

# Mother-Harness v3 Addendum

## Synchronous Orchestration, Capability Routing, RAG Contract, and Update Agent Workflow

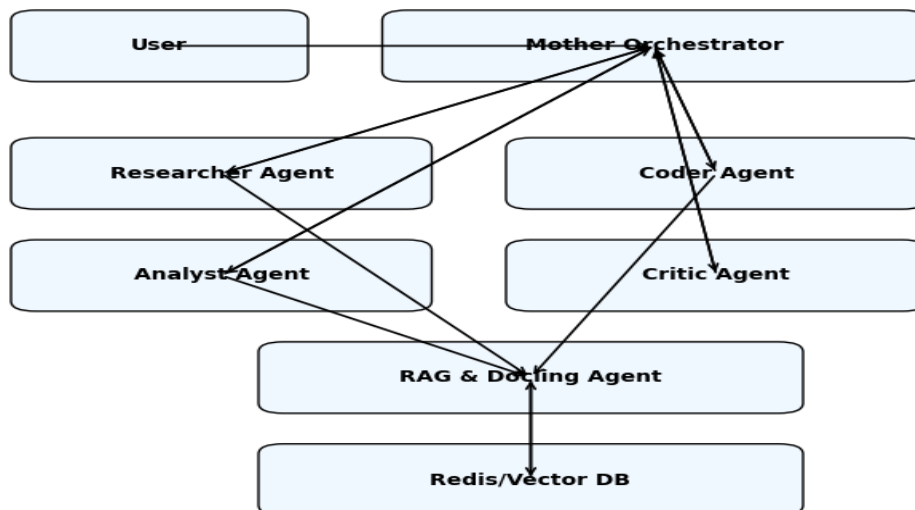
Source-of-truth additions to be merged into “Mother Harness Complete Project v2”.  
Generated December 19, 2025.

This addendum introduces four core upgrades:

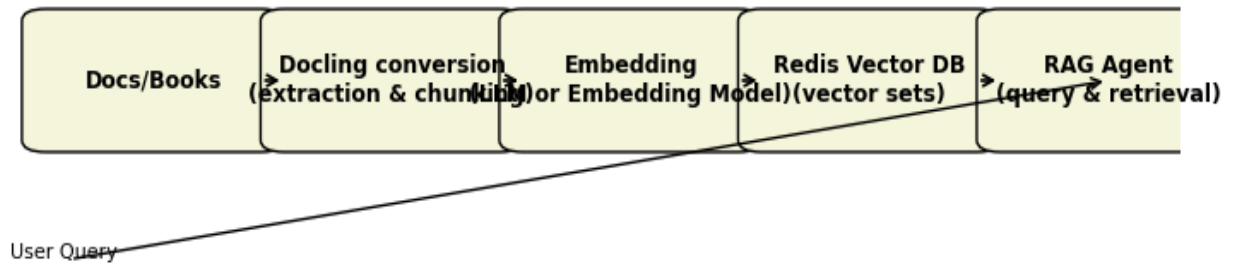
- 1) Execution and Coordination Model (synchronous multi-agent phases, quorums, and termination)
- 2) Capability-Based Routing and Parallelism (model and agent selection as a capability planner)
- 3) Embedding and Retrieval Contract (emb:v1, retrieval report, gating, citations)
- 4) Update Agent Workflow (inventory, evidence ingestion, impact analysis, approvals)

It also outlines how to extend ACI-style contracts beyond the Coder agent.

Reference: Current high-level architecture (from prior doc).



Reference: Current pipeline overview (from prior doc).



# 1. Execution and Coordination Model

Mother-Harness moves from linear “workflow chaining” to a synchronous orchestration model. Execution may be physically asynchronous (workers run in n8n, containers, or services), but coordination is logically synchronous: Mother advances a task through phases using explicit exit conditions, quorums, timeouts, and bounded repair loops.

## 1.1 Concepts

**Run:** a single orchestrated attempt at completing a task. Runs are replayable and checkpointed.

**Phase:** a coordination barrier where one or more agents/workflows run concurrently and Mother decides whether to advance.

**Quorum:** which agent results are required to advance (all/any/majority) and how disagreement is handled.

**Exit Conditions:** deterministic rules for phase completion or termination; prevents “agent loop” failure modes.

**Idempotent Step:** each tool/workflow call can be safely re-run (inputs are hashed; outputs are stored by run\_id).

## 1.2 Phase state machine

```
type Phase =  
| "plan"  
| "execute"  
| "review"  
| "repair"  
| "approve"  
| "finalize"  
| "terminate";  
  
type PhaseExit =  
| { kind: "success"; reason: string }  
| { kind: "needs_repair"; reason: string }  
| { kind: "needs_approval"; reason: string; approval_id: string }  
| { kind: "needs_input"; reason: string; questions: string[] }  
| { kind: "fail"; reason: string };
```

Key rule: Agents never “decide” completion. They produce ResultEnvelopes with artifacts, risk, confidence, and next\_actions. Mother decides phase exit.

## 1.3 Concurrency and quorums

Within a phase, Mother may dispatch multiple agents concurrently. Typical quorums:

- Execute phase: requires the primary worker (e.g., Coder) to return a validated artifact set.
- Review phase: requires Critic + (optional) Security/Update checks depending on policy.
- Arbitration: if agents disagree, Mother applies policy: prefer higher-confidence, higher-evidence outputs; optionally run a tie-breaker model; or request user input.

```
type Quorum =  
| { mode: "all"; agents: string[] }
```

```
| { mode: "any"; agents: string[] }
| { mode: "majority"; agents: string[]; min?: number };

type ArbitrationPolicy = {
prefer: Array<"evidence"|"confidence"|"lower_risk"|"lower_cost">;
tie_breaker?: { role: "critic"|"arbiter"; model_tier: "local"|"cloud" };
timeout_ms: number;
};
```

## 1.4 Termination and anti-stuck guarantees

To avoid getting stuck inside a workflow:

- Every phase has max retries and timeout.
- Repair loops are bounded (e.g., max 2 repairs at local tier, then optional cloud escalation, then stop).
- Mother can emit a needs\_input status when the task cannot proceed without clarification.
- Mother can emit a blocked status when approvals are required.
- Mother always writes a final RunSummary artifact on termination.

```
const Defaults = {
  timeouts: {
    plan_ms: 30_000,
    execute_ms: 20 * 60_000,
    review_ms: 5 * 60_000,
    repair_ms: 10 * 60_000,
  },
  retries: {
    execute: 1,
    repair: 2,
    critic: 1,
    escalate_to_cloud_after_repairs: 2,
  }
};
```

## 2. Capability-Based Routing and Parallelism

Mother selects agents, models, and tools based on required capabilities. Routing is policy-driven and can execute “council style” parallel runs when needed. The default posture remains: local-first, with cloud escalation as a controlled, observable decision.

### 2.1 Capability matrix

```
type Capability =  
| "tool_calling"  
| "vision"  
| "long_context"  
| "strict_json"  
| "coding"  
| "math"  
| "retrieval_synthesis"  
| "policy_reasoning";  
  
type ModelProfile = {  
  model_id: string; // e.g., "qwen2.5:14b" or "qwen2-vl:72b-cloud"  
  provider: "ollama_local" | "ollama_cloud";  
  supports: Capability[];  
  max_ctx_tokens: number;  
  typical_json_reliability: number; // 0..1 (measured)  
  cost_tier: "free" | "flat_cloud";  
};
```

Mother maintains measured reliability stats (per model + per agent role), updated from Critic outcomes and schema validation results.

### 2.2 Routing algorithm

```
type RouteRequest = {  
  role: "research" | "coder" | "critic" | "vision" | "analyst" | "update";  
  requires: Capability[];  
  sensitivity: "public" | "private" | "restricted";  
  target_latency: "fast" | "normal" | "slow";  
};  
  
type RouteDecision = {  
  model: ModelProfile;  
  parallel?: { models: ModelProfile[]; quorum: "any" | "majority"; tie_breaker?:  
    ModelProfile };  
  escalation?: { when: "schema_fail" | "retrieval_fail" | "critic_fail" | "manual"; to:  
    ModelProfile };  
};
```

Parallelism triggers (examples):

- strict\_json required and historical JSON reliability < threshold → run two models in parallel, accept first valid.
- critic disagreement → run an arbiter critic model (possibly cloud) to resolve.
- retrieval gates failing → run “retrieval expansion” in parallel with “query rewrite” (deterministic or LLM-based).

## 2.3 Replay implications

Every routing decision is recorded (selected model, parameters, retrieval inputs, tool outputs). Replay mode can:

- re-run with the same routing (for determinism)
- re-run with a different routing (for evaluation)
- compare outputs and store deltas as artifacts

## 3. Embedding and Retrieval Contract (emb:v1)

This contract defines how documents, notes, and artifacts are chunked, embedded, stored, retrieved, and cited. It is designed to be model-agnostic while remaining audit-friendly and replayable.

### 3.1 Embedding space

embed\_space is the stable identifier for a semantic vector space. For this system:  
embed\_space = "emb:v1"

All embedded records MUST store:

- embed\_space ("emb:v1")
- embedding\_model\_id (audit only, not for routing)
- embedding\_dim
- created\_at

```
type EmbeddedRecord = {  
  id: string;  
  library_id: string;  
  document_id: string;  
  chunk_id: string;  
  
  text: string; // canonical chunk text  
  metadata: Record<string, unknown>;  
  
  embed_space: "emb:v1";  
  embedding_model_id: string;  
  embedding_dim: number;  
  embedding: number[];  
  
  created_at: string; // ISO-8601  
};
```

### 3.2 Chunking rules

Default chunking is Docling-hierarchy first (sections/headers), with token targeting:

- target\_tokens: 450
- overlap\_tokens: 80
- preserve\_section\_path for citations and adjacency fetch.

### 3.3 Retrieval API and outputs

```
type RetrievalQuery = {  
  run_id: string;  
  query: string;  
  embed_space: "emb:v1";  
  topk: number;  
  method: "vector" | "hybrid";  
  filters?: {  
    library_id?: string;  
    document_ids?: string[];  
  };  
};
```

```

tags?: string[];
doc_types?: string[];
};

type RetrievalHit = {
library_id: string;
document_id: string;
chunk_id: string;
rank: number;
score: number; // 0..1 normalized when possible
token_count?: number;
section_path?: string[];
page?: number;
};

```

All RAG responses MUST include a RetrievalReport artifact. All agent answers based on RAG MUST include Citations pointing to hits used.

### 3.4 RAG quality gates (deterministic)

```

type RetrievalGates = {
min_top1_score: number; // e.g., 0.30
min_total_tokens: number; // e.g., 900
min_unique_documents?: number; // e.g., 2
};

type GateResult = { pass: boolean; reasons: string[] };

```

If gates fail, Mother attempts these fallbacks in order (policy may alter order):

- 1) increase topk
- 2) fetch adjacent chunks (same section\_path, page window)
- 3) enable hybrid retrieval (keyword + vector)
- 4) query rewrite (deterministic first; LLM-assisted second)
- 5) escalate model tier for synthesis only (not retrieval)
- 6) mark needs\_input if still insufficient

### 3.5 Citation contract

```

type Citation = {
library_id: string;
document_id: string;
chunk_id: string;
page?: number;
section_path?: string[];
quote?: string; // <= 240 chars
};

```

## 4. Update Agent Workflow

The Update Agent monitors deployed software and relevant dependencies, detects new releases, ingests upstream documentation into the library, and produces an evidence-backed upgrade recommendation that can flow through approvals and replay.

### 4.1 Inventory schema

```
type SoftwareInventoryItem = {
  id: string;

  name: string; // e.g., "redis", "ollama", "n8n", "open-webui"
  kind: "container"|"service"|"binary"|"library"|"model";
  current_version: string;
  deployment: {
    host: string; // core1/core2/core3
    location: string; // container name, path, repo, etc.
    criticality: "low"|"med"|"high";
  };

  upstream: {
    type: "github"|"dockerhub"|"website"|"pypi"|"npm"|"other";
    ref: string; // repo slug, image name, URL
  };

  update_policy: {
    channel: "stable"|"lts"|"nightly";
    auto_update: boolean;
    requires_approval: boolean;
  };
};
```

### 4.2 Evidence ingestion

When a new version is detected, the agent fetches and stores evidence:

- release notes / changelog
- migration guides
- breaking change notes
- security advisories (when applicable)

All evidence is ingested via Docling, chunked, embedded into emb:v1, and linked to the inventory item.

```
type UpdateEvidence = {
  inventory_item_id: string;
  version_new: string;

  sources: Array<{
    kind: "release_notes"|"changelog"|"migration"|"security"|"docs";
    title: string;
    fetched_at: string;
    stored_document_id: string; // in the library
  }>;
};
```

```
};
```

## 4.3 Impact analysis and recommendation

```
type UpdateRecommendation = {  
  inventory_item_id: string;  
  version_current: string;  
  version_new: string;  
  
  summary: string;  
  breaking_changes: string[];  
  migration_steps: string[];  
  risk_score: number; // 0..100  
  confidence: number; // 0..1  
  
  recommendation: "upgrade_now"|"upgrade_later"|"hold"|"investigate";  
  rationale_citations: Citation[];  
  
  requires_approval: boolean;  
  suggested_window?: string; // e.g., "Sat 02:00-04:00 ET"  
  rollback_plan?: string;  
};
```

## 4.4 Scheduling, approvals, and replay

Update checks run on a schedule (e.g., daily for critical items, weekly for low).

Recommendations can trigger:

- Approval requests (required\_approval true OR risk\_score >= threshold OR criticality high)
- Automatic creation of a “Proposed Upgrade Run” (dry-run plan) for replay and review
- Post-upgrade verification tasks (smoke tests, metrics checks)

## 4.5 Exit conditions

The Update Agent workflow terminates with one of:

- upgrade\_now / upgrade\_later / hold / investigate

and always emits an UpdateRecommendation artifact plus a RunSummary.

# 5. Extending ACI-Style Contracts to Other Agents

You noted we may want to expand other agents “the same way as Coder.” The general pattern:

- constrain each agent to a small, explicit action set
- require specific artifacts per phase
- validate artifacts to schema before advancing phases

Agent contract matrix (starter):

Agent	Primary Actions (bounded)	Required Artifacts (examples)
Research	search, fetch_source, extract_claims,	Claims[], Citations[], OpenQuestions[], RetrievalReport
Vision	analyze_image, extract_text_ocr, label_image,	MissionFindings JSON, ImageRefs, Citations/Links
Analyst	run_notebook/script, compute_stats,	AnalyzeReport, DataFrames/CSV, Provenance
Critic	validate_schema, check_grounding, critique,	Verify, FailReasons[], RequiredFixes[]
Update	scan_inventory, check_upstream, ingest_data,	UpdateRecommendations, EvidenceRefs, RollbackPlan

## Appendix: Where the Coder ACI spec goes next

Per your note: the next iteration should expand the Coder Agent ACI into a full contract section (allowed actions, required artifacts, approvals, and exit criteria), then mirror that structure for the other agents above.