KahnDAGEngine`. Route workflows containing cycles, human-in-the-loop nodes, or checkpointing requirements to LangGraph; route pure computation DAGs where maximum parallelism matters to the Kahn's engine. As LangGraph matures and parallelism improves, migrate remaining workflows to LangGraph and retire the native engine.

Consequences

Positive: Immediate risk reduction through interface abstraction. Incremental migration path. Preserves Kahn's algorithm's parallelism advantage for compute-heavy DAGs during transition. Conformance testing becomes scoped to the interface contract rather than full behavioral equivalence.

Negative: Transitional architecture has a carrying cost. The interface must be narrow enough to be implementable by both engines but expressive enough to expose each engine's strengths. LangGraph's Pregel API is explicitly documented as "not intended to be instantiated directly by consumers" — the abstraction must work at the `StateGraph` level, not the Pregel level.

Code-level recommendations

1. **Create `agentic_v2/engine/protocol.py`** defining `ExecutionEngine(Protocol)` with `execute`, `checkpoint`, and `get_status` methods.
2. **Wrap `agentic_v2/langchain/` in `LangGraphEngine`** implementing the protocol, delegating to `StateGraph.compile().ainvoke()`.
3. **Wrap `agentic_v2/engine/dag_executor.py` in `KahnEngine`** implementing the same protocol.
4. **Add `engine_selector.py`** that inspects `WorkflowDefinition` metadata (cycles, HiTL nodes, checkpoint requirements) and dispatches to the appropriate engine.
5. **Implement shadow execution mode** in CI: run a canonical workflow suite through both engines, compare final state outputs using property-based assertions (dependency ordering, final state field equality). Use `hypothesis` for property-based test generation.

6. **Refactor `orchestrator.py`'s `execute_as_dag`** to call through the protocol interface rather than directly constructing DAGs.

ADR-002: SmartModelRouter circuit-breaker hardening for multi-backend LLM routing

Context

The `SmartModelRouter` (`smart_router.py`) extends `ModelRouter` with per-model `ModelStats` tracking (EMA latency, percentile tracking from p50–p99, sliding window), a three-state circuit breaker (CLOSED → OPEN → HALF_OPEN), adaptive cooldowns via `CooldownConfig` (base 30s failure / 120s rate-limit / 60s timeout, **1.5× consecutive multiplier**, 600s max), and health-weighted model selection scoring success_rate at 60%, low latency at 20%, and recency at 20%. The fallback chain executes across **5+ providers** (GitHub Models, OpenAI, Azure OpenAI, Gemini, Anthropic, Ollama, local Phi Silica), classifying errors into rate-limit, timeout, and permanent categories.

Five specific risks threaten production reliability: cascade failures when multiple providers fail simultaneously, incorrect cooldown timing from wall-clock dependence, thundering-herd effects during half-open recovery, flat-rate cooldowns that ignore provider-specific rate-limit signals, and file-based stats persistence that breaks under multi-process deployments.

Risk analysis

(A) Cascade failure prevention

When one provider fails, its traffic redistributes to remaining providers. If OpenAI goes down and its load shifts entirely to Anthropic, Anthropic may become overloaded and trip its own circuit breaker — a classic cascade. Google's SRE book (Chapter 22) documents this precisely: [Google] "If cluster B fails, requests to cluster A increase to 1,200 QPS...the rate of successfully handled requests in A dips well below 1,000 QPS." [Google]

Recommendation: Implement **per-provider bulkhead isolation** with concurrent request limits. Each provider gets an independent semaphore (e.g., max 10 concurrent to Ollama, max 50 to OpenAI). When a provider's circuit opens, redistribute traffic proportionally across remaining providers *weighted by their remaining capacity*, not uniformly. [Microsoft Learn] Implement Google's **client-side adaptive throttling**: track `requests` (attempted) and `accepts` (succeeded) per provider over a 2-minute window, reject new requests with probability `max(0, (requests - K × accepts) / (requests + 1))` where K=2. [sre] This prevents a failing provider from consuming all retry budget.

(B) Partial failure handling

The current `get_model_for_tier` returns `None` when all candidates are exhausted. Three of five providers down simultaneously is not a theoretical edge case — correlated failures happen when providers share infrastructure (e.g., Azure outages affecting both Azure OpenAI and GitHub Models).

Recommendation: Implement a **tiered degradation strategy**. When primary tier models are exhausted, automatically fall through to lower tiers with reduced capability rather than returning `None`. Add a `degraded_mode` flag to responses so callers know they received a lower-tier model. Keep local Ollama or Phi Silica as an **always-available last-resort** bulkhead — local models cannot experience network-level failures. Add a global `system_health` metric: when >50% of providers are in OPEN state, activate load-shedding (reject low-priority requests, queue medium-priority, serve only high-priority).

(C) Cooldown timer correctness

The current implementation uses `datetime.now(timezone.utc)` for cooldown comparisons. Wall clocks are subject to **NTP step corrections** that can jump forward or backward by seconds or even minutes. Medium A backward jump extends cooldowns unexpectedly; a forward jump ends them prematurely. Python's `time.monotonic()` uses `CLOCK_MONOTONIC` on Linux, which is immune to NTP steps and never runs backward.

Recommendation: Replace all cooldown/timeout tracking with `time.monotonic()`. Store wall-clock timestamps separately for logging and debugging only. The pattern:

```python
# BEFORE (dangerous)
cooldown_end = datetime.now(timezone.utc) + timedelta(seconds=60)
is_cooled = datetime.now(timezone.utc) >= cooldown_end

# AFTER (correct)
cooldown_end = time.monotonic() + 60.0
is_cooled = time.monotonic() >= cooldown_end
```

For JSON persistence, store monotonic offsets relative to a reference point (process start time) and recalculate absolute values on load.

(D) Half-open probe strategy

The current design requires **2 successes** to transition from HALF_OPEN to CLOSED. Research from Bolshakov (2025) identifies a critical coordination issue: "Multiple actors racing to probe, racing to evaluate, racing to transition. Without coordination, your carefully configured recovery threshold becomes a suggestion." Bolshakov In a multi-threaded SmartModelRouter, two threads could simultaneously probe a half-open provider, one succeeding and one failing, leading to inconsistent state transitions.

Recommendation: Implement **serialized recovery probes with a lock**. Only one request at a time should probe a HALF_OPEN provider; all others receive immediate fallback. Bolshakov Use an `asyncio.Lock` per provider for probe coordination. Additionally, implement **exponential backoff on the reset timeout**: if a probe fails in HALF_OPEN, return to OPEN with `reset_timeout × 1.5` (capped at the max cooldown). Resilience4j's production default of **10 permitted calls in half-open** readme is appropriate for high-throughput systems, but for LLM routing where each call is expensive, **1 probe request** is more cost-effective. Use a lightweight endpoint (`/models` for OpenAI, `/api/tags` for Ollama) as the health check probe rather than a full chat completion.

(E) Rate-limit awareness

The current flat **120s cooldown** for rate limits ignores that providers communicate exactly when limits reset. OpenAI returns `x-ratelimit-reset-requests` and `x-ratelimit-reset-tokens` on every response (not just 429s), plus `Retry-After` on 429 errors. Milvus A 120s cooldown when the actual reset is in 8 seconds wastes provider capacity; a 120s cooldown when the reset is in 300s causes premature retries.

Recommendation: Parse provider-specific rate-limit headers and use them to set precise cooldown durations. Implement a **dual token bucket** per provider tracking both RPM (requests per minute) and TPM (tokens per minute). Pre-fill bucket state from `x-ratelimit-remaining-*` headers received on successful responses. Fall back to exponential backoff with jitter when headers are unreliable — Azure OpenAI's Responses API is known to return incorrect header values (`-1` and `0`) as of early 2025. OpenAI Developer Community Microsoft Learn Classify 429 errors separately from 5xx errors: 429 should trigger a rate-limit cooldown honoring `Retry-After`, while 5xx should trigger the standard failure circuit breaker. Shadecoder

Severity and likelihood risk matrix

Risk	Likelihood	Severity	Current mitigation	Recommended mitigation
All providers down simultaneously	Low	Critical	Returns None	Local Ollama last-resort + load shedding
Cascade failure (overload redistribution)	Medium	High	None	Bulkhead semaphores + adaptive throttling
Thundering herd on HALF_OPEN recovery	Medium	Medium	None	Lock-based serialized probes
Clock skew corrupts cooldowns	Low	Medium	None	Switch to `time.monotonic()`
Flat rate-limit cooldown wastes capacity	High	Medium	120s fixed cooldown	Parse Retry-After + dual token bucket
Stats file corruption under multi-process	Medium	Low	Atomic JSON write	SQLite WAL mode or `multiprocessing.Manager()`
Correlated provider failures (shared infra)	Medium	High	Independent breakers	Diversify across cloud regions + infrastructure
Half-open race conditions	Medium	Medium	None	`asyncio.Lock` per provider

Decision

Implement all five hardening measures (A–E) in priority order: **(C) clock correctness** first (lowest effort, prevents silent bugs), then **(E) rate-limit awareness** (highest capacity recovery), then **(D) half-open serialization** (prevents thundering herd), then **(A) cascade prevention** (bulkheads + adaptive throttling), and finally **(B) partial failure strategy** (most architectural change). Each is independently deployable.

Consequences

Positive: Eliminates the five identified reliability risks. Provider-aware rate limiting recovers capacity faster — honoring a 3s `Retry-After` instead of waiting 120s yields **40× faster recovery** for rate-limited providers. Monotonic clock usage prevents all clock-skew-related timer bugs. `Tty4` Serialized probes eliminate half-open state races.

Negative: Increased complexity in `smart_router.py` and `model_stats.py`. Per-provider token buckets require parsing provider-specific header formats, creating a maintenance burden as providers change their APIs. Bulkhead semaphore limits must be tuned per-provider and per-deployment.

Code-level recommendations

1. **`model_stats.py`**: Replace all `datetime.now(timezone.utc)` comparisons with `time.monotonic()` for cooldown tracking. Add a `_monotonic_reference` field for serialization.
2. **`smart_router.py`**: Add `_provider_semaphores: dict[str, asyncio.Semaphore]` for bulkhead isolation. Add `_probe_locks: dict[str, asyncio.Lock]` for half-open serialization.
3. **Create `agentic_v2/models/rate_limit_tracker.py`**: Implement `TokenBucket` class with `consume(tokens: int) → bool` and `refill_from_headers(headers: dict)`. Maintain dual buckets (RPM/TPM) per provider.
4. **`smart_router.py` `_execute_with_fallback`**: After any successful response, call `rate_limit_tracker.update_from_headers(response.headers)`. On 429, parse `Retry-After` and set cooldown to `max(retry_after_seconds, base_rate_limit_cooldown)`.
5. **`smart_router.py` `get_model_for_tier`**: When returning `None`, attempt lower-tier models before giving up. Set `response.degraded = True` flag when serving from a lower tier.
6. **`model_stats.py` persistence**: Replace JSON atomic write with SQLite using WAL mode for multi-process safety, or use `multiprocessing.Manager().dict()` for shared-memory state.

ADR-003: Deep research supervisor state machine with composite CI gating

Context

The `deep_research.yaml` workflow defines a **10-node pipeline** with bounded unrolled rounds R1–R4:

intake_scope → source_policy → [hypothesis_tree_tot → retrieval_react → analyst_ai + analyst_swe → cove_verify → coverage_confidence_audit] × R1-R4 → supervisor_decide → final_synthesis → rag_package. Each round has conditional when expressions gating on the composite Confidence Index: **CI = 0.25×coverage + 0.20×source_quality + 0.20×agreement + 0.20×verification + 0.15×recency**, with a threshold of CI ≥ 0.80 and additional gating on recent_source_count and critical_contradictions > 0. The supervisor uses a coalesce() pattern to select which round's output to forward.

This design implements techniques from three foundational papers: **ReAct** (Yao et al., ICLR 2023) for interleaved reasoning and action in the retrieval loop, arXiv **Tree of Thoughts** (Yao et al., NeurIPS 2023) for the hypothesis_tree_tot node's branching exploration, arXiv OpenReview and **Chain of Verification** (Dhuliawala et al., ACL 2024) for the cove_verify node's independent verification step. arXiv Semanticscholar The bounded unrolling approach (R1–R4 as explicit YAML steps) trades configuration duplication for deterministic, inspectable execution paths — aligned with bounded model checking's practice of "unrolling the transition relation of a finite state machine for a fixed number of steps k."

Formal state transition model

Applying Harel statechart formalism (Harel, 1987), the supervisor operates as a **hierarchical state machine** with three levels. The top-level superstate DeepResearch contains nested substates for each pipeline phase. Harel's three extensions to conventional state machines map directly: **hierarchy** (rounds as nested substates within an Execution superstate), **orthogonality** (parallel analyst agents analyst_ai + analyst_swe as concurrent regions), and **broadcast events** (CI evaluation results triggering transitions across the graph). ResearchGate Sofab

```
SUPERSTATE: DeepResearch
├─ INIT → Planning
├─ Planning
│   ├─ intake_scope (entry: parse query, decompose search space)
│   └─ source_policy (entry: define source constraints)
│       → [complete] → Execution
├─ Execution (SUPERSTATE with history H*)
│   ├─ Round[n] for n ∈ {1..4}    [guard: max_rounds >= n]
│   │   ├─ hypothesis_tree_tot    (ToT branching exploration)
│   │   ├─ retrieval_react        (ReAct search-fetch loop)
│   │   ├─ PARALLEL REGION:
│   │   │   ├─ analyst_ai         (domain analysis)
│   │   │   └─ analyst_swe        (technical analysis)
│   │   ├─ cove_verify            (CoVe independent verification)
│   │   └─ coverage_confidence_audit (compute CI, 6 metrics)
│   │       → [CI >= 0.80 AND all floors pass] → Coalesce
│   │       → [CI < 0.80 AND n < max_rounds] → Round[n+1]
│   │       → [n == max_rounds] → Coalesce
│   └─ H* (deep history: remembers best round state)
├─ Coalesce
│   ├─ entry: best_round = argmax(CI(r) for r in completed_rounds)
│   ├─ [best_CI >= 0.80] → Synthesis(full_confidence)
│   └─ [best_CI < 0.80] → Synthesis(degraded)
└─ Synthesis
    ├─ final_synthesis (entry: compose report from best_round output)
    └─ rag_package (entry: package for RAG indexing)
      → FINAL
```

Harel's **deep history mechanism** (H*) is critical here: it remembers which substate was active when the `Execution` superstate was last exited, Jannik Wempe enabling the `coalesce()` pattern to return to the best prior round's state if a later round regresses. The guard conditions on round transitions encode the pipeline's bounded iteration logic directly as statechart guards.

(B) Partial CI pass handling

The current design gates on composite CI only. This creates a **compensability vulnerability**: a research output with `verification_score = 0.10` (near-zero factual verification) could still pass if `coverage = 0.95`, `source_quality = 0.90`, `agreement = 0.95`, and `recency = 0.90`, yielding CI = 0.25(0.95) + 0.20(0.90) + 0.20(0.95) + 0.20(0.10) + 0.15(0.90) = **0.76** — which actually fails, but only barely. More dangerously, `verification = 0.35` with other scores at 0.90+ yields CI = 0.81 — a passing score despite severely inadequate verification.

Multi-criteria decision analysis (MCDA) literature identifies this as the **compensability problem** inherent in weighted arithmetic means. Healthcare quality measurement has converged on a hybrid approach: "If even one of the key metrics receives a failing score, the software cannot progress further" Dynatrace (Dynatrace quality gates methodology).

Recommendation: Implement **two-tier gating**:

- **Tier 1 — Per-metric floors** (non-compensatory): `source_quality ≥ 0.40`, `verification ≥ 0.40`, `coverage ≥ 0.35`, `agreement ≥ 0.35`, `recency ≥ 0.30`, `recent_source_count ≥ inputs.min_recent_sources`
- **Tier 2 — Composite gate** (compensatory): CI ≥ 0.80

Both tiers must pass. If Tier 1 fails but Tier 2 passes, the supervisor should trigger a **targeted remediation round** that re-executes only the failing dimension's nodes rather than a full round. For example, if `verification < 0.40` but all other floors pass and CI ≥ 0.80, re-run only `cove_verify → coverage_confidence_audit` rather than the entire `hypothesis_tree_tot → retrieval_react → analyst → cove_verify → audit` pipeline.

(C) Round regression mitigation

The SELF-REFINE paper (Madaan et al., NeurIPS 2023) explicitly acknowledges that "output quality can vary during iteration with improvement in one aspect but decline in another." OpenReview In the deep research pipeline, Round 3 might discover sources that introduce contradictions absent in Round 2's output, causing `agreement_score` to drop even as `coverage_score` improves.

The coalesce() pattern already provides the correct architectural response: **best-of-N selection** rather than always-use-latest. This aligns with Tree of Thoughts' backtracking mechanism — selecting the best path through the reasoning tree rather than committing to the most recent one. OpenReview The Iterative Agent Decoding paper (2025) confirms that "BON improvement ceases after 2 iterations" for certain tasks, suggesting diminishing returns are fundamental to iterative refinement. arXiv

Recommendation: The supervisor should maintain a **running best** across rounds:

```python
if CI(round_n) > CI(best_round_so_far):
    best_round = round_n        # improvement
elif CI(round_n) < CI(round_n-1):
    regression_count += 1       # track regressions
    if regression_count >= 2:
        break  # early-stop: consecutive regressions signal diminishing returns
```

When two consecutive regressions occur, stop executing further rounds even if `max_rounds` hasn't been reached — additional rounds are unlikely to improve quality and consume resources. The supervisor should log which specific metrics regressed and why (e.g., "R3 introduced 3 contradicting sources, dropping agreement from 0.85 to 0.71").

(D) Metric independence vs. composite gating

The CI formula's weights (0.25 + 0.20 + 0.20 + 0.20 + 0.15 = 1.0) are a valid weighted arithmetic mean, the simplest MCDA aggregation. The current weight assignment prioritizes **coverage** (0.25) as the most important dimension — justified because a research report with gaps is fundamentally flawed regardless of other qualities. **Recency** carries the lowest weight (0.15), appropriate since many research topics don't require ultra-recent sources.

TREC evaluation methodology, established by NIST in 1992, uses **NDCG (Normalized Discounted Cumulative Gain)** as its primary metric because it handles graded relevance with position-based discounting. OER Commons The pipeline's multi-dimensional CI is more expressive than NDCG but faces the same challenge: sensitivity to weight selection. MCDA literature "strongly recommends testing how different weight assignments affect outcomes" through sensitivity analysis. PubMed Central

Recommendation: Conduct **weight sensitivity analysis** by computing CI under ±0.05 perturbations of each weight. If small weight changes cause outcomes to flip between pass/fail, the weights need recalibration. Consider using the **geometric mean** as an alternative aggregation — it is less compensatory than the arithmetic mean and more heavily penalizes low scores on any dimension, which may be desirable for research quality:

python

```python
# Geometric mean alternative (less compensatory)
CI_geometric = (coverage**0.25 * source_quality**0.20 * agreement**0.20
               * verification**0.20 * recency**0.15)
```

The geometric mean naturally enforces that no single metric can be near-zero without dragging the composite down significantly. With the arithmetic mean, `verification = 0.10` contributes only -0.16 to CI; with the geometric mean, it contributes a multiplicative factor of `0.10^0.20 ≈ 0.63`, which is far more punishing.

(E) Graceful degradation when max_rounds exhausted

When all R1–R4 rounds complete without achieving CI ≥ 0.80, the system must deliver the best available result rather than failing entirely. Google SRE's principle is direct: "Serve lower-quality, cheaper-to-compute results to the user" when under stress. Google Google AWS Well-Architected's reliability pillar states: "Application components should continue to perform their core function even if dependencies become unavailable." AWS

Recommendation: Implement **tiered delivery** based on the best CI achieved:

Best CI achieved	Delivery tier	Behavior
≥ 0.80	Full confidence	Deliver without qualification
0.65 – 0.79	Moderate confidence	Deliver with warning; highlight weak metrics
0.50 – 0.64	Low confidence	Deliver with prominent disclaimer; mark unverified sections
< 0.50	Insufficient	Flag for human review; deliver only if explicitly requested

Attach structured **quality metadata** to every output:

```yaml
confidence_level: "moderate"
best_ci_achieved: 0.74
target_ci: 0.80
selected_round: 2      # best-of-4, not necessarily the last
rounds_completed: 4
failing_metrics:
  - verification: 0.55  # below 0.80 composite target contribution
  - recency: 0.48
passing_metrics:
  - coverage: 0.85
  - source_quality: 0.82
  - agreement: 0.78
```

Bounded unrolling is the correct design choice

The current approach of explicitly defining R1–R4 as separate YAML steps (static unrolling) rather than using engine-level dynamic looping is the right architectural decision. Bounded model checking literature confirms the pattern: "BMC operates by unrolling the transition relation of a finite state machine for a fixed number of steps k, and then checking whether a property violation can occur." Static unrolling provides **deterministic execution paths** (each round is a known, inspectable node), **no dynamic graph construction overhead**, **independent checkpointability** per round, and **bounded resource consumption** known a priori.

The tradeoff is configuration duplication — each round repeats similar node definitions — which is mitigable through parameterized node templates in the YAML schema. Dynamic looping would be more compact but introduces runtime graph modification, complicates checkpointing (which iteration state to restore?), and risks unbounded execution without explicit enforcement.

Decision

Adopt the **two-tier gating system** (per-metric floors + composite CI), implement **best-of-N selection with consecutive regression early-stopping**, add **targeted remediation rounds** for partial CI failures, and implement **tiered graceful degradation** with structured quality metadata. Retain the bounded unrolled R1–R4 architecture. Evaluate geometric mean aggregation as an alternative to the arithmetic mean through sensitivity analysis.

Consequences

Positive: Two-tier gating eliminates the compensability vulnerability where one critical metric near zero passes due to strong performance on others. Best-of-N selection with regression detection prevents wasted computation when iterative refinement has peaked. Tiered degradation ensures users always receive the best available output with transparent quality signaling. Structured metadata enables downstream systems to make informed decisions about output reliability.

Negative: Per-metric floors introduce 6 additional thresholds to tune and maintain. Targeted remediation rounds add conditional branching complexity to the YAML workflow. The regression detection heuristic (2 consecutive regressions → stop) may occasionally stop too early if Round 4 would have recovered.

Code-level recommendations

1. **`deep_research.yaml`**: Add `floor_thresholds` to the workflow inputs: `{coverage: 0.35, source_quality: 0.40, agreement: 0.35, verification: 0.40, recency: 0.30}`. Add `when` conditions to each round that check both floor gates and composite CI.
2. **`coverage_confidence_audit` node**: Return both `ci_score` (composite) and individual metric scores as separate output fields. Add a `floor_violations: list[str]` output field listing any metrics below their floor.
3. **`supervisor_decide` node**: Implement `best_round = argmax(ci_scores)` rather than always selecting the latest round. Add `regression_count` tracking and early-stop logic. Add `confidence_level` classification (full/moderate/low/insufficient).
4. **Create `agentic_v2/workflows/lib/ci_calculator.py`**: Centralize the CI formula, floor checks, and sensitivity analysis utilities. Support both arithmetic and geometric mean aggregation with a config switch.
5. **`final_synthesis` node**: Accept `confidence_level` and `quality_metadata` from the supervisor. Embed quality disclaimers in the output when `confidence_level != "full"`.
6. **Add YAML template macros** for round definitions to reduce duplication across R1–R4. Each round should reference a shared `research_round` template with `round_number` as a parameter.

Conclusion

The three ADRs address interconnected reliability concerns in the `agentic-workflows-v2` architecture. **ADR-001** resolves the dual-engine tension through an interface abstraction with planned LangGraph convergence — the key insight being that Airflow's multi-executor precedent validates parallel backends, but Temporal's single-engine success argues for eventual consolidation. **ADR-002** transforms the SmartModelRouter from a basic circuit breaker into a production-grade resilience layer — the highest-impact change is replacing flat 120s rate-limit cooldowns with provider-aware `Retry-After` parsing, which alone yields up to **40× faster capacity recovery**. **ADR-003** hardens the supervisor state machine against the compensability vulnerability in composite CI gating — the two-tier system (per-metric floors + composite threshold) is the central recommendation, drawn directly from healthcare quality measurement methodology where individual measure minimums prevent dangerous metric compensation.

A cross-cutting theme emerges: **the tension between simplicity and correctness**. The current designs chose simple approaches (single composite metric, flat cooldowns, wall clocks, dual engines without conformance testing) that work in development but carry latent production risks. Each ADR provides an incremental hardening path — no recommendation requires a rewrite, and all changes are independently deployable, testable, and reversible.