Conceptual Agent Architecture

Career Strategy Intelligence System

Version 1.0

Technical Architecture Documentation

February 13, 2026

Contents

1 Introduction

1.1 Architecture Overview

This document specifies the conceptual agent architecture for a career strategy intelligence system designed to operate under severe computational constraints while providing sophisticated strategic reasoning capabilities. The architecture implements a multi-agent system where specialized cognitive roles collaborate through structured communication to deliver strategic career guidance.

The system addresses the fundamental challenge of providing high-quality strategic intelligence within the constraints of free-tier API access (approximately 3,000,000 tokens per month) and consumer-grade hardware (8-16GB RAM, CPU-only processing). This constraint-driven design necessitates architectural decisions that prioritize computational efficiency as a first-class concern, not an afterthought.

1.2 Design Philosophy

The architecture embodies four core principles:

Cognitive Specialization: Each agent implements one distinct cognitive role rather than functioning as a general-purpose assistant. This specialization enables optimization of reasoning strategies, memory structures, and computational costs specific to each role's domain.

Compressed Intelligence: Agents produce compressed insights rather than exhaustive analyses. Output density is maximized while verbosity is eliminated. Inter-agent communication uses structured data objects measured in tens of tokens, not natural language measured in hundreds of tokens.

Lazy Evaluation: Computation occurs only when necessary. Staleness checks precede analysis. Cached results are preferred over fresh computation. The system is designed to avoid work, not to perform work.

Computational Consciousness: Every agent is designed with explicit awareness of token costs. High-cost operations require justification. The default action is algorithmic processing consuming zero tokens, with language model invocation reserved for genuinely ambiguous cases requiring semantic understanding.

1.3 Constraint-Driven Architecture

Unlike traditional software systems where performance optimization occurs after functional design, this architecture begins with computational constraints and derives functional capabilities within those bounds. The constraint of 3,000,000 monthly tokens is not a performance target but a hard physical limit analogous to memory constraints in embedded systems.

This inverted design process produces an architecture fundamentally different from unconstrained systems:

- Pre-computation replaces real-time analysis wherever temporal staleness is acceptable
- Caching is architectural, not optimization
- Algorithmic processing is preferred over learned models

- Structured data replaces natural language in system internals
- Tiered processing allocates expensive resources to high-value decisions

1.4 Document Structure

Section 2 defines the agent topology and communication infrastructure. Section 3 specifies each agent's cognitive role, memory architecture, decision logic, and conflict resolution powers. Section 4 describes system-level coordination including message routing, rate limit management, and failure handling. Section 5 analyzes how computational constraints shape architectural decisions. Section 6 presents detailed interaction examples demonstrating agent orchestration.

1.5 Terminology

Agent: An autonomous reasoning component with specialized cognitive role, isolated memory, and defined communication protocols.

Tier 0/1/2/3: Computational cost classification where Tier 0 consumes zero tokens (algorithmic), Tier 1 consumes 50-100 tokens (cached templates), Tier 2 consumes 200-400 tokens (novel analysis), and Tier 3 consumes 500-1000 tokens (complex synthesis).

Staleness: Temporal distance between cached data and current reality, measured against domain-specific thresholds.

Message Bus: Central coordination infrastructure routing structured messages between agents.

Episodic Memory: Agent memory of specific events with temporal context.

Semantic Memory: Agent memory of general patterns and rules without temporal context.

Working Memory: Agent memory of current active state.

2 Architecture Topology

2.1 Agent Roster

The system comprises seven specialized agents plus central coordination infrastructure:

1. **State Estimator Agent:** Maintains authoritative representation of candidate's market position
2. **Opportunity Filter Agent:** Performs rapid triage of opportunities into ignore/consider/prioritize categories
3. **Information Gap Analyzer:** Identifies critical unknowns blocking decisions and prioritizes information-gathering
4. **Temporal Coordination Agent:** Manages deadlines, detects resource conflicts, optimizes action sequencing
5. **Strategy Synthesizer Agent:** Integrates inputs from other agents into coherent action recommendations
6. **Reality Feedback Agent:** Compares predictions to outcomes and updates system priors
7. **Central Message Bus:** Coordinates communication, enforces rate limits, manages batching

2.2 Communication Topology

2.2.1 Structural Constraints

The architecture enforces strict communication topology to prevent computational runaway:

No Direct Agent-to-Agent Invocation: Agents cannot invoke each other directly. All communication flows through the central message bus. This prevents recursion bombs where Agent A calls Agent B which calls Agent C which calls Agent A.

Maximum Interaction Depth: Agent interactions are limited to two hops: Agent → Message Bus → Agent. No transitive chains (Agent → Agent → Agent) are permitted. This bounds worst-case computational cost.

Structured Communication Only: Agents communicate using predefined structured schemas, not natural language. A state update message contains explicit fields (credentials changed, timeline shifted, preferences updated) rather than prose description. This enables deterministic parsing, validation, and zero-token message handling.

2.2.2 Message Flow Patterns

State Propagation Pattern:

```
User Input → State Estimator (staleness check)
             ↓
             IF stale: State Estimator updates state
             ↓
             State Estimator → Message Bus → [Opportunity Filter,
                                              Information Gap,
                                              Strategy Synthesizer]
```

Synthesis Pattern:

```
Strategy Synthesizer ← Message Bus ← [State Estimator (current state),
                                      Opportunity Filter (ranked queue),
                                      Information Gap (uncertainties),
                                      Temporal Coordinator (timeline)]
            ↓
Strategy Synthesizer (integration)
            ↓
Strategy Synthesizer → Message Bus → User Response
```

Feedback Pattern:

```
User Outcome Report → Reality Feedback
                      ↓
Reality Feedback (compare prediction vs outcome)
                      ↓
IF error > threshold: Reality Feedback → Message Bus →
                                         Responsible Agent (recalibration)
```

2.2.3 Message Schema

All inter-agent messages conform to the following structure:

```
Message {
    type: string,               # Message classification
    sender: agent_id,           # Originating agent
    recipient: agent_id,        # Target agent
    payload: {                  # Structured data (no prose)
        field_1: value_1,
        field_2: value_2,
        ...
    },
    priority: {urgent, normal, low},
    timestamp: datetime,
    requires_llm: boolean,      # Does processing need LLM?
    estimated_tokens: int       # Cost estimate for budgeting
}
```

Listing 1: Message Schema Structure

2.2.4 Routing Rules

The message bus implements deterministic routing based on message type:

2.3 Architectural Invariants

The following properties are maintained as architectural invariants:

Invariant 1 (No Recursion): Agent A cannot trigger Agent B which triggers Agent A within a single user request. The message bus detects and rejects circular invocation patterns.

Message Type	Recipients
STATE_DELTA	Opportunity Filter, Information Gap, Strategy Synthesizer
FILTER_RESULTS	Strategy Synthesizer
UNCERTAINTY_MAP	Strategy Synthesizer, Temporal Coordinator
TIMELINE_CONFLICT	Strategy Synthesizer
STRATEGY_DECISION	Temporal Coordinator, Information Gap
OUTCOME_OBSERVED	Reality Feedback
RECALIBRATION_REQUIRED	Specific Agent (identified in payload)

Table 1: Message Routing Table

Invariant 2 (Bounded Depth): Any user request triggers at most two levels of agent activation. User → Agent → Agent is permitted. User → Agent → Agent → Agent is forbidden.

Invariant 3 (Structured Data): All inter-agent communication uses structured data formats (dictionaries, arrays, primitives). Natural language generation occurs only at the user-facing boundary.

Invariant 4 (Staleness Veto): State Estimator's staleness veto cannot be overridden except by explicit user force-refresh request. This prevents hallucinated analyses based on outdated state.

Invariant 5 (Feasibility Constraint): Temporal Coordinator's feasibility constraints cannot be violated. Strategy Synthesizer cannot propose physically impossible timelines.

2.4 Agent Autonomy and Coordination

Agents exhibit autonomy within their cognitive domain while respecting coordination constraints:

Domain Autonomy: Within its domain, each agent makes independent decisions. Opportunity Filter scores opportunities using its own criteria. Information Gap determines which uncertainties matter. No agent overrides another agent's domain expertise.

Coordination Points: Agents must coordinate at defined interfaces:

- State Estimator dictates when state has changed enough to warrant re-analysis
- Temporal Coordinator dictates what is physically feasible given time constraints
- Strategy Synthesizer resolves conflicts when agents propose incompatible actions
- Reality Feedback adjusts all agents' credibility weights based on prediction accuracy

Veto Powers: Specific agents hold veto authority in their domain:

- State Estimator: Can veto computation requests if state unchanged
- Temporal Coordinator: Can veto infeasible action proposals
- Information Gap: Can delay decisions by flagging critical unknowns (overrideable by Strategy Synthesizer with explicit risk acceptance)

3 Agent Specifications

3.1 State Estimator Agent

3.1.1 Cognitive Role

The State Estimator implements a change detection sensor that maintains authoritative representation of the candidate's current market position. Its primary cognitive function is determining when state changes are material enough to warrant downstream re-computation.

The agent operates as a differential state model: only deltas (changes) propagate through the system. This architectural decision reduces computational load by preventing redundant analysis of unchanged state dimensions.

Intelligence Type: Pattern recognition and threshold detection. Minimal reasoning, maximum efficiency through algorithmic comparison.

3.1.2 Memory Architecture

Working Memory

```
CandidateState {
    credentials: {
        degrees: [Degree],
        publications: [Publication],
        skills: [Skill],
        certifications: [Certification]
    },
    constraints: {
        geographic: [Location],
        financial: {salary_floor: float, ...},
        timeline: {min_start: date, max_start: date}
    },
    preferences: {
        field: [string],
        role_type: [string],
        priorities: {dimension: weight}
    },
    search_history: {
        applications: [Application],
        outcomes: [Outcome],
        timestamps: [datetime]
    },
    metadata: {
        last_updated: datetime,
        version_hash: string
    }
}
```

Listing 2: Working Memory Structure

Storage: Single active state object, approximately 5-10KB serialized.

Episodic Memory

```
StateChangeLog {
    timestamp: datetime,
    changed_dimensions: [string],
    delta_magnitude: {dimension: float},
```

```
    triggered_downstream: [agent_id],
    reason: string
}
```

Listing 3: Episodic Memory Structure

Retention Policy: 90-day rolling window. Weekly summaries after 30 days. Compression ratio approximately 10:1.

Purpose: Audit trail for understanding state evolution. Debugging unexpected agent behavior. Learning common change patterns.

Semantic Memory

```
BenchmarkDistributions {
    field: string,
    career_stage: string,
    credential_percentiles: {
        dimension: {
            mean: float,
            std: float,
            percentiles: [float]
        }
    },
    cached_until: datetime
}
```

Listing 4: Semantic Memory Structure

Update Frequency: Weekly batch update from aggregated user data and public benchmarks.

Size: Approximately 50-100KB for all tracked fields and career stages.

3.1.3 Decision Logic

Algorithm 1 State Staleness Detection

```
function IsStale(state, last_computation)
    time_delta ← now() − last_computation.timestamp
    if time_delta > MAX_CACHE_AGE then
        return TRUE
    end if
    for dimension ∈ state.dimensions do
        delta ← |state[dimension] − last_computation.state[dimension]|
        threshold ← STALENESS_THRESHOLDS[dimension]
        if delta > threshold then
            return TRUE
        end if
    end for
    return FALSE
end function
```

Staleness Detection Algorithm Staleness Thresholds:

Maximum Cache Age: 14 days absolute staleness limit regardless of dimension changes.

Dimension	Threshold
Publications	1 new publication
Applications	5 new applications
Credentials	1 new credential
Timeline (days)	30 day shift
Preferences (weight change)	0.2 (20% change)

Table 2: Material Change Thresholds

Algorithm 2 State Delta Propagation

function PROPAGATECHANGES($state\_delta$)
  $affected \leftarrow DetermineAffectedAgents(state\_delta)$
  for $agent \in affected$ **do**
    $message \leftarrow \{$
      $type$: "$STATE\_DELTA$",
      $changed\_dimensions$: $state\_delta.dimensions$,
      new_values : $\{dim : state[dim] \mid dim \in state\_delta.dimensions\}$,
      $trigger\_reason$: $state\_delta.reason$
    $\}$
    $MessageBus.send(agent, message)$
  end for
end function

Delta Propagation Output Size: 20-50 tokens per state delta message. Only changed dimensions included, not entire state.

3.1.4 Conflict Resolution Powers

Veto Authority The State Estimator holds absolute veto power over computational requests when state has not materially changed. This veto cannot be overridden by any agent except through explicit user force-refresh command.

Enforcement Mechanism:

Algorithm 3 Computation Request Handling

function HANDLEREQUEST($agent, request$)
  if $IsStale(current\_state, agent.last\_computation)$ **then**
    return `ALLOW` request
  else
    return `VETO` request
    return $cached\_result(agent.last\_computation)$
  end if
end function

Rationale: Prevents wasteful re-computation when inputs have not changed. Enforces computational discipline across the system. State Estimator is the only agent with complete visibility into state changes, making it uniquely positioned to make staleness determinations.

Exception Handling User force-refresh requests bypass staleness veto:

```
if user_request.force_refresh == True:
```

```
    OVERRIDE staleness_veto
    MARK all_cached_analyses as stale
    TRIGGER fresh_computation
```

3.1.5 Token Economics

Computational Cost Breakdown:

Operation	Frequency	Token Cost
Staleness check	Per request (100%)	0
State update (structured)	Per material change (5%)	0
Ambiguous change interpretation	Per ambiguous case (1%)	50
Expected per request		**0.5**

Table 3: State Estimator Token Costs

Optimization Strategy:

99% of operations consume zero tokens through algorithmic comparison. Language model invocation occurs only for ambiguous state changes requiring semantic interpretation.

Example Ambiguous Case:

User reports: "I attended a 2-day Python workshop"

Algorithmic check: Workshop not in credential taxonomy → ambiguous

LLM invocation (50 tokens):

> "Classify: Does '2-day Python workshop' constitute a credential that would materially change competitive positioning for PhD applicants in computational biology? YES/NO with brief justification."

LLM response: "NO. Short workshop insufficient to change positioning. Requires semester-level coursework or demonstrated project work to be material."

Decision: Do not update state, cache classification for future.

3.2 Opportunity Filter Agent

3.2.1 Cognitive Role

The Opportunity Filter implements a low-pass filter that rapidly triages opportunities to prevent wasted analytical effort on mismatched positions. Its cognitive function is binary classification (reject/proceed) followed by probabilistic ranking of opportunities that pass initial screening.

The agent prioritizes throughput over precision: false negatives (rejecting good opportunities) are acceptable; false positives (advancing bad opportunities to expensive analysis) are costly. This asymmetry shapes the agent's decision thresholds.

Intelligence Type: Heuristic pattern matching with learned weights. Minimal reasoning required for most decisions. Language model invoked only for ambiguous cases near decision boundary.

3.2.2 Memory Architecture

Working Memory

```
OpportunityQueue {
    max_size: 25,
    opportunities: [
        {
            id: string,
            source: string,
            details: {structured_fields},
            score: float,              # Match quality [0,1]
            confidence: float,         # Confidence in score [0,1]
            timestamp: datetime
        }
    ],
    sorted_by: score descending
}
```

Listing 5: Working Memory Structure

Rationale for size limit: Top 25 opportunities sufficient for user decision-making. Larger queues would waste memory and introduce choice paralysis. Agent maintains only the priority queue, discarding lower-ranked opportunities.

Episodic Memory None. The Opportunity Filter is stateless, relying entirely on State Estimator for candidate context. This architectural decision simplifies the agent and prevents state synchronization issues.

Semantic Memory

```
FilterRules {
    hard_constraints: {
        geographic: [allowed_locations],
        timeline: {min_start_date, max_start_date},
        visa_status: requirements,
        salary_floor: minimum_acceptable
    },

    scoring_weights: {
        field_match: float,
        credential_fit: float,
        career_stage: float,
        prestige_tier: float,
        funding_security: float
    },

    learned_from_outcomes: boolean,
    last_calibration: datetime
}
```

Listing 6: Semantic Memory Structure

Storage: Less than 1KB. Designed for fast lookup.

Update Frequency: Monthly after Reality Feedback performs calibration analysis comparing predicted scores to actual outcomes.

3.2.3 Decision Logic

Algorithm 4 Hard Constraint Check (Tier 0)

```
function PassesHardConstraints(opportunity, user_state)
    if opportunity.location ∉ user_state.allowed_locations then
        return REJECT
    end if
    if opportunity.start_date < user_state.min_start_date then
        return REJECT
    end if
    if opportunity.visa ∉ user_state.visa_status then
        return REJECT
    end if
    if opportunity.salary < user_state.salary_floor then
        return REJECT
    end if
    return PASS
end function
```

Hard Constraint Filtering Implementation: Regular expressions, keyword matching, numeric comparisons. No language model required.

Token Cost: 0 for all evaluations.

Rejection Rate: Approximately 60% of opportunities rejected at this stage.

Algorithm 5 Opportunity Match Scoring

```
function ComputeMatchScore(opportunity, user_state)
    features ← ExtractFeatures(opportunity)
    score ← 0
    for (feature, value) ∈ features do
        weight ← FilterRules.scoring_weights[feature]
        score ← score + weight × value
    end for
    confidence ← EstimateConfidence(features)
    if |score − THRESHOLD| < AMBIGUITY_MARGIN then
        llm_score ← LLM_Classify(opportunity, user_state)
        score ← 0.7 × score + 0.3 × llm_score
        confidence ← confidence × 0.8
    end if
    return (score, confidence)
end function
```

Match Scoring Parameters:

- THRESHOLD: 0.6 (minimum score to proceed to detailed analysis)
- AMBIGUITY_MARGIN: 0.1 (range triggering LLM disambiguation)

Token Cost Analysis:

- Algorithmic scoring: 0 tokens (95% of cases)
- LLM disambiguation: 50 tokens (5% of cases near threshold)

• Expected cost per opportunity: $0.05 \times 50 = 2.5$ tokens

For 1,000 opportunities per month: 2,500 tokens total for filtering.

Algorithm 6 Opportunity Queue Prioritization

```
function PrioritizeQueue(opportunities)
    scored ← [ComputeMatchScore(opp) | opp ∈ opportunities]
    sorted ← Sort(scored, key = λx : x.score × x.confidence, reverse = True)
    return sorted[: 25]
end function
```

Prioritization Ranking Metric: priority = score × confidence

This risk-adjusted ranking down-weights opportunities where confidence is low, preventing overconfident recommendations based on uncertain information.

3.2.4 Conflict Resolution Powers

Proposal Authority The Opportunity Filter proposes prioritization order to Strategy Synthesizer. Strategy Synthesizer may override but must provide justification.

Algorithm 7 Override Rate Monitoring

```
function CheckOverrideRate
    overrides ← Count(StrategySynthesizer.overrides, last_30_days)
    total ← Count(self.recommendations, last_30_days)
    rate ← overrides/total
    if rate > 0.30 then
        TRIGGER recalibration_request
        MessageBus.send(RealityFeedback, {
          type : "FILTER_MISCALIBRATION",
          override_rate : rate,
          examples : recent_overrides
        })
    end if
end function
```

Recalibration Trigger Threshold Rationale: 30% override rate indicates systematic miscalibration. Lower rates expected from normal disagreement on borderline cases. Higher rates suggest filter's criteria diverge from optimal strategy.

Bypass Prevention Every opportunity must pass through Opportunity Filter. No exceptions. This invariant is enforced by the message bus routing rules.

Attempting to bypass the filter generates a routing error and request rejection.

3.2.5 Token Economics

Expected Cost per Opportunity:

Monthly Capacity:

Stage	Probability	Tokens	Expected
Hard constraints (fail)	60%	0	0.0
Algorithmic scoring	38%	0	0.0
LLM disambiguation	2%	50	1.0
Total per opportunity			**1.0**

Table 4: Opportunity Filter Cost Breakdown

For monthly budget allocation of 5,000 tokens to opportunity filtering:

- Capacity: 5,000 opportunities evaluated
- With 100 active users: 50 opportunities per user per month
- More than sufficient for typical search volumes

3.3 Information Gap Analyzer

3.3.1 Cognitive Role

The Information Gap Analyzer implements an uncertainty cartographer that maps decision trees, annotates unknowns, and determines which uncertainties are resolvable versus irreducible. Its cognitive function is information-theoretic: calculating expected value of information (EVOI) to prioritize which uncertainties to resolve before deciding.

The agent distinguishes three classes of uncertainty:

1. **Resolvable:** Can be resolved through information-gathering at acceptable cost
2. **Irreducible:** Cannot be resolved regardless of effort (inherent randomness)
3. **Costly:** Resolvable but cost exceeds value of resolution

Intelligence Type: Decision-theoretic reasoning with information value calculation.

3.3.2 Memory Architecture

Working Memory

```
UncertaintyGraph {
    decision_nodes: [
        {
            decision_id: string,
            type: {accept, reject, delay, negotiate},
            prerequisites: [uncertainty_id],
            sensitivity: float  # Impact magnitude
        }
    ],

    uncertainties: [
        {
            uncertainty_id: string,
            description: string,
            type: {resolvable, irreducible, costly},
            resolution_cost: {
```

```
                time: hours,
                money: float,
                social_capital: float
            },
            impact_on_decisions: [decision_id],
            evoi: float
        }
    ]
}
```

Listing 7: Working Memory Structure

Episodic Memory

```
InformationGatheringHistory {
    action: string,
    cost_actual: {time, money, social_capital},
    information_gained: string,
    decision_impact: {
        changed_decision: boolean,
        value_improvement: float
    },
    value_realized: float
}
```

Listing 8: Episodic Memory Structure

Retention: All history maintained for learning EVOI estimation accuracy.

Purpose: Calibrate predictions of information-gathering value. Learn which question types yield high versus low information gain.

Semantic Memory

```
UncertaintyTemplates {
    decision_type: string,
    common_uncertainties: [
        {
            uncertainty_type: string,
            typical_resolution_methods: [string],
            average_cost: {time, money, social_capital},
            average_value: float,
            success_rate: float
        }
    ]
}
```

Listing 9: Semantic Memory Structure

Purpose: Fast uncertainty enumeration for common decision types. Enables zero-token operation for routine decisions matching templates.

Update Frequency: Quarterly distillation from episodic memory.

3.3.3 Decision Logic

Uncertainty Enumeration Customization Cost: 50 tokens if context-specific adaptation required, 0 tokens if template applies directly.

Algorithm 8 Uncertainty Enumeration

```
function EnumerateUnknowns(decision)
    if decision.type ∈ UncertaintyTemplates then
        base ← UncertaintyTemplates[decision.type]
        token_cost ← 0
    else
        base ← LLM_EnumerateUnknowns(decision)
        token_cost ← 200
        UncertaintyTemplates.add(decision.type, base)
    end if
    specific ← CustomizeToContext(base, decision)
    return specific
end function
```

Algorithm 9 EVOI Calculation

```
function ComputeEVOI(uncertainty)
    value_resolved ← EstimateDecisionQuality(
      current_decision,
      assume_resolved = uncertainty)
    value_unresolved ← EstimateDecisionQuality(
      current_decision,
      assume_unresolved = uncertainty)
    value_gain ← value_resolved − value_unresolved
    cost ← EstimateResolutionCost(uncertainty)
    evoi ← value_gain − cost
    return evoi
end function
```

Expected Value of Information Implementation: Decision quality estimation uses simplified utility model (not full optimization). Resolution cost estimated from historical data in episodic memory.

Token Cost: 0 (algorithmic calculation using cached parameters).

Algorithm 10 Information Gathering Task Generation

```
function PrioritizeInformationGathering(uncertainties)
    ranked ← Sort(uncertainties, key = λu : u.evoi, reverse = True)
    high_value ← Filter(ranked, λu : u.evoi > THRESHOLD)
    for uncertainty ∈ high_value do
        EmitTask({
          type : "INFORMATION_GATHERING",
          uncertainty : uncertainty,
          method : uncertainty.best_resolution_method,
          expected_cost : uncertainty.resolution_cost,
          expected_value : uncertainty.evoi
        })
    end for
    irreducible ← Filter(uncertainties, λu : u.type == "irreducible")
    for uncertainty ∈ irreducible do
        EmitDecisionRuleUnderUncertainty(uncertainty)
    end for
end function
```

Information Gathering Prioritization Output: Prioritized action recommendations for user. High-EVOI uncertainties generate "ask this question to that person" tasks. Irreducible uncertainties generate "decide using this criterion despite uncertainty" guidance.

3.3.4 Conflict Resolution Powers

Delay Authority Information Gap Analyzer can delay decisions by flagging critical information gaps. Strategy Synthesizer can override but must explicitly accept risk.

Algorithm 11 Override Handling

```
function HandleOverride(decision_to_proceed)
    Log({
      decision : decision_to_proceed,
      unresolved : current_gaps,
      risk_accepted_by : "Strategy_Synthesizer",
      timestamp : now()
    })
    return {
      decision : decision_to_proceed,
      acknowledged_risks : current_gaps,
      confidence_penalty : ComputePenalty(current_gaps)
    }
end function
```

Confidence Penalty: Reduction in decision confidence proportional to unresolved uncertainty magnitude. Typical penalty: 0.1 to 0.3 reduction in confidence score.

Bypass Prevention Information gaps must be acknowledged even if accepted. No decisions can proceed without Information Gap analysis unless explicitly marked as "low-stakes routine decision" by user.

3.3.5 Token Economics

Cost per Decision Analysis:

Operation	Probability	Tokens	Expected
Template match	80%	0	0
Context customization	15%	50	7.5
Novel enumeration	5%	200	10.0
EVOI calculation	100%	0	0
Average per decision			**17.5**

Table 5: Information Gap Analyzer Costs

Optimization Strategy:

Build comprehensive template library over time. Initial decisions expensive (200 tokens for novel enumeration). Subsequent similar decisions cheap (0 tokens using cached template). System learns from experience, amortizing initial investment.

3.4 Temporal Coordination Agent

3.4.1 Cognitive Role

The Temporal Coordination Agent implements a constraint satisfaction scheduler that ensures proposed plans are physically executable within available time. Its cognitive function is feasibility checking: detecting resource conflicts, identifying forced choices, and constructing valid timelines.

The agent enforces hard constraints. Unlike other agents that provide recommendations (which can be overridden), Temporal Coordinator's feasibility determinations are non-negotiable: physical time cannot be violated.

Intelligence Type: Algorithmic optimization using graph algorithms and constraint satisfaction. Minimal language model usage.

3.4.2 Memory Architecture

Working Memory

```
Timeline {
    horizon: 6 months,

    deadlines: [
        {
            opportunity_id: string,
            deadline_type: {application, decision, start_date},
            date: datetime,
            flexibility: {hard, soft, negotiable}
        }
    ],
```

```
    tasks: [
        {
            task_id: string,
            description: string,
            estimated_duration: hours,
            dependencies: [task_id],
            deadline: datetime
        }
    ],

    available_time: [
        {
            start: datetime,
            end: datetime,
            hours_available: float
        }
    ]
}
```

Listing 10: Working Memory Structure

Episodic Memory

```
TaskDurationHistory {
    task_type: string,
    estimated_duration: hours,
    actual_duration: hours,
    error: float,
    timestamp: datetime
}
```

Listing 11: Episodic Memory Structure

Retention: 12-month rolling window for seasonal pattern detection.

Purpose: Calibrate duration estimates. User-specific learning: some users consistently underestimate preparation time, others overestimate. Agent adapts estimates to individual patterns.

Semantic Memory

```
TaskTemplates {
    task_type: string,
    typical_duration: hours,
    variability: float,
    dependencies: [string],
    source: {learned, general_knowledge}
}
```

Listing 12: Semantic Memory Structure

Examples:

- Faculty application: 12-20 hours (high variability)
- Reference letter request: 2 weeks lead time (dependency constraint)
- Interview preparation: 4-8 hours (medium variability)

Update Frequency: Monthly calibration from episodic memory.

3.4.3 Decision Logic

Algorithm 12 Timeline Feasibility Check (Tier 0)

```
function IsFeasible(tasks, deadlines, available_time)
    total_required ← Σ_{t∈tasks} t.duration
    total_available ← Σ_{b∈available_time} b.hours
    if total_required > total_available then
        return INFEASIBLE
    end if
    schedule ← GreedyScheduler(tasks, deadlines, available_time)
    if schedule = NULL then
        return INFEASIBLE
    end if
    return FEASIBLE
end function
```

Feasibility Check Token Cost: 0 (pure algorithmic scheduling).

GreedyScheduler: Earliest-deadline-first heuristic with dependency resolution. Not optimal but polynomial-time computable.

Algorithm 13 Resource Conflict Detection

```
function DetectConflicts(schedule)
    conflicts ← []
    for (task_A, task_B) ∈ Combinations(tasks, 2) do
        if Overlaps(task_A, task_B) then
            conflicts.append({
              type : "RESOURCE_CONFLICT",
              task_a : task_A,
              task_b : task_B,
              resolution : "Choose_one_or_extend_timeline"
            })
        end if
        if DeadlineImpossible(task_A) ∨ DeadlineImpossible(task_B) then
            impossible ← task_A if DeadlineImpossible(task_A) else task_B
            conflicts.append({
              type : "DEADLINE_CONFLICT",
              task : impossible,
              resolution : "Negotiate_deadline_or_abandon"
            })
        end if
    end for
    return conflicts
end function
```

Conflict Detection

Conflict Resolution Presentation

Algorithm 14 Forced Choice Generation

```
function ResolveConflicts(conflicts)
    for conflict ∈ conflicts do
        if conflict.type = "RESOURCE_CONFLICT" then
            EmitForcedChoice({
              option_a : conflict.task_a,
              option_b : conflict.task_b,
              message : "Insufficient_time_for_both.",
              decide_by : earliest(task_a.deadline, task_b.deadline)
            })
        end if
        if conflict.type = "DEADLINE_CONFLICT" then
            feasible ← ComputeFeasibleDeadline(conflict.task)
            EmitDeadlineNegotiation({
              task : conflict.task,
              current_deadline : conflict.task.deadline,
              feasible_deadline : feasible,
              message : "Cannot_meet_current_deadline."
            })
        end if
    end for
end function
```

3.4.4 Conflict Resolution Powers

Hard Priority on Feasibility Temporal Coordinator's feasibility constraints cannot be violated. Strategy Synthesizer can adjust priorities but cannot propose physically impossible timelines.

Algorithm 15 Infeasible Override Rejection

```
function HandleInfeasibleOverride(strategy_request)
    Log({
      requested_by : "Strategy_Synthesizer",
      requested : strategy_request,
      violation : "Exceeds_available_time_by_X_hours",
      timestamp : now()
    })
    return {
      status : "REJECTED",
      reason : "Physical_impossibility",
      forced_choices : EnumForcedChoices(strategy_request)
    }
end function
```

3.4.5 Token Economics

Typical Costs:

Ambiguous Cases Requiring LLM:

Duration estimation: "How long does it take to prepare a teaching statement for faculty appli-

Operation	Frequency	Tokens
Feasibility check	100%	0
Conflict detection	100%	0
Ambiguous duration	5%	50
Deadline interpretation	5%	50
Average per timeline		**5**

Table 6: Temporal Coordinator Costs

cation?" (depends on experience, field, institution type)

Deadline interpretation: "Application says 'reviewed on rolling basis'. What's the effective deadline?" (requires understanding of competitive dynamics)

3.5 Strategy Synthesizer Agent

3.5.1 Cognitive Role

The Strategy Synthesizer implements an executive decision-maker that integrates inputs from all other agents into coherent action recommendations. Its cognitive function is conflict resolution and strategic synthesis when agent recommendations are incompatible.

The agent serves as final arbiter of strategic direction: explore versus exploit, niche versus broad positioning, risk acceptance versus risk avoidance. It resolves tensions between competing objectives and makes meta-strategic pivots when current strategy fails.

Intelligence Type: Strategic reasoning with case-based learning. Uses episodic memory to identify analogous past situations and apply learned resolutions.

3.5.2 Memory Architecture

Working Memory

```
StrategicPosture {
    current_strategy: {
        type: {explore, exploit, hybrid},
        risk_tolerance: {conservative, moderate, aggressive},
        focus: {quality, quantity, balanced},
        active_until: datetime
    },

    current_plan: {
        priority_actions: [
            {
                action: string,
                deadline: datetime,
                rationale: string
            }
        ],
        deferred_actions: [
            {
                action: string,
                defer_until: datetime,
                reason: string
            }
```

```
        ]
    },

    compressed_representation: string  # <200 tokens
}
```

Listing 13: Working Memory Structure

Compressed Representation: Natural language summary of current strategic posture for quick context loading in LLM calls. Example: "Conservative quality-focused strategy targeting top-20

Episodic Memory

```
StrategyOutcomes {
    strategy_instance: {
        type: string,
        context: {
            market_regime: string,
            user_state: {snapshot},
            timestamp: datetime
        },
        actions_taken: [string],
        outcomes: [
            {
                action: string,
                result: {success, failure, neutral},
                impact: float
            }
        ],
        total_value: float,
        lessons_learned: string  # Compressed insight
    }
}
```

Listing 14: Episodic Memory Structure

Retention: All history maintained for case-based reasoning.

Compression: Quarterly summarization reduces storage by extracting general lessons from specific episodes.

Semantic Memory

```
StrategyHeuristics {
    pattern: string,
    condition: string,
    recommended_action: string,
    confidence: float,
    evidence_count: int,
    success_rate: float,
    learned_from: [episode_id]
}
```

Listing 15: Semantic Memory Structure

Examples:

- IF market_regime=buyer's AND user_competitiveness=high THEN strategy=quality_over_quantity
- IF advisor_risk=high AND alternatives=available THEN action=abort

• IF search_duration>6months AND interview_rate<10% THEN action=pivot_strategy

Update Frequency: Monthly distillation from episodic memory identifying reliable patterns.

3.5.3 Decision Logic

Algorithm 16 Straightforward Synthesis (Tier 1)

```
function SynthesizePlan(agent_inputs)
    state ← StateEstimator.get_current()
    opps ← OpportunityFilter.get_ranked()
    gaps ← InformationGap.get_critical_unknowns()
    timeline ← TemporalCoord.get_feasible_schedule()
    conflicts ← DetectConflicts(agent_inputs)
    if len(conflicts) = 0 then
        template ← RetrieveStrategyTemplate(state, opps)
        plan ← template.instantiate(opps[: 5], timeline, gaps)
        return plan
    else
        return ResolveConflicts(agent_inputs, conflicts)
    end if
end function
```

Conflict-Free Integration Token Cost: 100-150 tokens for template instantiation when no conflicts exist.

Algorithm 17 Conflict Resolution (Tier 2-3)

```
function ResolveConflicts(agent_inputs, conflicts)
    for conflict ∈ conflicts do
        similar ← EpisodicMemory.search(conflict.context, k = 3)
        if similar ≠ ∅ ∧ confidence(similar) > 0.7 then
            resolution ← ApplyHeuristic(conflict, similar)
            token_cost ← 0
        else
            resolution ← LLM_ResolveConflict(conflict, context)
            token_cost ← 250-500
            EpisodicMemory.add(conflict, resolution, context)
            UpdateHeuristics(conflict, resolution)
        end if
    end for
    return IntegratedPlan(resolutions)
end function
```

Conflict Resolution

Meta-Strategy Adaptation Token Cost: 500-1000 tokens for meta-strategy pivot. Invoked rarely (when current strategy demonstrably failing).

Algorithm 18 Meta-Strategy Pivot (Tier 3)

```
function AdaptMetaStrategy(outcomes)
    current ← StrategicPosture.current_strategy
    performance ← EvaluateStrategy(outcomes, current)
    if performance.below_threshold() then
        new ← LLM_MetaStrategyPivot(
            current, outcomes, market_regime, user_state)
        StrategicPosture.update(new)
        EmitStrategyChangeNotification(current, new)
    end if
end function
```

3.5.4 Conflict Resolution Powers

Final Decision Authority Strategy Synthesizer makes ultimate strategic direction choices. Can override all agents except:

- State Estimator's staleness veto (prevents hallucinated analyses)
- Temporal Coordinator's feasibility constraints (cannot violate physics)

Algorithm 19 Override Justification

```
function OverrideAgent(agent, original_rec, override_decision)
    justification ← GenerateJustification(
        original_rec, override_decision, self.reasoning)
    Log({
        overridden_agent : agent,
        original : original_rec,
        override : override_decision,
        justification : justification,
        timestamp : now()
    })
    MessageBus.send(RealityFeedback, {
        type : "OVERRIDE_EVENT",
        details : override_log
    })
    return override_decision
end function
```

Override Documentation Justification Cost: 25-50 tokens.

Purpose: Creates learning signal for Reality Feedback. If overrides consistently correct, validates Strategy Synthesizer judgment. If overrides consistently wrong, triggers recalibration.

3.5.5 Token Economics

Cost Distribution:

Monthly Allocation:

For 100 syntheses per month: 11,850 tokens

Synthesis Type	Frequency	Tokens	Expected
Straightforward (Tier 1)	80%	120	96.0
Cached conflict (Tier 0)	15%	0	0.0
Novel conflict (Tier 2)	4%	375	15.0
Meta-strategy (Tier 3)	1%	750	7.5
Average per synthesis			**118.5**

Table 7: Strategy Synthesizer Cost Distribution

For 500 syntheses per month: 59,250 tokens

Well within reactive query budget of 1,500,000 tokens per month.

3.6 Reality Feedback Agent

3.6.1 Cognitive Role

The Reality Feedback Agent implements an empirical validator that compares predictions to actual outcomes and updates system priors. Its cognitive function is calibration: detecting when reality diverges from predictions and triggering model updates.

The agent serves as the system's learning mechanism, closing the feedback loop between predictions and outcomes. Without this agent, the system would make systematic errors without self-correction.

Intelligence Type: Statistical analysis with pattern detection for systematic bias identification. Minimal language model usage.

3.6.2 Memory Architecture

Working Memory

```
PendingPredictions {
    predictions: [
        {
            prediction_id: string,
            agent_source: string,
            predicted_outcome: string,
            confidence: float,
            timestamp: datetime,
            resolution_date: datetime,
            user_id: string
        }
    ]
}
```

Listing 16: Working Memory Structure

Episodic Memory

```
OutcomeHistory {
    prediction_id: string,
    agent_source: string,
    predicted_outcome: string,
    predicted_confidence: float,
    actual_outcome: string,
```

```
    error_magnitude: float,
    error_direction: {over, under, correct},
    timestamp: datetime,
    context: {
        market_regime: string,
        user_characteristics: {snapshot}
    }
}
```

Listing 17: Episodic Memory Structure

Retention: Permanent (all outcomes logged for calibration analysis).

Purpose: Foundation for all calibration and learning. Enables detection of systematic biases and context-dependent accuracy patterns.

Semantic Memory

```
CalibrationCurves {
    agent_id: string,
    prediction_type: string,
    calibration_data: [
        {
            predicted_confidence: float,
            actual_accuracy: float,
            n_samples: int
        }
    ],
    systematic_biases: [
        {
            bias_type: string,
            magnitude: float,
            conditions: string
        }
    ],
    last_updated: datetime
}
```

Listing 18: Semantic Memory Structure

Update Frequency: Weekly batch processing.

3.6.3 Decision Logic

Outcome Processing Token Cost: 0 (pure statistical processing).

Bias Detection Token Cost: 0 for statistical tests, 200 tokens for root cause analysis when bias detected.

Frequency: Bias detection runs weekly. Root cause analysis triggered only when statistical significance achieved.

Credibility Weighting Credibility Threshold: 0.6 (below this, agent recommendations receive reduced weight in Strategy Synthesizer's integration).

Algorithm 20 Outcome Processing (Tier 0)

```
function ProcessOutcome(outcome_event)
    prediction ← PendingPredictions.get(outcome_event.id)
    error ← ComputeError(prediction, outcome_event.actual)
    OutcomeHistory.add({
        prediction : prediction,
        actual : outcome_event.actual,
        error : error,
        context : outcome_event.context
    })
    if |error| > ERROR_THRESHOLD then
        TRIGGER ErrorInvestigation(prediction.agent, error)
    end if
    CalibrationCurves.update(prediction.agent, prediction, error)
end function
```

Algorithm 21 Systematic Bias Detection

```
function DetectSystematicBias(agent_id)
    recent ← OutcomeHistory.filter(agent = agent_id, last_90_days)
    bias_tests ← {
        directional : TestDirectionalBias(recent),
        overconfidence : TestOverconfidence(recent),
        conditional : TestConditionalPatterns(recent)
    }
    detected ← Filter(bias_tests, λt : t.p_value < 0.05)
    if len(detected) > 0 then
        root_cause ← LLM_AnalyzeBias(agent_id, detected, recent)
        EmitRecalibration(agent_id, detected, root_cause)
    end if
end function
```

Algorithm 22 Agent Credibility Update (Tier 0)

```
function UpdateCredibility(agent_id, error)
    current ← AgentCredibilityWeights[agent_id]
    learning_rate ← 0.1
    new ← (1 − learning_rate) × current + learning_rate × (1 − error)
    AgentCredibilityWeights[agent_id] ← new
    if new < CREDIBILITY_THRESHOLD then
        EmitPerformanceAlert(agent_id, new)
    end if
end function
```

3.6.4 Conflict Resolution Powers

Forced Recalibration Authority Reality Feedback cannot override prospective decisions but can force recalibration of agents whose predictions are systematically wrong.

Algorithm 23 Forced Recalibration

```
function ForceRecalibration(agent_id, evidence)
    MessageBus.send(agent_id, {
        type : "RECALIBRATION_REQUIRED",
        evidence : evidence,
        mandatory : TRUE,
        deadline : now() + 7_days
    })
    ScheduleCheck(now() + 7_days, VerifyRecalibration(agent_id))
end function
function VerifyRecalibration(agent_id)
    if ¬recalibrated then
        DisableAgent(agent_id)
        AlertAdministrator(agent_id)
    end if
end function
```

Algorithm 24 Credibility-Weighted Integration

```
function ApplyCredibilityWeights(recommendations)
    weighted ← []
    for rec ∈ recommendations do
        agent ← rec.source_agent
        weight ← AgentCredibilityWeights[agent]
        weighted.append({
            recommendation : rec,
            credibility : weight,
            adjusted_confidence : rec.confidence × weight
        })
    end for
    return weighted
end function
```

Credibility Weight Application This mechanism allows the system to automatically down-weight recommendations from agents with poor track records without removing them entirely.

3.6.5 Token Economics

Operational Costs:

Efficiency Rationale:

Reality Feedback Agent is the most token-efficient agent because its core function (statistical analysis) requires no language model. LLM invocation occurs only when systematic bias is detected and human-interpretable explanation is needed.

Operation	Frequency	Tokens
Outcome processing	Per outcome	0
Calibration update	Weekly batch	0
Credibility update	Per outcome	0
Bias detection (tests)	Weekly batch	0
Root cause analysis	Quarterly (1-2 agents)	200
Monthly total		**400**

Table 8: Reality Feedback Agent Costs

4 System-Level Coordination

4.1 Message Bus Infrastructure

4.1.1 Central Coordinator Architecture

The message bus implements a centralized coordination point that manages all inter-agent communication, enforces rate limits, and handles batching.

```
MessageBus {
    queues: {agent_id: [message]},

    routing_rules: {
        message_type: [eligible_recipient_agents]
    },

    rate_limiter: {
        current_rpm: int,
        current_tpm: int,
        window_start: datetime
    },

    batch_scheduler: {
        pending_requests: [api_call],
        next_batch_time: datetime,
        min_cooldown: 5 minutes
    }
}
```

Listing 19: Message Bus Structure

4.1.2 Routing Implementation

Non-LLM messages (Tier 0 operations) are processed immediately without batching. LLM-requiring messages are queued for batch processing.

4.1.3 Interaction Depth Enforcement

MAX_DEPTH: 2 (User → Agent → Agent)

Circular dependency detection: Message bus maintains call stack for current request. Detects if Agent A invokes Agent B which attempts to invoke Agent A.

Algorithm 25 Message Routing

```
function RouteMessage(message)
    recipients ← RoutingRules[message.type]
    for recipient ∈ recipients do
        queues[recipient].append(message)
    end for
    if message.requires_llm then
        batch_scheduler.pending_requests.append(message)
    else
        ProcessImmediately(message)
    end if
end function
```

Algorithm 26 Interaction Depth Limit Enforcement

```
function EnforceDepthLimit(message)
    depth ← message.interaction_depth
    if depth > MAX_DEPTH then
        RejectMessage(message, "Exceeds_interaction_depth_limit")
        LogViolation(message)
        return REJECTED
    end if
    return ACCEPTED
end function
```

4.2 Rate Limit Management

4.2.1 Token Budget Tracking

Algorithm 27 Token Budget Management

```
function ManageTokenBudget
    usage_1min ← ∑ tokens_consumed(last_1_minute)
    usage_1day ← ∑ tokens_consumed(last_24_hours)
    LIMIT_TPM ← 12,000
    LIMIT_TPD ← 100,000
    available_minute ← LIMIT_TPM − usage_1min
    available_day ← LIMIT_TPD − usage_1day
    safe_available ← min(available_minute, available_day) × 0.9
    return safe_available
end function
```

Safety Factor: Use only 90% of available tokens to maintain buffer for usage spikes and estimation errors.

4.2.2 Request Batching Strategy

Minimum Cooldown: 5 minutes between batches ensures requests are well-distributed across rate limit windows.

Algorithm 28 Batch Request Scheduling

```
function ScheduleAPICall(request)
    batch_scheduler.pending.append(request)
    if TimeSinceLastBatch() > 5_minutes then
        BatchDispatch()
    else if request.priority = "urgent" ∧ AvailableTokens() > request.tokens then
        ImmediateDispatch(request)
    else
        WAIT for next batch
    end if
end function
function BatchDispatch
    available ← ManageTokenBudget()
    prioritized ← Sort(pending, key = λr : r.priority)
    batch ← []
    tokens_used ← 0
    for req ∈ prioritized do
        if tokens_used + req.tokens ≤ available then
            batch.append(req)
            tokens_used ← tokens_used + req.tokens
        end if
    end for
    for req ∈ batch do
        ExecuteAPICall(req)
    end for
    next_batch_time ← now() + 5_minutes
end function
```

Algorithm 29 Rate Limit Degradation Handling

```
function HandleRateLimitApproach
    usage_percent ← current_usage/daily_limit
    if usage_percent > 0.9 then
        DisableProactiveUpdates()
        QueueNonUrgentRequests()
        ServeCacheAggressively()
        NotifyUser("Reduced_capability_mode")
    else if usage_percent > 0.7 then
        IncreaseBatchCooldown(from = 5min, to = 10min)
        DeferLowPriorityUpdates()
    else
        StandardBatchingPolicy()
    end if
end function
```

4.2.3 Degradation Strategy

Degradation Levels:

- 70-90% usage: Conservative mode (longer batch intervals)
- 90-100% usage: Emergency mode (only critical operations)
- 100% usage: Cache-only mode (no API calls until limit resets)

4.3 Pre-Computation and Caching

4.3.1 Batch Processing Schedule

Analysis Type	Frequency	Token Budget
Field vitality snapshots	Yearly (Sept)	200,000
Funding regime analysis	Quarterly	200,000
Market regime detection	Monthly	200,000
Institutional trajectories	Quarterly	150,000
Outcome integration	Weekly	75,000
Model calibration	Monthly	62,500
Heuristic compression	Quarterly	50,000

Table 9: Batch Processing Schedule and Budgets

Total Batch Budget: 750,000 tokens per month (25% of total budget).

4.3.2 Cache Structure

```
PrecomputedAnalyses {
    field_vitality: {
        field_id: {
            status: {ascending, stable, declining},
            indicators: {funding, hiring, publications, enrollment},
            confidence: float,
            computed_date: datetime,
            valid_until: datetime  # +12 months
        }
    },

    funding_regimes: {
        field_id: {
            trajectory: {increasing, stable, declining},
            emerging_priorities: [string],
            confidence: float,
            computed_date: datetime,
            valid_until: datetime  # +3 months
        }
    },

    market_regimes: {
        field_id: {
            regime: {buyer, seller, transitional},
```

```
            indicators: {
                postings,
                applications_per_posting,
                time_to_decision
            },
            confidence: float,
            computed_date: datetime,
            valid_until: datetime  # +1 month
        }
    },

    institutional_trajectories: {
        institution_id: {
            trajectory: {ascending, stable, declining},
            indicators: {hiring, funding, enrollment},
            risk_factors: [string],
            confidence: float,
            computed_date: datetime,
            valid_until: datetime  # +3 months
        }
    }
}
```

Listing 20: Pre-Computed Analysis Cache

4.3.3 Cache Serving

Algorithm 30 Pre-Computed Analysis Retrieval

```
function GetFieldVitality(field_id)
    cached ← PrecomputedAnalyses.field_vitality[field_id]
    if now() < cached.valid_until then
        return cached                                   ▷ 0 tokens
    else
        QueueForNextBatchUpdate(field_id)
        return cached_with_staleness_warning
    end if
end function
```

Staleness Handling: When cached data exceeds validity period, return cached result with staleness warning rather than blocking on fresh computation. Queue field for next batch update cycle.

4.4 Memory Compression

4.4.1 Episodic Memory Compression

Example Compression:

Original Episode (500 tokens):

> User applied to postdoc at Lab X on 2025-01-15. Application included cover letter emphasizing computational methods, CV with 8 publications, 3 reference letters

Algorithm 31 Episodic Memory Compression

```
function CompressEpisodicMemory
    old ← EpisodicMemory.filter(age > 30_days)
    for episode ∈ old do
        summary ← CreateSummary(episode)                ▷ 10:1 compression
        EpisodicMemory.replace(episode, summary)
    end for
end function
```

> from advisors A, B, C. Received interview invitation 2025-02-10. Interview conducted 2025-02-25 with PI and 2 lab members. Topics discussed: research interests, methodological expertise, lab culture fit, funding situation. Offered position 2025-03-05 with salary $65K, 2-year term, startup package $10K. Accepted offer 2025-03-12 after negotiating startup to $15K.

Compressed Summary (50 tokens):

> Lab X postdoc (2025-01): computational methods focus. Outcome: Accepted, $65K, 2yr, $15K startup (negotiated from $10K). Time-to-offer: 7 weeks. Lesson: Negotiation successful for startup funds.

Compression Ratio: 10:1

Information Retained: Outcome, timing, key decision factors, negotiation result.

Information Discarded: Detailed application contents, interview specifics.

4.4.2 Semantic Memory Pruning

Algorithm 32 Semantic Memory Pruning

```
function PruneSemanticMemory
    for rule ∈ SemanticMemory do
        if rule.access_count < THRESHOLD ∧ rule.age > 90_days then
            if rule.success_rate < 0.6 then
                SemanticMemory.remove(rule)
            else
                SemanticMemory.archive(rule)            ▷ De-prioritize
            end if
        end if
    end for
end function
```

Threshold: 5 accesses per quarter.

Rationale: Rules accessed fewer than 5 times in 90 days are unlikely to be useful. Poor-performing rules (success rate below 60%) are removed. Successful but rarely-used rules are archived for potential future reactivation.

Algorithm 33 API Unavailability Handling

```
function HandleAPIUnavailable
    EnableCacheOnlyMode()
    QueueManager.pause_proactive_updates()
    UserNotification({
      type : "DEGRADED_MODE",
      reason : "API_unavailable",
      capabilities : "Read_only_cached_data",
      expected_restoration : "When_API_resumes"
    })
    ScheduleRetry(initial = 1min, max = 30min, exponential_backoff)
end function
```

4.5 Failure Handling

4.5.1 API Unavailability

4.5.2 Cache-Only Operation

Algorithm 34 Cache-Only Mode

```
function ServeFromCache(user_query)
    cached ← Cache.get(hash(user_query))
    if cached ∧ ¬too_stale(cached) then
        return {
          response : cached.response,
          metadata : {
            source : "cached",
            computed : cached.timestamp,
            freshness : "API_unavailable"
          }
        }
    else
        return {
          response : "Cannot_analyze_without_API",
          recommendation : "Retry_when_service_resumes"
        }
    end if
end function
```

4.5.3 Rate Limit Exhaustion

4.5.4 Agent Failure

Critical Agents: State Estimator failure causes complete system unavailability. Strategy Synthesizer failure causes degradation to rule-based synthesis. Other agent failures cause partial capability loss but system remains operational.

Algorithm 35 Rate Limit Exhaustion Handling

```
function HandleRateLimitExhausted
    urgent ← Filter(queue, λr : r.priority = "urgent")
    normal ← Filter(queue, λr : r.priority = "normal")
    low ← Filter(queue, λr : r.priority = "low")
    for req ∈ urgent do
        cached ← AttemptCacheServe(req)
        if cached then
            RespondFromCache(req, cached)
        else
            QueueForNextWindow(req)
            NotifyUser(req, "Rate_limit_reached_queued")
        end if
    end for
    DeferUntilLimitReset(normal + low)
end function
```

Algorithm 36 Agent Failure Handling

```
function HandleAgentFailure(agent_id, error)
    LogFailure(agent_id, error, timestamp)
    if agent_id = "State_Estimator" then
        EnterEmergencyMode()                    ▷ Cannot operate without state
    else if agent_id = "Opportunity_Filter" then
        EnableBypassMode(agent_id)
        NotifyUser("Reduced_filtering")
    else if agent_id = "Strategy_Synthesizer" then
        EnableFallbackSynthesizer()
    else
        DisableAgent(agent_id)
        NotifyMaintenance(agent_id, error)
    end if
    ScheduleRestart(agent_id, delay = 5min)
end function
```

5 Computational Constraints Shaping Architecture

5.1 Constraint-Driven Design Methodology

Traditional software architecture begins with functional requirements and optimizes for performance afterward. This architecture inverts that process: computational constraints are primary, functional capabilities are derived within those bounds.

This inversion is necessitated by the hard physical limits of free-tier API access. Unlike soft performance targets that can be exceeded temporarily, token budgets are hard limits: exceeding monthly allocation means complete service interruption.

5.2 Token Budget as Architectural Constraint

5.2.1 Budget Allocation Strategy

The monthly budget of 3,000,000 tokens dictates three-way partitioning:

Regime Updates (25%, 750,000 tokens): Pre-computed analyses that serve many users. High initial cost amortized across user base. Updates infrequent (monthly to yearly depending on data volatility).

User Queries (50%, 1,500,000 tokens): Reactive responses to user requests. Must be rationed across expected user base. Average query budget (250 tokens) determines maximum monthly active users (6,000 users).

Learning and Calibration (25%, 750,000 tokens): System improvement through outcome integration and model calibration. Investment in future capability improvement.

This allocation reflects strategic priorities: serving users is paramount (50%), but investing in future capability (25%) and maintaining high-quality background intelligence (25%) are essential for long-term viability.

5.2.2 Tiered Processing Architecture

Token budget constraints necessitate tiered processing where operations are classified by cost:

Tier 0 (0 tokens): Algorithmic processing requiring no language model. Target: 95% of operations.

Design Implication: Agents must implement algorithmic fallbacks for routine cases. State Estimator uses threshold comparisons. Opportunity Filter uses weighted scoring. Temporal Coordinator uses constraint satisfaction algorithms. Only genuinely ambiguous cases invoke language models.

Tier 1 (50-100 tokens): Cached template application requiring minimal language model usage. Target: 3% of operations.

Design Implication: System must accumulate and maintain template library. Strategy Synthesizer builds catalog of strategy templates. Information Gap builds uncertainty templates. Templates enable rapid response to common situations.

Tier 2 (200-400 tokens): Novel analysis requiring moderate language model usage. Target: 1.5% of operations.

Design Implication: Reserved for genuinely novel situations without precedent. Information

Gap enumerating uncertainties for unprecedented decision types. Strategy Synthesizer resolving conflicts without historical analogues.

Tier 3 (500-1000 tokens): Complex synthesis requiring extensive language model usage. Target: 0.5% of operations.

Design Implication: Reserved for meta-strategic decisions. Strategy Synthesizer pivoting overall approach when current strategy fails. Root cause analysis when systematic biases detected.

5.2.3 Cache-First Architecture

Token constraints make caching architectural, not optimization:

State Estimator Staleness Checking: Every request begins with staleness check. If state unchanged, cached analysis served at zero token cost. This architectural decision prevents redundant computation and enables sustainable operation.

Without staleness checking: Every user query triggers fresh analysis consuming 250 tokens average. Monthly capacity: 12,000 queries.

With staleness checking: 80% of queries served from cache (0 tokens), 20% trigger fresh analysis (250 tokens). Monthly capacity: 60,000 queries.

Factor of 5 increase in capacity through architectural caching.

Pre-Computed Regime Analyses: Field vitality, funding regimes, market conditions, institutional trajectories are computed in batch and cached. Users query cache, not trigger fresh computation.

Cost without pre-computation: Each user query about field vitality requires fresh analysis (500 tokens). For 100 users: 50,000 tokens.

Cost with pre-computation: One batch analysis (200,000 tokens yearly) serves unlimited queries (0 tokens per query). Amortized over 12 months: 16,667 tokens per month for unlimited field vitality queries.

Break-even: After 33 queries, pre-computation becomes cheaper. System expects thousands of queries, making pre-computation overwhelmingly superior.

5.3 Hardware Constraints Shaping Architecture

5.3.1 Consumer Hardware Limitations

Requirement to run on consumer hardware (8-16GB RAM, CPU-only) shapes memory architecture and algorithm selection.

Memory Architecture:

Episodic Memory Compression: Without compression, episodic memory grows unbounded. Strategy outcomes at 500 tokens per episode, 100 episodes per month: 50,000 tokens per month, 600,000 tokens per year. At 1 byte per token: 600KB per year, 6MB over 10 years.

With compression after 30 days (10:1 ratio): 50 tokens per episode. Year 1: 50,000 tokens detailed + 550,000 tokens compressed (55,000 tokens). Total: 105,000 tokens (105KB). Sustainable indefinitely.

Semantic Memory Pruning: Without pruning, low-value rules accumulate. With pruning (quarterly removal of rules accessed <5 times with $<60\%$ success rate): Semantic memory stabilizes

at 1,000 high-value rules (approximately 100KB).

Algorithm Selection:

Constraint Satisfaction Scheduling: Temporal Coordinator uses greedy earliest-deadline-first heuristic, not optimal integer programming. Greedy algorithm runs in O(n log n) time, requires minimal memory. Optimal algorithm requires exponential time and memory.

Trade-off: Accept suboptimal schedules to maintain feasibility on consumer hardware. In practice, greedy algorithm finds near-optimal solutions for typical workloads.

Statistical Calibration: Reality Feedback uses lightweight statistical tests (directional bias, overconfidence) not deep learning. Statistical tests require constant memory regardless of data volume. Deep learning models require GPU and gigabytes of model weights.

Trade-off: Accept simpler bias detection to maintain CPU-only operation. Statistical tests detect major biases; deep learning might detect subtle patterns but exceeds hardware constraints.

5.4 Rate Limit Constraints Shaping Coordination

5.4.1 Request Batching Necessity

Groq API limits: 30 requests per minute, 12,000 tokens per minute.

Without batching: Agents make individual API calls as needed. Temporal distribution unpredictable. Risk of bursting through per-minute limits.

With batching: Requests accumulate for 5 minutes, dispatched in batch. Controlled temporal distribution. Predictable load on API.

Architectural Impact: Agents cannot make synchronous API calls. All LLM-requiring operations are asynchronous. Agent submits request to message bus, continues other work, receives callback when result available.

This asynchrony complicates agent design but is necessary for rate limit compliance.

5.4.2 Degradation Hierarchy

Rate limit constraints necessitate explicit degradation hierarchy defining what to disable under resource pressure:

90%+ Usage (Emergency Mode):

- Disable: Proactive updates, low-priority requests, exploratory analyses
- Enable: Cache-only serving, request queueing
- User Impact: Degraded freshness, delayed responses, but core functionality maintained

70-90% Usage (Conservative Mode):

- Disable: Background optimizations, experimental features
- Modify: Increase batch intervals (5min → 10min)
- User Impact: Slightly slower responses, minimal functional impact

<70% Usage (Normal Mode):

- Enable: All features, standard batch intervals
- User Impact: Full functionality, optimal performance

Without explicit degradation hierarchy: System operates normally until limit exhausted, then crashes completely.

With explicit degradation hierarchy: System gracefully reduces capability under pressure, maintains core functionality even at limit.

5.5 Data Availability Constraints Shaping Capabilities

5.5.1 Public Data Only

Constraint: System limited to publicly available data (no proprietary databases, no authenticated access to private data).

Architectural Impact:

Advisor Toxicity Prediction: Cannot access confidential student complaints, exit interviews, or department internal communications. Must infer quality from public signals:

- Time-to-degree (longer = potential dysfunction)
- Authorship patterns (advisor always first author = credit appropriation concern)
- Turnover rates (high = retention problems)
- Publication gaps (sudden silence = crisis indicator)

This constraint shapes capability: System provides probabilistic risk assessment, not definitive judgment. Cannot match quality of insider knowledge but provides value above zero information.

Field Vitality Assessment: Cannot access proprietary funding databases or confidential hiring data. Must use public proxies:

- NSF award database (public funding trends)
- Academic job wiki (hiring volume)
- Google Scholar (publication and citation trends)
- University websites (enrollment numbers)

This constraint shapes update frequency: Public data updated yearly (NSF awards) to quarterly (hiring wikis). System cannot provide real-time intelligence but can provide leading indicators 3-5 years ahead of consensus.

5.5.2 User-Contributed Outcome Data

Constraint: Causal inference requires outcome data, but outcome data arrives years after initial decisions.

Architectural Impact:

Bootstrapping Problem: Year 1 system has no outcome data. Counterfactual analysis impossible. System must provide value through non-causal features (field vitality, advisor signals, negotiation leverage).

Network Effects Delay: Value increases with user base and data accumulation. Year 3: First cohort outcomes arrive, enable basic counterfactual analysis. Year 5: Sufficient data for confident causal claims.

This shapes business model: Early adopters receive less value from counterfactual features. Must be retained through other features until network effects materialize.

Data Quality Dependency: Outcome quality depends on user honesty and completeness. System must validate outcomes, detect systematic reporting bias, handle missing data.

Reality Feedback Agent designed to handle noisy outcome data: Statistical tests detect impossible outcomes, cross-validation identifies systematic misreporting, multiple imputation handles missingness.

5.6 Incentive Alignment Constraints Shaping Features

5.6.1 No Institutional Revenue

Constraint: System cannot accept institutional funding to preserve ability to critique universities, companies, advisors.

Architectural Impact:

Sustainable Features: Features must be valuable enough to justify user subscription without institutional subsidy. Cannot compete on job volume (Indeed has institutional relationships for exclusive postings) or comprehensive coverage (LinkedIn has network effects from institutional partnerships).

Must compete on truth-telling: Advisor toxicity prediction, prestige arbitrage detection, field collapse warnings. Features institutions would never fund because they contradict institutional interests.

Cost Structure: Without institutional revenue, must operate within extremely low cost structure. Free-tier API access mandatory, not optional. Cannot scale to paid tiers when user base grows.

This makes computational efficiency existential: System viability depends on maintaining free-tier operation indefinitely. Architectural decisions (caching, pre-computation, tiering) are not optimizations but survival requirements.

5.6.2 Truth-Telling Mandate

Constraint: System must optimize for user outcomes, not engagement metrics or platform retention.

Architectural Impact:

Stopping Rules: Strategy Synthesizer implements Bayesian stopping rules that tell users to stop searching. LinkedIn/Indeed cannot implement this (reduces engagement). This feature requires explicit architectural support: stopping rule computation, user notification, system shutdown for that user.

Field Collapse Warnings: System warns users their chosen field is declining. University

career services cannot do this (institutional loyalty). ResearchGate cannot do this (contradicts researcher optimism). This feature requires Field Collapse Detector agent with independence from institutional constraints.

Advisor Toxicity Prediction: System flags advisors as high-risk. Career services cannot do this (internal politics). This feature requires legal review (defamation risk) and careful methodology (public signals only, fact-based analysis, confidence intervals).

Without truth-telling mandate in architectural requirements, features would be watered down to avoid uncomfortable truths. Mandate embedded in agent cognitive roles and conflict resolution powers.

5.7 Synthesis: Constraint-Driven Architecture Principles

The architecture embodies five constraint-driven principles:

Principle 1 (Inversion): Constraints precede capabilities. Design begins with token budget, derives functions within bounds. Not: design functions, optimize for constraints afterward.

Principle 2 (Amortization): Expensive computations must be amortized across many users or uses. Pre-computed regime analyses serve unlimited queries. Template libraries enable zero-cost application to routine cases.

Principle 3 (Hierarchical Degradation): System must operate gracefully under resource pressure. Explicit degradation modes defined. Core functionality preserved even at resource limits.

Principle 4 (Lazy Evaluation): Default action is no computation. Staleness checks precede analysis. Cached results preferred. Fresh computation requires justification.

Principle 5 (Algorithmic Primacy): Algorithms consume zero tokens and are preferred over language models. Language models reserved for genuinely ambiguous cases requiring semantic understanding.

These principles are unusual in modern software architecture where computational resources are assumed abundant. Constraint-driven architecture accepts scarcity as fundamental and designs accordingly.

6 Agent Interaction Examples

6.1 Example 1: Straightforward Query (Tier 1)

6.1.1 User Query

"Should I keep applying or accept my current offer?"

6.1.2 Agent Execution Trace

Step 1: State Estimator (0 tokens)

```
INPUT: User query received

OPERATION: Staleness check
- Last state update: 3 days ago
- State changes since: None
- Material change threshold: Not exceeded

DECISION: State current, cache valid

OUTPUT: Return cached state (0 tokens consumed)
```

Step 2: Strategy Synthesizer (100 tokens)

```
INPUT:
- Current state from State Estimator
- Query type: "stopping_rule_decision"

OPERATION: Template matching
- Match found: "bayesian_stopping_rule" template
- Context:
  * Application history: 10 applications, 2 offers
  * Market regime: buyer's market (cached, 0 tokens)
  * Best offer: $65K, State University
  * User profile: 75th percentile for field

DECISION: Apply template

TEMPLATE INSTANTIATION (100 tokens):
"Based on your search history (10 applications, 2 offers),
current market regime (buyer's market, low offer rates),
and your best offer ($65K, State University):

RECOMMENDATION: ACCEPT current offer.

RATIONALE: Your offer quality is at 75th percentile for
your profile. Expected value of continued search is
negative: low probability of better offer (15% estimated),
high search costs (time, stress, opportunity cost).

Bayesian stopping rule threshold: REACHED.

Confidence: 0.8"

OUTPUT: User response (100 tokens consumed)
```

Step 3: Reality Feedback (0 tokens)

```
OPERATION: Log prediction
- Prediction: User will accept State University offer
- Confidence: 0.8
- Resolution date: +2 weeks
- Track for outcome validation

OUTPUT: Prediction logged (0 tokens consumed)
```

6.1.3 Summary

Total Tokens: 100

Response Time: <5 seconds

Agent Activation: State Estimator (staleness check), Strategy Synthesizer (template application), Reality Feedback (prediction logging)

Tier Classification: Tier 1 (cached template application)

6.2 Example 2: Complex Query with Conflicts (Tier 2)

6.2.1 User Query

"Help me choose between postdoc at Harvard (PI has toxicity signals) and tenure-track at State University."

6.2.2 Agent Execution Trace

Step 1: State Estimator (0 tokens)

```
INPUT: User query received

OPERATION: Staleness check
- Last state update: Current
- New decision point: Material change

DECISION: State current, proceed

OUTPUT: Return current state (0 tokens)
```

Step 2: Information Gap Analyzer (50 tokens)

```
INPUT:
- Decision type: Harvard postdoc vs State tenure-track
- Opportunities: [Harvard postdoc, State TT]

OPERATION: Uncertainty enumeration
- Template match: "postdoc_vs_TT_comparison" (0 tokens)
- Customize for Harvard toxicity concern (50 tokens):

CUSTOMIZATION (50 tokens):
"Key uncertainties for Harvard/State comparison:

1. Harvard PI actual mentorship quality (HIGH EVOI, resolvable)
   Method: Interview current students off-the-record
```

```
2. State University tenure probability (MEDIUM EVOI, partially
   resolvable)
   Method: Review department tenure rates, recent cases

3. Career trajectory differences (MEDIUM EVOI, partially resolvable)
   Method: Track outcomes of similar choices

4. Option value differential (HIGH EVOI, resolvable)
   Method: Model future paths from each choice

CRITICAL: Harvard toxicity signals elevate mentorship
quality from standard uncertainty to decision-blocking gap."

OUTPUT: Uncertainty map with EVOI rankings (50 tokens)
```

Step 3: Strategy Synthesizer (350 tokens)

```
INPUT:
- State from State Estimator
- Uncertainties from Information Gap
- Decision: Harvard vs State

OPERATION: Conflict detection
- Conflict identified: Prestige vs Risk trade-off
- Template match attempted: No exact match for toxicity scenario
- Heuristic search: No confident heuristic (requires LLM)

LLM INVOCATION (350 tokens):

PROMPT:
"Decision conflict analysis:

OPTION A (Harvard postdoc):
- Prestige: Highest tier
- Risk: PI toxicity signals detected (time-to-degree 7.2yr
  vs field avg 5.5yr, authorship patterns concerning)
- Career ceiling: High if successful
- Career floor: Low if toxic environment

OPTION B (State tenure-track):
- Prestige: Mid-tier (regional university)
- Risk: Lower (permanent position, no advisor dependency)
- Career ceiling: Moderate (teaching-focused institution)
- Career floor: Moderate (tenure probability 78%)

USER CONTEXT:
- Profile: Strong (can succeed at either)
- Market: Buyer's (opportunities scarce)
- Alternatives: None currently

ANALYZE: Which choice optimizes long-term expected utility
given risk-return trade-off and user context?"

LLM RESPONSE (300 tokens):
"RECOMMENDATION: Accept State tenure-track.

ANALYSIS:
```

```
Prestige premium calculation:
- Harvard brand worth ~2-3 years career advantage
  (better first industry transition, conference invitations)
- State acceptable (regional but R2 research active)

Risk assessment:
- Harvard downside: Toxic advisor = career catastrophe
  * Delayed degree (opportunity cost: 2 years)
  * Damaged confidence and mental health
  * Limited publication record if credit appropriated
  * Probability: 40% (based on public signals)

- State downside: Limited prestige ceiling
  * Reduced industry transition value
  * Harder to move to R1 later
  * Probability: 80% (structural, not failure)

Expected utility:
- Harvard: 0.6 * (high_outcome) + 0.4 * (catastrophic) = NEGATIVE
- State: 0.78 * (tenure_success) + 0.22 * (denial) = POSITIVE

CRITICAL FACTOR: Downside protection prioritized.
User has strong profile; can build prestige from stable base.
Toxic advisor risk not justified by prestige premium.

Confidence: 0.7 (uncertainty about actual PI toxicity level)"

OUTPUT: Recommendation with analysis (350 tokens consumed)
```

Step 4: Reality Feedback (0 tokens)

```
OPERATION: Log prediction
- Prediction: User will accept State University
- Confidence: 0.7
- Rationale: Risk-adjusted expected utility
- Track for outcome validation

OUTPUT: Prediction logged (0 tokens)
```

6.2.3 Summary

Total Tokens: 400 (50 customization + 350 conflict resolution)

Response Time: 10-15 seconds

Agent Activation: State Estimator, Information Gap (customization), Strategy Synthesizer (novel conflict), Reality Feedback

Tier Classification: Tier 2 (novel analysis)

6.3 Example 3: Meta-Strategy Pivot (Tier 3)

6.3.1 User Query

"I've applied to 50 positions with 0 interviews. What am I doing wrong?"

6.3.2 Agent Execution Trace

Step 1: State Estimator (0 tokens)

```
INPUT: User query with outcome data

OPERATION: State update
- New data: 50 applications, 0 interviews
- Material change: YES (significant outcomes)
- Update state with application history

OUTPUT: Updated state, propagate to downstream agents (0 tokens)
```

Step 2: Opportunity Filter (0 tokens)

```
INPUT: State delta from State Estimator

OPERATION: Recalibration trigger
- User success rate: 0/50 = 0%
- Expected success rate: ~15% for user profile
- Systematic mismatch detected

DECISION: Trigger recalibration

OUTPUT: Recalibration request to Reality Feedback (0 tokens)
```

Step 3: Reality Feedback (200 tokens)

```
INPUT: 50 applications, 0 interviews

OPERATION: Rejection pattern analysis

STATISTICAL ANALYSIS (0 tokens):
- Interview rate: 0% (user) vs 15% (expected)
- Deviation: -15 percentage points
- Statistical significance: p < 0.001
- Systematic problem confirmed

ROOT CAUSE ANALYSIS (200 tokens):

PROMPT:
"User applied to 50 positions, received 0 interviews.
Expected rate for profile: 15% (7-8 interviews).

Possible causes:
1. User overestimating competitiveness
2. Opportunity Filter scoring miscalibrated
3. Application materials deficient
4. Market regime worse than estimated
5. User targeting wrong opportunity tier

EVIDENCE:
- Market regime data: Confirmed buyer's market (not underestimated)
- User profile: Credentials appear competitive for field
- Filter scoring: Opportunities scored 0.7-0.8 (high confidence)

ANALYZE: Most likely root cause and recommended intervention?"

LLM RESPONSE (150 tokens):
"ROOT CAUSE: Opportunity Filter over-optimistic.
```

```
EVIDENCE:
- Filter predicted 70-80% match quality
- Actual outcomes: 0% success rate
- Systematic overestimation of ~0.7-0.8 points

LIKELY MECHANISM:
Filter targeting opportunities too competitive for user profile.
Scoring algorithm weights credentials heavily but underweights
competitive positioning (percentile within applicant pool).

RECOMMENDATION:
1. Recalibrate Filter to target 50th-75th percentile institutions
   (was targeting 90th+ percentile)
2. Increase application volume at lower tier (3x)
3. User should seek external review of materials (to rule out
   deficiency)

EXPECTED IMPROVEMENT: Interview rate 15-20% after recalibration."

OUTPUT: Root cause diagnosis, recalibration parameters (200 tokens)
```

Step 4: Opportunity Filter (0 tokens)

```
INPUT: Recalibration parameters from Reality Feedback

OPERATION: Update scoring weights
- Decrease prestige tier weight: 0.3     0.15
- Increase competitive positioning weight: 0.2     0.35
- Lower acceptance threshold: 0.6     0.5

OUTPUT: Updated filter rules (0 tokens)
```

Step 5: Strategy Synthesizer (750 tokens)

```
INPUT:
- Current strategy: "Quality-focused, selective applications"
- Performance: 0/50 = catastrophic failure
- Root cause: Filter miscalibration + targeting too high

OPERATION: Meta-strategy pivot required

LLM INVOCATION (750 tokens):

PROMPT:
"Current strategy failing. 50 applications, 0 interviews.

ROOT CAUSE: Targeting opportunities too competitive for profile.

CONTEXT:
- Market regime: Buyer's (confirmed, not improving)
- User profile: Competitive but not elite tier
- Current approach: Quality over quantity
- Filter: Now recalibrated to lower tier

DESIGN: New strategy for next phase (10-20 applications).
Consider: target tier, application volume, material review,
timeline adjustment, risk tolerance."
```

```
LLM RESPONSE (700 tokens):
"NEW STRATEGY: Expanded Tier, Increased Volume

STRATEGIC PIVOT:

1. TARGET TIER ADJUSTMENT:
   OLD: Top 20% institutions (90th+ percentile)
   NEW: 40th-75th percentile institutions

   RATIONALE: User competitive for this tier, current
   tier resulted in 100% rejection. Better to succeed
   at good institution than fail at great institution.

2. VOLUME INCREASE:
   OLD: Selective (50 applications to top tier)
   NEW: 3x volume at appropriate tier (30 applications)

   RATIONALE: At 15% expected interview rate, need 20-30
   applications to generate 3-5 interviews (minimum for
   choice optionality).

3. MATERIAL REVIEW:
   Seek external review of CV, research statement, cover letter.
   Current materials may have deficiencies not detected.
   Cost: 2-4 hours, high potential value.

4. TIMELINE:
   Re-evaluate after 10 new applications at adjusted tier.
   If interview rate reaches 15-20%, continue strategy.
   If interview rate still <10%, deeper problem exists
   (likely materials, not targeting).

5. PSYCHOLOGICAL ADJUSTMENT:
   Reframe from "top institutions" to "best fit institutions."
   User value not determined by institutional rank.

EXPECTED OUTCOMES:
- Interview rate: 15-20% (vs 0% previous)
- Interviews generated: 3-6 (from 30 applications)
- Confidence: 0.65 (multiple assumptions, market volatile)

RISK: If materials deficient, tier adjustment alone insufficient.
External review critical to rule out this failure mode."

OUTPUT: Comprehensive strategy pivot (750 tokens consumed)
```

Step 6: Reality Feedback (0 tokens)

```
OPERATION: Log meta-strategy change
- Old strategy: Quality-focused, top-tier targeting
- New strategy: Expanded tier, volume increase
- Predicted improvement: 0%     15-20% interview rate
- Track for validation after 10 new applications

OUTPUT: Strategy change logged (0 tokens)
```

6.3.3 Summary

Total Tokens: 950 (200 root cause + 750 strategy pivot)

Response Time: 25-30 seconds

Agent Activation: State Estimator (update), Opportunity Filter (recalibration), Reality Feedback (root cause), Strategy Synthesizer (meta-pivot)

Tier Classification: Tier 3 (meta-strategy pivot)

Learning: Opportunity Filter recalibrated based on outcome data. Future users with similar profiles will benefit from improved targeting.

6.4 Token Economics Across Examples

Example	Tokens	Tier	Time (sec)
Straightforward query	100	1	<5
Complex conflict	400	2	10-15
Meta-strategy pivot	950	3	25-30
Weighted Average	**250**		

Table 10: Token Consumption by Query Complexity

Assuming distribution: 80% Tier 1, 15% Tier 2, 5% Tier 3:

Expected tokens per query: $0.80 \times 100 + 0.15 \times 400 + 0.05 \times 950 = 80 + 60 + 47.5 = 187.5$

Actual average (250 tokens) includes additional overhead: state propagation, uncertainty analysis, prediction logging.

Monthly capacity at 1,500,000 token reactive budget: 6,000 queries.

7 Conclusion

7.1 Architectural Summary

This document specifies a multi-agent architecture for career strategy intelligence operating under severe computational constraints. The architecture inverts traditional design methodology: beginning with hard resource limits (3,000,000 monthly tokens, consumer hardware, free-tier API access) and deriving functional capabilities within those bounds.

The system comprises seven specialized agents coordinated through a central message bus. Each agent implements a distinct cognitive role with isolated memory and explicit conflict resolution powers. Communication occurs through structured data objects, not natural language, minimizing token consumption.

7.2 Key Architectural Innovations

Constraint-Driven Design: Computational limits are primary architectural constraints, not afterthought optimizations. Token budget allocation (25% regime updates, 50% user queries, 25% learning) reflects strategic priorities and enables sustainable operation.

Tiered Processing: Operations classified into four cost tiers (0, 50-100, 200-400, 500-1000 tokens) with 95% of operations occurring at Tier 0 through algorithmic processing. Language models reserved for genuinely ambiguous cases.

Cache-First Architecture: Staleness checking precedes all computation. Pre-computed regime analyses amortize expensive computations across user base. Template libraries enable zero-cost application to routine cases.

Graceful Degradation: Explicit degradation modes defined for resource pressure. System maintains core functionality even at rate limits through cache-only operation and request queueing.

Continuous Learning: Reality Feedback Agent closes prediction-outcome loop, enabling system improvement through calibration and heuristic extraction.

7.3 Computational Viability

Monthly token budget: 3,000,000 tokens

Expected consumption:

- Regime updates: 750,000 tokens (batch processing)
- User queries: 1,500,000 tokens (6,000 queries at 250 tokens average)
- Learning and calibration: 750,000 tokens (outcome integration, model updates)

Capacity: 100-1,000 active users per month depending on usage intensity.

The architecture demonstrates that sophisticated strategic reasoning is achievable within free-tier constraints through careful architectural design prioritizing computational efficiency.

7.4 Future Extensions

This architecture provides foundation for extensions:

Additional Agents: New specialized agents can be integrated through message bus (e.g., Writing Coach for application materials, Interview Simulator for practice).

Improved Heuristics: As outcome data accumulates, heuristic library grows, reducing Tier 2-3 operations to Tier 0-1.

Model Upgrades: When more efficient language models become available, token budget enables higher-quality reasoning at same cost.

Scaling: Architecture scales horizontally through sharding users across multiple instances, each operating within free-tier limits.

The constraint-driven architecture ensures long-term viability: computational efficiency is not optimization but survival requirement, embedded in fundamental design decisions.