

Mother-Harness v4 Addendum: Robustness & Operational Safety

- Document Purpose
 - 1. Explicit Termination Semantics
 - 2. Conflict Resolution Policy
 - 3. Context Budget Ownership
 - 4. Deterministic Tool Registry
 - 5. Resource Budget Guards
 - 6. Artifact Lifecycle & Garbage Collection
 - 7. Library Access Control Policy
- Implementation Roadmap
- Validation Checklist
- Appendix A: Redis Schema Summary
- Appendix B: Integration with Existing v3 Features
- Document Changelog

Mother-Harness v4 Addendum: Robustness & Operational Safety

Version: v4
Date: December 19, 2025
Status: Production-Ready Specification
Applies to: Mother-Harness Complete Project v2 + v3 Addendum

Document Purpose

This addendum defines seven critical robustness features that eliminate ambiguity, prevent failure modes, and establish operational safety boundaries. All features are **mechanical** (no philosophy), **enforceable** (deterministic rules), and **observable** (audit trails).

Target audience: Systems engineers implementing Mother-Harness in production environments.

1. Explicit Termination Semantics

Problem Statement

Current design: runs stop, but **why they stopped** is implicit. This causes: - Debugging nightmares ("it just died") - No clear distinction between success and various failure modes - Replay cannot explain termination - Unclear when to retry vs. escalate vs. give up

Solution: Mandatory Termination Reason

Every run **MUST** terminate with exactly one reason code.

Type Definition

```
type TerminationReason =  
  | "success"           // All phases completed, artifacts  
valid  
  | "approval_denied"   // User/policy rejected a gated  
action  
  | "policy_violation"  // Agent attempted disallowed  
action  
  | "retries_exhausted" // Max retries exceeded in any
```

```

phase
    | "timeout" // Phase or run timeout exceeded
    | "insufficient_evidence" // RAG gates failed, retrieval
inadequate
    | "conflicting_agents" // Agents disagree, no tie-breaker
succeeded
    | "user_cancelled" // Explicit user termination
    | "budget_exhausted" // Resource limits exceeded (see
Section 5)
    | "blocked" // Cannot proceed without
input/approval
    | "catastrophic_error"; // Unrecoverable system error

type TerminationRecord = {
    run_id: string;
    reason: TerminationReason;
    phase_at_termination: Phase;
    timestamp: string; // ISO-8601

    // Context
    details: string; // Human-readable explanation
    contributing_factors: string[]; // e.g., ["low retrieval score",
"3 repair loops"]

    // Recommendations
    can_retry: boolean;
    suggested_action: "retry" | "escalate_model" | "broaden_scope" |
"user_input" | "abandon";

    // Audit
    logged_by: "mother" | "agent_id";
    final_artifacts: string[]; // artifact_ids produced before
termination
};

```

Enforcement Rules

1. **Mandatory field:** Every run writes a TerminationRecord to Redis before closing
2. **Single source:** Only Mother (orchestrator) writes termination records
3. **Immutable:** Once written, termination record cannot be modified
4. **Indexed:** Redis index on reason for analytics

Storage Schema

```

// RedisJSON key
termination:{run_id}

// Example
{
    "run_id": "run-abc123",
    "reason": "retries_exhausted",
    "phase_at_termination": "repair",
    "timestamp": "2025-12-19T22:30:45.123Z",
    "details": "Coder agent failed to produce valid CodePatch after 3
repair loops",
    "contributing_factors": [
        "Test suite failed with 5 errors",
        "Critic rejected patch twice",
        "Local model (qwen2.5:14b) struggled with complex refactoring"
    ],
    "can_retry": true,
    "suggested_action": "escalate_model",
    "logged_by": "mother",
    "final_artifacts": ["critic-report-1", "test-results-2"]
}

```

```
}
```

Integration with ResultEnvelope

```
// Final ResultEnvelope includes termination
type ResultEnvelope = {
  // ... existing fields ...
  termination?: TerminationRecord; // Present only on final
envelope
};
```

Replay Implications

Replay logs MUST show: - Why the run stopped (not just where) -
What suggested action was - Which artifacts existed at termination

This enables: - Retry with escalation ("last time failed on local model,
use cloud") - Root cause analysis - Policy tuning

2. Conflict Resolution Policy

Problem Statement

When agents disagree (e.g., Critic rejects but Coder is confident), **who decides** and **how**?

Current design supports quorums but doesn't define arbitration rules.

Solution: Deterministic Conflict Resolution Table

Conflict Types

```
type ConflictType =
  | "critic_vs_coder"           // Critic fails, Coder confident
  | "low_confidence_output"    // Agent produced output but
confidence < threshold
  | "competing_proposals"      // Multiple agents suggest
different approaches
  | "evidence_contradiction"    // Research finds conflicting
sources
  | "test_vs_review"           // Tests pass but Critic flags
issues
  | "retrieval_ambiguity";      // RAG returns conflicting chunks

type ConflictResolutionStrategy =
  | "prefer_higher_confidence" // Pick agent with higher
confidence score
  | "prefer_higher_evidence"   // Pick agent with more
citations/artifacts
  | "run_arbiter"              // Spawn tie-breaker agent/model
  | "escalate_model"           // Re-run primary agent with
cloud model
  | "request_user_input"       // Human arbitration
  | "fail_safe"                // Pick lowest-risk option
  | "terminate_blocked";       // Cannot resolve, stop run
```

Resolution Policy Table

```
const ConflictResolutionPolicy: Record<ConflictType,
ConflictResolutionStrategy[]> = {
  "critic_vs_coder": [
    "run_arbiter",           // Run Critic again with cloud model
```

```

        "escalate_model",          // Re-run Coder with cloud model
        "request_user_input"      // If still conflicted
    ],

    "low_confidence_output": [
        "escalate_model",          // Try cloud model
        "prefer_higher_evidence",  // If multiple attempts, pick one
        "request_user_input"       // User decides if all attempts low
    ],
    with most artifacts
    confidence

    "competing_proposals": [
        "prefer_higher_confidence",
        "run_arbiter",             // Arbiter agent evaluates both
        "request_user_input"
    ],

    "evidence_contradiction": [
        "prefer_higher_evidence",  // More citations wins
        "run_arbiter",             // Research agent investigates
        "terminate_blocked"        // Mark "needs_investigation"
    ],
    discrepancy

    "test_vs_review": [
        "prefer_higher_evidence",  // Tests pass = strong evidence
        "run_arbiter",             // Senior Critic review
        "fail_safe"                // Reject if safety-critical
    ],

    "retrieval_ambiguity": [
        "escalate_model",          // Better model for synthesis
        "request_user_input",      // User clarifies intent
        "terminate_blocked"        // Mark insufficient_evidence
    ]
};

```

Conflict Resolution Artifact

```

type ConflictResolution = {
  conflict_id: string;
  run_id: string;
  conflict_type: ConflictType;

  // Parties
  agents_involved: string[];
  their_outputs: {
    agent_id: string;
    confidence: number;
    evidence_count: number;
    artifact_ids: string[];
  }[];

  // Resolution
  strategy_applied: ConflictResolutionStrategy;
  winner_agent?: string;
  rationale: string;

  // Outcome
  resolution_successful: boolean;
  final_decision: "accepted" | "rejected" | "escalated" | "blocked";

  // Timestamps
  detected_at: string;
  resolved_at: string;

```

```
        resolution_duration_ms: number;
    };
```

Enforcement Rules

1. Mother detects conflicts by comparing agent outputs in review phase
2. Mother applies strategies in order until one succeeds
3. All conflicts logged to conflict:{conflict_id} in Redis
4. Runs can have multiple conflicts; each gets own resolution record

Example Resolution Flow

```
// Scenario: Critic rejects Coder's patch

1. Mother detects conflict: "critic_vs_coder"
2. Applies first strategy: "run_arbiter"
   → Spawns Critic agent with cloud model (deepseek-v3.1)
3. Cloud Critic returns: still rejects
4. Applies second strategy: "escalate_model"
   → Re-runs Coder with cloud model (devstral-2:123b)
5. Cloud Coder produces new patch
6. Run Critic on new patch → passes
7. Conflict resolved, continue run

ConflictResolution artifact:
{
  "conflict_type": "critic_vs_coder",
  "strategy_applied": "escalate_model",
  "winner_agent": "coder",
  "rationale": "Cloud model (devstral-2:123b) produced patch that
passed cloud Critic review",
  "resolution_successful": true,
  "final_decision": "accepted"
}
```

3. Context Budget Ownership

Problem Statement

LLMs have context windows. Naive truncation causes hallucinations.
Who is responsible for fitting everything into the context window?

Solution: Mother Owns Context Composition

Core Rules

```
const ContextOwnershipRules = {
  agents_never_truncate: true,      // Agents receive full context
OR error
  mother_composes_context: true,    // Mother assembles context
per agent
  summarization_is_explicit: true,   // All summarization produces
artifacts
  summarization_is_versioned: true  // Summaries have versions and
hashes
};
```

Context Budget

```
type ContextBudget = {
  max_tokens: number;              // e.g., 100_000 for 200k window
};
```

```

model
    reserved_for_system: number;    // e.g., 5_000 for system prompt
    reserved_for_output: number;    // e.g., 20_000 for agent
response
    available_for_context: number;  // max_tokens - reserved
};

// Example for qwen2.5:32b (128k context)
const LocalModelBudget: ContextBudget = {
    max_tokens: 128_000,
    reserved_for_system: 3_000,
    reserved_for_output: 15_000,
    available_for_context: 110_000
};

// Example for cloud model (200k context)
const CloudModelBudget: ContextBudget = {
    max_tokens: 200_000,
    reserved_for_system: 5_000,
    reserved_for_output: 25_000,
    available_for_context: 170_000
};

```

Context Composition Algorithm

Mother builds context in priority order:

```

type ContextPriority = {
    section: string;
    content: string;
    tokens: number;
    priority: number; // Higher = more important
    required: boolean; // If true, run fails if cannot fit
};

const ContextCompositionOrder: ContextPriority[] = [
    {
        section: "task_definition",
        content: "Original user query + goals",
        priority: 100,
        required: true
    },
    {
        section: "current_phase",
        content: "Phase context + exit criteria",
        priority: 95,
        required: true
    },
    {
        section: "recent_messages",
        content: "Last 10 messages in conversation",
        priority: 90,
        required: false
    },
    {
        section: "retrieval_hits",
        content: "RAG results (if applicable)",
        priority: 85,
        required: false
    },
    {
        section: "session_summaries",
        content: "Past session summaries (Tier 2 memory)",
        priority: 80,
        required: false
    },
];

```

```

{
  section: "project_knowledge",
  content: "Project-specific context",
  priority: 75,
  required: false
},
{
  section: "previous_artifacts",
  content: "Artifacts from earlier phases",
  priority: 70,
  required: false
}
];

```

Summarization Artifact

When Mother must summarize to fit budget, it produces:

```

type ContextSummary = {
  summary_id: string;
  run_id: string;
  created_at: string;

  // Provenance
  source_sections: string[];           // Which sections were
summarized
  source_token_count: number;          // Original size
  summary_token_count: number;        // Compressed size
  compression_ratio: number;          // source / summary

  // Content
  summary_text: string;

  // Method
  summarization_method: "extractive" | "llm_generated" | "hybrid";
  model_used?: string;                // If LLM-generated

  // Quality
  information_preserved: string[];     // Key facts retained
  information_lost: string[];          // What was dropped (if
known)

  // Versioning
  summary_version: string;             // e.g., "summary:v1"
  content_hash: string;               // SHA-256 of summary_text
};

```

Enforcement Rules

1. **No silent truncation:** If context exceeds budget, Mother either:
 - Summarizes (produces ContextSummary artifact)
 - Escalates to larger context model
 - Fails with termination_reason: "context_budget_exceeded"
2. **Summarization is observable:** Every summarization logged to activity stream
3. **Agents validate context:** Agents check they received all required sections; if not, return error
4. **Replay includes summaries:** Replay bundle contains ContextSummary artifacts

Example Scenario

```

// Run with large retrieval results

1. Mother composes context for Research agent
2. Calculates token counts:
  - task_definition: 500 tokens (required)
  - current_phase: 300 tokens (required)
  - recent_messages: 5,000 tokens
  - retrieval_hits: 85,000 tokens (too big!)
  - session_summaries: 10,000 tokens

Total: 100,800 tokens > 100,000 budget

3. Mother applies compression:
  - Keeps task_definition + current_phase (required)
  - Keeps recent_messages (high priority)
  - Summarizes retrieval_hits:
    → Extractive summarization (top 20 chunks → 15,000 tokens)
    → Produces ContextSummary artifact
  - Includes session_summaries

New total: 30,800 tokens ✓

4. Research agent receives context + ContextSummary artifact
5. Agent output includes reference to summary:
  "Note: Context included summarized retrieval (see summary-
abc123)"

```

4. Deterministic Tool Registry

Problem Statement

“Tool-first” philosophy is mentioned throughout but not mechanically enforced. Agents may call LLMs when a deterministic tool exists.

Solution: Formal Tool Registry with Routing

Tool Schema

```

type DeterministicTool = {
  tool_id: string; // e.g., "git_status",
  "resize_image"
  tool_name: string; // Display name
  version: string; // e.g., "1.0.0"

  // Categorization
  category: "file_ops" | "git" | "compute" | "ocr" | "format" |
"validation" | "db" | "network";

  // Contract
  input_schema: JSONSchema; // What inputs it requires
  output_schema: JSONSchema; // What it returns
  side_effects: "read_only" | "idempotent_write" | "destructive";

  // Routing
  handles_patterns: string[]; // Regex patterns it can
  handle
  keywords: string[]; // Keywords that trigger this
  tool
  priority: number; // Higher = try first (1-100)

  // Execution
  timeout_ms: number; // Max execution time
  max_retries: 0 | 1 | 2; // Retry on transient failures

```

```

// Policy
approval_required: boolean;
allowed_roles: string[];           // Which agents can use it

// Cost
cost_tier: "free";                // Tools are always free

// Implementation
entrypoint: string;               // Function/script to execute
};

```

Tool Registry Document

```

type ToolRegistry = {
  registry_version: "tools:v1";
  updated_at: string;

  tools: DeterministicTool[];

  // Global settings
  routing_order: string[];        // tool_ids in priority order
  default_timeout_ms: number;
  allow_fallback_to_llm: boolean; // If tool fails, try LLM?

  // Statistics (measured)
  tool_success_rates: Record<string, number>; // tool_id → success
rate
};

```

Example Tools

```

const ExampleTools: DeterministicTool[] = [
  {
    tool_id: "git_status",
    tool_name: "Git Status Check",
    version: "1.0.0",
    category: "git",
    input_schema: {
      type: "object",
      properties: {
        repo_path: { type: "string" }
      },
      required: ["repo_path"]
    },
    output_schema: {
      type: "object",
      properties: {
        branch: { type: "string" },
        modified_files: { type: "array", items: { type: "string" } },
        staged_files: { type: "array", items: { type: "string" } }
      }
    },
    side_effects: "read_only",
    handles_patterns: [
      /git status/i,
      /check.*git/i,
      /what.*changed.*repo/i
    ],
    keywords: ["git", "status", "repository", "changes"],
    priority: 95,
    timeout_ms: 5000,
    max_retries: 1,
    approval_required: false,
    allowed_roles: ["coder", "critic", "update"],
  },
];

```

```

    cost_tier: "free",
    entrypoint: "tools/git/status.ts"
  },
  {
    tool_id: "resize_image",
    tool_name: "Image Resize",
    version: "1.0.0",
    category: "format",
    input_schema: {
      type: "object",
      properties: {
        input_path: { type: "string" },
        width: { type: "number" },
        height: { type: "number" }
      },
      required: ["input_path", "width", "height"]
    },
    output_schema: {
      type: "object",
      properties: {
        output_path: { type: "string" },
        original_size: { type: "object" },
        new_size: { type: "object" }
      }
    },
    side_effects: "idempotent_write",
    handles_patterns: [
      /resize.*image.*(\d+)x(\d+)/i,
      /scale.*image/i
    ],
    keywords: ["resize", "image", "scale", "dimensions"],
    priority: 90,
    timeout_ms: 10000,
    max_retries: 0,
    approval_required: false,
    allowed_roles: ["vision", "analyst"],
    cost_tier: "free",
    entrypoint: "tools/image/resize.ts"
  },
  {
    tool_id: "run_tests",
    tool_name: "Test Suite Runner",
    version: "1.0.0",
    category: "validation",
    input_schema: {
      type: "object",
      properties: {
        repo_path: { type: "string" },
        test_command: { type: "string" },
        env_vars: { type: "object" }
      },
      required: ["repo_path", "test_command"]
    },
    output_schema: {
      type: "object",
      properties: {
        exit_code: { type: "number" },
        stdout: { type: "string" },
        stderr: { type: "string" },
        duration_ms: { type: "number" },
        tests_run: { type: "number" },
        tests_passed: { type: "number" },
        tests_failed: { type: "number" }
      }
    }
  }

```

```

    },
    side_effects: "read_only",
    handles_patterns: [
      /run.*tests/i,
      /execute.*test.*suite/i,
      /npm.*test/i,
      /pytest/i
    ],
    keywords: ["test", "suite", "jest", "pytest", "mocha"],
    priority: 100, // Always run tests via tool, not LLM
    timeout_ms: 300000, // 5 minutes
    max_retries: 0,
    approval_required: false,
    allowed_roles: ["coder", "critic"],
    cost_tier: "free",
    entrypoint: "tools/test/runner.ts"
  },

  {
    tool_id: "ocr_extract",
    tool_name: "OCR Text Extraction",
    version: "1.0.0",
    category: "ocr",
    input_schema: {
      type: "object",
      properties: {
        image_path: { type: "string" },
        language: { type: "string", default: "eng" }
      },
      required: ["image_path"]
    },
    output_schema: {
      type: "object",
      properties: {
        text: { type: "string" },
        confidence: { type: "number" }
      }
    },
    side_effects: "read_only",
    handles_patterns: [
      /extract.*text.*image/i,
      /ocr/i,
      /read.*text.*from/i
    ],
    keywords: ["ocr", "extract", "text", "image"],
    priority: 85,
    timeout_ms: 30000,
    max_retries: 1,
    approval_required: false,
    allowed_roles: ["vision", "librarian"],
    cost_tier: "free",
    entrypoint: "tools/ocr/tesseract.ts"
  }
];

```

Routing Algorithm

```

class ToolRouter {
  async tryToolsFirst(query: string, role: string):
  Promise<ToolResult | null> {
    // 1. Get tools sorted by priority
    const eligibleTools = this.registry.tools
      .filter(t => t.allowed_roles.includes(role))
      .sort((a, b) => b.priority - a.priority);

    // 2. Check pattern matches

```

```

    for (const tool of eligibleTools) {
      if (this.matchesPattern(query, tool)) {
        try {
          const result = await this.executeTool(tool, query);

          // Log success
          await this.updateSuccessRate(tool.tool_id, true);

          return {
            handled: true,
            tool_id: tool.tool_id,
            result: result,
            cost: 0 // Tools are free!
          };
        } catch (error) {
          // Log failure
          await this.updateSuccessRate(tool.tool_id, false);

          // Try next tool or fallback to LLM
          continue;
        }
      }
    }

    // 3. No tool handled it
    return null;
  }

  private matchesPattern(query: string, tool: DeterministicTool):
boolean {
    // Check regex patterns
    for (const pattern of tool.handles_patterns) {
      if (pattern.test(query)) return true;
    }

    // Check keyword presence
    const queryLower = query.toLowerCase();
    const keywordMatches = tool.keywords.filter(kw =>
      queryLower.includes(kw.toLowerCase())
    );

    return keywordMatches.length >= 2; // At least 2 keywords
  }
}

```

Integration with Mother

```

// In Mother's routing logic

async route(query: string, role: string): Promise<ExecutionPlan> {
  // 1. Try deterministic tools FIRST
  const toolResult = await toolRouter.tryToolsFirst(query, role);

  if (toolResult?.handled) {
    // Tool handled it - no LLM needed!
    await activityStream.log({
      agent: "mother",
      type: "tool_success",
      message: `Handled by tool: ${toolResult.tool_id}`,
      metadata: { cost: 0, tool_id: toolResult.tool_id }
    });

    return {
      type: "tool_only",
      result: toolResult.result,

```

```

        cost: 0
    };
}

// 2. Fall back to LLM-based agent
const agent = await this.selectAgent(role, query);
return {
    type: "agent",
    agent: agent,
    estimated_cost: this.estimateCost(agent)
};
}

```

Storage Schema

```

// Tool registry
registry:tools

// Tool statistics
tool_stats:{tool_id}
{
    "success_count": 142,
    "failure_count": 3,
    "success_rate": 0.979,
    "avg_execution_ms": 234,
    "last_used": "2025-12-19T23:15:00Z"
}

```

5. Resource Budget Guards

Problem Statement

Ollama Cloud is \$20/month unlimited, but: - Runaway loops can still burn cloud credits if you upgrade tiers - Local resources (CPU, RAM, disk) are finite - Need circuit breakers to prevent DoS

Solution: Multi-Level Budget System

Budget Types

```

type ResourceBudget = {
    // Per-run limits
    per_run: {
        max_cloud_calls: number;           // e.g., 50
        max_total_tokens: number;          // e.g., 500,000
        max_duration_sec: number;          // e.g., 1,800 (30 min)
        max_tool_calls: number;            // e.g., 100
        max_retrieval_queries: number;     // e.g., 20
        max_parallel_agents: number;       // e.g., 5
    };

    // Per-user daily limits
    per_user_daily: {
        max_runs: number;                  // e.g., 50
        max_cloud_calls: number;           // e.g., 200
        max_total_tokens: number;          // e.g., 2,000,000
    };

    // Global system limits
    global: {
        max_concurrent_runs: number;       // e.g., 10
        max_queue_depth: number;           // e.g., 100
    };
}

```

```

        max_redis_memory_mb: number;    // e.g., 16,000 (16GB)
        max_docling_jobs: number;       // e.g., 5
    };

    // Warning thresholds (before hard limits)
    warnings: {
        per_run_tokens_warning: number;    // e.g., 400,000 (80% of
max)
        per_user_daily_runs_warning: number; // e.g., 40 (80% of max)
        global_concurrent_warning: number;  // e.g., 8 (80% of max)
    };
};

```

Default Budget Configuration

```

const DefaultBudget: ResourceBudget = {
    per_run: {
        max_cloud_calls: 50,
        max_total_tokens: 500_000,
        max_duration_sec: 1_800,    // 30 minutes
        max_tool_calls: 100,
        max_retrieval_queries: 20,
        max_parallel_agents: 5
    },

    per_user_daily: {
        max_runs: 50,
        max_cloud_calls: 200,
        max_total_tokens: 2_000_000
    },

    global: {
        max_concurrent_runs: 10,
        max_queue_depth: 100,
        max_redis_memory_mb: 16_000,
        max_docling_jobs: 5
    },

    warnings: {
        per_run_tokens_warning: 400_000,
        per_user_daily_runs_warning: 40,
        global_concurrent_warning: 8
    }
};

```

Budget Tracking

```

type BudgetCounter = {
    run_id: string;
    user_id: string;

    // Current consumption
    cloud_calls: number;
    total_tokens: number;
    duration_sec: number;
    tool_calls: number;
    retrieval_queries: number;
    parallel_agents: number;

    // Timestamps
    started_at: string;
    last_updated: string;

    // Status
    warnings_triggered: string[];
    limits_exceeded: string[];
};

```

```

};

type BudgetExhausted = {
  run_id: string;
  resource: "cloud_calls" | "tokens" | "duration" | "tools" |
"retrieval" | "parallel" | "queue";
  limit: number;
  consumed: number;
  timestamp: string;

  // Termination
  termination_reason: "budget_exhausted";
  termination_details: string;
};

```

Enforcement Rules

1. **Increment on every action:**
 - LLM call → increment cloud_calls + total_tokens
 - Tool execution → increment tool_calls
 - RAG query → increment retrieval_queries
 - Agent spawn → increment parallel_agents
2. **Check before action:**
 - Before dispatching, check if limit would be exceeded
 - If yes, terminate with budget_exhausted
3. **Warning notifications:**
 - At 80% of limit, log warning to activity stream
 - User can acknowledge warning or adjust budget
4. **Grace period:**
 - If limit exceeded mid-operation, allow operation to complete
 - But block next operation

Redis Storage

```

// Per-run budget
budget:run:{run_id}
{
  "cloud_calls": 23,
  "total_tokens": 145_234,
  "duration_sec": 456,
  "tool_calls": 12,
  "retrieval_queries": 5,
  "parallel_agents": 2
}

// Per-user daily budget
budget:user:{user_id}:daily:{date}
{
  "runs": 12,
  "cloud_calls": 87,
  "total_tokens": 567_890
}

// Global counters
budget:global
{
  "concurrent_runs": 7,
  "queue_depth": 34,
  "redis_memory_mb": 8_432,
  "docling_jobs": 2
}

```

Circuit Breaker

```

class CircuitBreaker {

```

```

    async checkGlobalLimits(): Promise<boolean> {
        const global = await redis.json.get("budget:global");

        // Check concurrent runs
        if (global.concurrent_runs >=
DefaultBudget.global.max_concurrent_runs) {
            await activityStream.log({
                agent: "system",
                type: "circuit_breaker",
                message: "Max concurrent runs exceeded, rejecting new runs",
                metadata: { limit: DefaultBudget.global.max_concurrent_runs
}

            });

            return false; // Circuit open
        }

        // Check queue depth
        if (global.queue_depth >= DefaultBudget.global.max_queue_depth)
{
            await activityStream.log({
                agent: "system",
                type: "circuit_breaker",
                message: "Queue full, rejecting new runs",
                metadata: { limit: DefaultBudget.global.max_queue_depth }
            });

            return false; // Circuit open
        }

        return true; // Circuit closed, accept run
    }
}

```

Integration Example

```

// In Mother's execute method

async execute(run: Run): Promise<void> {
    // 1. Check circuit breaker
    if (!await circuitBreaker.checkGlobalLimits()) {
        await this.terminate(run.id, {
            reason: "budget_exhausted",
            details: "System at capacity, try again later"
        });
        return;
    }

    // 2. Initialize budget tracker
    await budgetTracker.init(run.id, run.user_id);

    // 3. Execute phases
    for (const phase of phases) {
        // Check per-run budget before each action
        const canProceed = await budgetTracker.checkLimit(
            run.id,
            "cloud_calls"
        );

        if (!canProceed) {
            await this.terminate(run.id, {
                reason: "budget_exhausted",
                resource: "cloud_calls",
                details: `Exceeded max
${DefaultBudget.per_run.max_cloud_calls} cloud calls`
            });
        }
    }
}

```

```

        return;
    }

    // Execute phase...
    await budgetTracker.increment(run.id, "cloud_calls");
}
}

```

6. Artifact Lifecycle & Garbage Collection

Problem Statement

Runs accumulate artifacts. Without retention policy: - Redis bloats indefinitely - No compliance with data retention rules - Users lose important runs by accident

Solution: Explicit Retention Policies + GC

Retention Policy Schema

```

type ArtifactRetention = {
  // Default retention
  default_ttl_days: number; // e.g., 90

  // By run status
  by_status: {
    success: {
      keep_days: number; // e.g., 180
      archive_after_days?: number; // e.g., 90
    };
    failed: {
      keep_days: number; // e.g., 30
    };
    cancelled: {
      keep_days: number; // e.g., 7
    };
    blocked: {
      keep_days: number; // e.g., 60
    };
  };

  // Exceptions (never delete)
  exceptions: {
    approved_artifacts: boolean; // true = keep forever
    golden_tasks: boolean; // true = keep forever
    user_starred: boolean; // true = keep forever
    compliance_required: boolean; // true = keep forever
  };

  // Archive settings
  archive: {
    enabled: boolean;
    destination: "s3" | "filesystem" | "compressed_redis";
    compress: boolean;
  };
};

```

Default Retention Configuration

```

const DefaultRetention: ArtifactRetention = {
  default_ttl_days: 90,

```

```

    by_status: {
      success: {
        keep_days: 180,
        archive_after_days: 90
      },
      failed: {
        keep_days: 30
      },
      cancelled: {
        keep_days: 7
      },
      blocked: {
        keep_days: 60
      }
    },

    exceptions: {
      approved_artifacts: true,
      golden_tasks: true,
      user_starred: true,
      compliance_required: true
    },

    archive: {
      enabled: true,
      destination: "compressed_redis",
      compress: true
    }
  };

```

Artifact Metadata

```

type ArtifactMetadata = {
  artifact_id: string;
  run_id: string;
  created_at: string;

  // Lifecycle
  status: "active" | "archived" | "scheduled_deletion";
  expires_at?: string; // When it will be deleted
  archived_at?: string;

  // Protection
  protected: boolean; // Cannot be deleted
  protection_reason?: "approved" | "golden_task" | "user_starred" |
    "compliance";

  // User controls
  user_starred: boolean;
  user_notes?: string;
};

```

Garbage Collector Service

```

class ArtifactGarbageCollector {
  private scanFrequency = "daily";
  private notifyBeforeDeleteDays = 7;

  async scan(): Promise<GCScanReport> {
    const now = new Date();
    const artifacts = await this.getAllArtifacts();

    const report: GCScanReport = {
      scanned_count: artifacts.length,
      scheduled_deletion: 0,
      archived: 0,

```

```

        protected: 0,
        errors: []
    };

    for (const artifact of artifacts) {
        // Skip protected
        if (artifact.protected) {
            report.protected++;
            continue;
        }

        // Check if expired
        if (artifact.expires_at && new Date(artifact.expires_at) <
now) {
            try {
                await this.deleteArtifact(artifact);
                report.scheduled_deletion++;
            } catch (error) {
                report.errors.push({
                    artifact_id: artifact.artifact_id,
                    error: error.message
                });
            }
        }

        // Check if should be archived
        const run = await this.getRun(artifact.run_id);
        const agedays = this.daysSince(artifact.created_at);
        const archiveThreshold =
DefaultRetention.by_status[run.status]?.archive_after_days;

        if (archiveThreshold && agedays >= archiveThreshold &&
artifact.status === "active") {
            try {
                await this.archiveArtifact(artifact);
                report.archived++;
            } catch (error) {
                report.errors.push({
                    artifact_id: artifact.artifact_id,
                    error: error.message
                });
            }
        }
    }

    return report;
}

async deleteArtifact(artifact: ArtifactMetadata): Promise<void> {
    // 1. Notify user if starred or important
    if (artifact.user_starred) {
        await this.notifyUserBeforeDeletion(artifact);
        // Give user 7 days to object
        await this.scheduleActualDeletion(artifact, 7);
        return;
    }

    // 2. Delete from Redis
    await redis.del(`artifact:${artifact.artifact_id}`);

    // 3. Delete associated files
    await this.deleteArtifactFiles(artifact);

    // 4. Log to audit trail
    await this.logDeletion(artifact);
}

```

```

    async archiveArtifact(artifact: ArtifactMetadata): Promise<void> {
        const content = await
redis.json.get(`artifact:${artifact.artifact_id}`);

        // Compress
        const compressed = await gzip(JSON.stringify(content));

        // Store in archive location
        await redis.set(
            `archive:artifact:${artifact.artifact_id}`,
            compressed
        );

        // Delete active copy
        await redis.del(`artifact:${artifact.artifact_id}`);

        // Update metadata
        await redis.json.set(
            `artifact_meta:${artifact.artifact_id}`,
            '$.status',
            'archived'
        );
        await redis.json.set(
            `artifact_meta:${artifact.artifact_id}`,
            '$.archived_at',
            new Date().toISOString()
        );
    }
}

```

User Controls

```

        // User can star important runs
        async starArtifact(artifactId: string, userId: string):
Promise<void> {
            await redis.json.set(
                `artifact_meta:${artifactId}`,
                '$.user_starred',
                true
            );

            await redis.json.set(
                `artifact_meta:${artifactId}`,
                '$.protected',
                true
            );

            await redis.json.set(
                `artifact_meta:${artifactId}`,
                '$.protection_reason',
                'user_starred'
            );
        }

        // User can request early deletion
        async requestEarlyDeletion(artifactId: string, userId: string):
Promise<void> {
            const artifact = await
redis.json.get(`artifact_meta:${artifactId}`);

            // Cannot delete protected artifacts
            if (artifact.protected) {
                throw new Error("Cannot delete protected artifact");
            }
        }
    }
}

```

```

// Schedule immediate deletion
await redis.json.set(
  `artifact_meta:${artifactId}`,
  '$.expires_at',
  new Date().toISOString()
);
}

```

Scheduled GC Execution

```

// Cron job runs daily at 2am
cron.schedule("0 2 * * **", async () => {
  const report = await artifactGC.scan();

  await activityStream.log({
    agent: "system",
    type: "garbage_collection",
    message: `GC completed: ${report.archived} archived,
    ${report.scheduled_deletion} deleted`,
    metadata: report
  });

  // Store report
  await redis.json.set(
    `gc_report:${new Date().toISOString()}`,
    '$',
    report
  );
});

```

7. Library Access Control Policy

Problem Statement

Libraries contain sensitive documents. Need: - Agent-level permissions (who can access what) - Cloud model restrictions (private data stays local) - Rate limiting (prevent exfiltration) - PII detection (auto-redaction)

Solution: Library-Level Access Policies

Policy Schema

```

type LibraryAccessPolicy = {
  library_id: string;
  library_name: string;

  // Sensitivity classification
  sensitivity: "public" | "private" | "restricted" | "confidential";

  // Agent access control
  allowed_agents: string[]; // role_ids from registry
  denied_agents: string[]; // Explicit denies (overrides
allowed)

  // Model restrictions
  cloud_models_allowed: boolean;
  allowed_models: string[]; // Specific model allowlist
  denied_models: string[]; // Model denylist

  // Rate limiting
  retrieval_limits: {

```

```

        max_chunks_per_query: number;    // e.g., 20
        max_queries_per_run: number;     // e.g., 10
        max_queries_per_user_daily: number; // e.g., 100
    };

    // Content protection
    redaction_rules: {
        pii_detection: boolean;
        custom_patterns: RedactionPattern[];
        redaction_method: "mask" | "remove" | "hash";
    };

    // Audit requirements
    audit: {
        log_all_access: boolean;
        require_justification: boolean;    // User must explain why
        retention_days: number;            // How long to keep access
    };

    // Approval gates
    require_approval_for: {
        cloud_escalation: boolean;
        export_to_artifact: boolean;
        cross_library_joins: boolean;
    };
};

```

Redaction Pattern Schema

```

type RedactionPattern = {
    pattern_id: string;
    name: string;
    regex: string;
    replacement: string;    // What to replace with
    severity: "low" | "medium" | "high";
};

const DefaultRedactionPatterns: RedactionPattern[] = [
    {
        pattern_id: "ssn",
        name: "Social Security Number",
        regex: "\\b\\d{3}-\\d{2}-\\d{4}\\b",
        replacement: "***_*_*_*_*_*",
        severity: "high"
    },
    {
        pattern_id: "credit_card",
        name: "Credit Card",
        regex: "\\b\\d{4}[\\s-]?\\d{4}[\\s-]?\\d{4}[\\s-]?\\d{4}\\b",
        replacement: "****_*_*_*_*_*_*_*_*_*_*",
        severity: "high"
    },
    {
        pattern_id: "email",
        name: "Email Address",
        regex: "\\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Z|a-z]{2,}\\b",
        replacement: "***@***.*",
        severity: "medium"
    },
    {
        pattern_id: "phone",
        name: "Phone Number",
        regex: "\\b\\(\\d{3}\\)\\d{3}-\\d{4}\\b",
        replacement: "***_*_*_*_*_*",
    }
];

```

```

        severity: "medium"
    },
    {
        pattern_id: "account_number",
        name: "Bank Account",
        regex: "\\b[Aa]ccount[:\\s#]*\\d{8,12}\\b",
        replacement: "Account: *****",
        severity: "high"
    }
];

```

Example Policies

```

const ExamplePolicies: LibraryAccessPolicy[] = [
  // Public library - unrestricted
  {
    library_id: "public_research",
    library_name: "Public Research Papers",
    sensitivity: "public",
    allowed_agents: ["*"], // All agents
    denied_agents: [],
    cloud_models_allowed: true,
    allowed_models: ["*"],
    denied_models: [],
    retrieval_limits: {
      max_chunks_per_query: 50,
      max_queries_per_run: 50,
      max_queries_per_user_daily: 1000
    },
    redaction_rules: {
      pii_detection: false,
      custom_patterns: [],
      redaction_method: "mask"
    },
    audit: {
      log_all_access: false,
      require_justification: false,
      retention_days: 30
    },
    require_approval_for: {
      cloud_escalation: false,
      export_to_artifact: false,
      cross_library_joins: false
    }
  },

  // Financial library - highly restricted
  {
    library_id: "financials",
    library_name: "Financial Documents",
    sensitivity: "confidential",
    allowed_agents: ["research", "analyst"],
    denied_agents: ["vision", "update"], // No vision/update on
financials
    cloud_models_allowed: false, // Must use local models only
    allowed_models: ["qwen2.5:14b", "qwen2.5:32b"],
    denied_models: ["*-cloud"],
    retrieval_limits: {
      max_chunks_per_query: 10,
      max_queries_per_run: 5,
      max_queries_per_user_daily: 50
    },
    redaction_rules: {
      pii_detection: true,
      custom_patterns: [
        DefaultRedactionPatterns.find(p => p.pattern_id === "ssn"),

```

```

        DefaultRedactionPatterns.find(p => p.pattern_id ===
"account_number"),
        {
            pattern_id: "salary",
            name: "Salary Information",
            regex: "\\$[\\d,]+\\.\\.\\d{2}",
            replacement: "$***,***.***",
            severity: "high"
        }
    ],
    redaction_method: "mask"
},
audit: {
    log_all_access: true,
    require_justification: true,
    retention_days: 2555 // 7 years (compliance)
},
require_approval_for: {
    cloud_escalation: true,
    export_to_artifact: true,
    cross_library_joins: true
}
},

// Personal library - private
{
    library_id: "personal",
    library_name: "Personal Notes",
    sensitivity: "private",
    allowed_agents: ["research", "analyst", "coder"],
    denied_agents: [],
    cloud_models_allowed: true,
    allowed_models: ["*"],
    denied_models: [],
    retrieval_limits: {
        max_chunks_per_query: 30,
        max_queries_per_run: 20,
        max_queries_per_user_daily: 500
    },
    redaction_rules: {
        pii_detection: true,
        custom_patterns: [
            DefaultRedactionPatterns.find(p => p.pattern_id ===
"email"),
            DefaultRedactionPatterns.find(p => p.pattern_id === "phone")
        ],
        redaction_method: "mask"
    },
    audit: {
        log_all_access: true,
        require_justification: false,
        retention_days: 90
    },
    require_approval_for: {
        cloud_escalation: false,
        export_to_artifact: false,
        cross_library_joins: false
    }
}
}
];

```

Access Control Enforcement

```

class LibraryAccessController {
    async checkAccess(
        libraryId: string,

```

```

    agentRole: string,
    modelId: string,
    runId: string
  ): Promise<AccessDecision> {
    const policy = await this.getPolicy(libraryId);

    // 1. Check agent access
    if (!this.isAgentAllowed(policy, agentRole)) {
      return {
        allowed: false,
        reason: "policy_violation",
        details: `Agent '${agentRole}' not allowed to access library
'${libraryId}'`;
      };
    }

    // 2. Check model restrictions
    if (!this.isModelAllowed(policy, modelId)) {
      return {
        allowed: false,
        reason: "policy_violation",
        details: `Model '${modelId}' not allowed on library
'${libraryId}'`;
      };
    }

    // 3. Check rate limits
    const rateLimitOk = await this.checkRateLimits(policy, runId);
    if (!rateLimitOk) {
      return {
        allowed: false,
        reason: "rate_limit_exceeded",
        details: `Exceeded retrieval limits for library
'${libraryId}'`;
      };
    }

    // 4. Request justification if required
    if (policy.audit.require_justification) {
      const justification = await this.requestJustification(runId,
libraryId);
      if (!justification) {
        return {
          allowed: false,
          reason: "no_justification",
          details: "Library access requires justification"
        };
      }
    }

    return {
      allowed: true,
      policy_applied: policy.library_id
    };
  }

  private isAgentAllowed(policy: LibraryAccessPolicy, agentRole:
string): boolean {
    // Explicit deny overrides allow
    if (policy.denied_agents.includes(agentRole)) {
      return false;
    }

    // Check allowlist
    if (policy.allowed_agents.includes("*") ||
policy.allowed_agents.includes(agentRole)) {

```

```

        return true;
    }

    return false;
}

private isModelAllowed(policy: LibraryAccessPolicy, modelId:
string): boolean {
    // Check cloud restriction
    if (!policy.cloud_models_allowed && modelId.includes("-cloud"))
{
        return false;
    }

    // Explicit deny overrides allow
    for (const denied of policy.denied_models) {
        if (denied === "*" && modelId.includes("-cloud")) return
false;

        if (modelId.includes(denied)) return false;
    }

    // Check allowlist
    if (policy.allowed_models.includes("*")) return true;
    return policy.allowed_models.some(allowed =>
modelId.includes(allowed));
    }
}

```

Redaction Engine

```

class RedactionEngine {
    async applyRedactions(
        text: string,
        policy: LibraryAccessPolicy
    ): Promise<RedactedText> {
        if (!policy.redaction_rules.pii_detection) {
            return { text, redactions: [] };
        }

        let redactedText = text;
        const redactions: Redaction[] = [];

        for (const pattern of policy.redaction_rules.custom_patterns) {
            const regex = new RegExp(pattern.regex, 'g');
            const matches = [...text.matchAll(regex)];

            for (const match of matches) {
                redactions.push({
                    pattern_id: pattern.pattern_id,
                    original_position: match.index,
                    original_length: match[0].length,
                    severity: pattern.severity
                });
            }

            redactedText = redactedText.replace(
                regex,
                pattern.replacement
            );
        }

        return {
            text: redactedText,
            redactions,
            redaction_count: redactions.length
        };
    }
}

```

```

    }
  }
}

```

Access Audit Trail

```

type AccessLog = {
  log_id: string;
  timestamp: string;

  // Who
  run_id: string;
  user_id: string;
  agent_role: string;

  // What
  library_id: string;
  document_ids: string[];
  chunk_ids: string[];

  // How
  query: string;
  model_id: string;
  retrieval_count: number;

  // Why
  justification?: string;

  // Outcome
  redactions_applied: number;
  access_granted: boolean;
  policy_violations: string[];
};

class AccessAuditor {
  async logAccess(log: AccessLog): Promise<void> {
    // Store in Redis
    await redis.json.set(`access_log:${log.log_id}`, '$', log);

    // Add to timeline index
    await redis.zadd(
      `access_logs:${log.library_id}`,
      Date.now(),
      log.log_id
    );

    // Track violations
    if (log.policy_violations.length > 0) {
      await redis.lpush(
        'security:violations',
        JSON.stringify(log)
      );

      // Alert on high-severity violations
      await this.alertOnViolation(log);
    }
  }

  async getAccessReport(
    libraryId: string,
    startDate: string,
    endDate: string
  ): Promise<AccessReport> {
    const logs = await redis.ft.search(
      'idx:access_logs',
      `@library_id:${libraryId} @timestamp:[${startDate}
    ${endDate}]`

```

```
);

return {
  library_id: libraryId,
  total_accesses: logs.total,
  unique_users: new Set(logs.documents.map(d =>
d.user_id)).size,
  agents_used: this.countByAgent(logs.documents),
  violations: logs.documents.filter(d =>
d.policy_violations.length > 0).length,
  redactions_applied: logs.documents.reduce((sum, d) => sum +
d.redactions_applied, 0)
};
}
```

Implementation Roadmap

Phase 1: Core Robustness (Week 1-2)

1. Explicit Termination Semantics
2. Resource Budget Guards
3. Context Budget Ownership

Why first: These prevent catastrophic failures and runaway costs.

Phase 2: Operational Safety (Week 3-4)

4. Deterministic Tool Registry
5. Conflict Resolution Policy
6. Artifact Lifecycle & GC

Why second: Makes the system production-ready and maintainable.

Phase 3: Security & Compliance (Week 5-6)

7. Library Access Control Policy

Why last: Builds on stable foundation; compliance layer on top.

Validation Checklist

Before deploying v4:

- ☐ Every run terminates with explicit TerminationReason
 - ☐ Conflict resolution policy tested with real agent disagreements
 - ☐ Context never silently truncated (ContextSummary artifacts present)
 - ☐ Tool registry routes deterministic tasks before LLM calls
 - ☐ Resource budgets enforced on all runs
 - ☐ Garbage collector running daily, no Redis bloat
 - ☐ Library access policies prevent unauthorized access
 - ☐ Redaction engine tested on PII-containing docs
 - ☐ All 7 features have integration tests
 - ☐ Activity stream logs all policy events
 - ☐ Termination reasons analytics dashboard created
-

Appendix A: Redis Schema Summary

```
# Termination records
termination:{run_id}

# Budget tracking
budget:run:{run_id}
budget:user:{user_id}:daily:{date}
budget:global

# Tool registry
registry:tools
tool_stats:{tool_id}

# Conflict resolutions
conflict:{conflict_id}

# Context summaries
context_summary:{summary_id}

# Artifact metadata
artifact_meta:{artifact_id}
archive:artifact:{artifact_id}

# Library policies
policy:library:{library_id}

# Access logs
access_log:{log_id}
access_logs:{library_id} # Sorted set for timeline

# GC reports
gc_report:{timestamp}
```

Appendix B: Integration with Existing v3 Features

v4 Feature	Integrates With (v3)	How
Termination Semantics	Phase State Machine	Replaces implicit exits with explicit reasons
Conflict Resolution	Quorum System	Defines arbitration when quorum fails
Context Budget	Memory Tiers	Mother uses tiers to compose context within budget
Tool Registry	Capability Routing	Tools checked before model selection
Resource Budgets	Cloud Escalation	Budget gates cloud model

Artifact GC	Artifact Storage	usage
		Adds lifecycle to existing artifacts
Library Access	RAG Contract	Adds policy layer to retrieval

Document Changelog

Version	Date	Changes
v4.0	2025-12-19	Initial specification: 7 robustness features

End of Addendum