Contents

Cognitive Layer: Core Reasoning Components **2**
Overview 2
1. Goal Analysis Module 2
1.1 Purpose & Responsibilities 2
1.2 Architecture Interna 2
1.3 Semantic Parser 4
1.4 Goal Extractor 5
1.5 Constraint Analyzer 7
1.6 Context Assessor 9
1.7 Complexity Classifier 10
1.8 Goal Analysis Output Schema 12
2. Planning Engine 13
2.1 Purpose & Responsibilities 13
2.2 Architecture Interna 14
2.3 Strategy Selector 15
2.4 Task Decomposer (HTN Focus) 16
2.5 Dependency Analyzer 18
2.6 Resource Estimator 21
2.7 Plan Optimizer 23
2.8 Contingency Planner 25
2.9 Planning Engine Output Schema 28
3. Execution Engine 30
3.1 Purpose & Responsibilities 30
3.2 Architecture Interna 30
3.3 Execution Coordinator 31
3.4 Task Executor 33
3.5 State Manager 35
3.6 Monitoring System 37
3.7 Verification Layer 39
3.8 Adaptation Engine 42
3.9 Execution Engine Output Schema 46
4. Reflection Module 48
4.1 Purpose & Responsibilities 48
4.2 Architecture Interna 48
4.3 Episode Analyzer 49
4.4 Pattern Extractor 51
4.5 Strategy Evaluator 54
4.6 Performance Analyzer 56
4.7 Knowledge Distiller 58
4.8 Memory Updater 60
4.9 Reflection Output Schema 62
5. Cognitive Layer Integration 64
5.1 Complete Flow Example 64
5.2 Caratteristiche del Cognitive Layer 67
5.3 Caratteristiche di Performance 67

Cognitive Layer: Core Reasoning Components

Overview

Il Cognitive Layer è il cuore dell'agente - dove avviene il reasoning di alto livello. Comprende 4 moduli interconnessi che implementano il ciclo completo di problem-solving: comprensione del problema, pianificazione, esecuzione, e apprendimento dall'esperienza.

```
+------------------------------------------------------------------+
|                       COGNITIVE LAYER                            |
|                                                                  |
|  +---------------+      +---------------+      +---------------+ |
|  |     GOAL      |  ->  |   PLANNING    |  ->  |  EXECUTION    | |
|  |   ANALYSIS    |      |    ENGINE     |      |    ENGINE     | |
|  +------+-------+      +------+-------+      +------+-------+ |
|         |                     |                       |         |
|         |                     |                       |         |
|         |      +-------------------------+          |         |
|         +------|    REFLECTION MODULE    |----------+         |
|                |  (Cross-Episode Learning)|                     |
|                +-------------------------+                     |
|                                                                  |
|  Data Flows:                                                    |
|  * Task -> Goal Analysis -> {goals, constraints, context}        |
|  * Goals -> Planning -> {plan, subtasks, dependencies}           |
|  * Plan -> Execution -> {actions, results, state changes}        |
|  * Episode -> Reflection -> {insights, patterns, strategies}     |
+------------------------------------------------------------------+
```

1. Goal Analysis Module

1.1 Purpose & Responsibilities

Core Function: Trasformare task in input (spesso ambiguo, natural language) in rappresentazione strutturata e computazionale.

Responsibilities: 1. **Parsing**: Estrarre semantica da input naturale 2. **Goal Extraction**: Identificare obiettivi primari e secondari 3. **Constraint Identification**: Rilevare vincoli impliciti ed espliciti 4. **Context Assessment**: Valutare informazioni contestuali rilevanti 5. **Complexity Classification**: Determinare classe problema e strategia appropriata

1.2 Architecture Interna

```
+-----------------------------------------------------------------+
|                     GOAL ANALYSIS MODULE                        |
|                                                                 |
|  Input: Natural Language Task + Context                        |
```

```
|    |                                                            |
|  +-----------------------------------------+                   |
|  |  1. SEMANTIC PARSER                     |                   |
|  |  * Intent classification                |                   |
|  |  * Entity extraction                    |                   |
|  |  * Relation identification              |                   |
|  +-------------+---------------------------+                   |
|                |                                                |
|  +-----------------------------------------+                   |
|  |  2. GOAL EXTRACTOR                      |                   |
|  |  * Primary goal identification          |                   |
|  |  * Sub-goal decomposition               |                   |
|  |  * Success criteria extraction          |                   |
|  +-------------+---------------------------+                   |
|                |                                                |
|  +-----------------------------------------+                   |
|  |  3. CONSTRAINT ANALYZER                 |                   |
|  |  * Explicit constraints                 |                   |
|  |  * Implicit constraints (from context)  |                   |
|  |  * Safety bounds                        |                   |
|  |  * Resource limits                      |                   |
|  +-------------+---------------------------+                   |
|                |                                                |
|  +-----------------------------------------+                   |
|  |  4. CONTEXT ASSESSOR                    |                   |
|  |  * Retrieve relevant history            |                   |
|  |  * Identify domain/environment          |                   |
|  |  * Assess available resources           |                   |
|  +-------------+---------------------------+                   |
|                |                                                |
|  +-----------------------------------------+                   |
|  |  5. COMPLEXITY CLASSIFIER               |                   |
|  |  * Estimate problem difficulty          |                   |
|  |  * Classify into taxonomy category      |                   |
|  |  * Select appropriate strategy          |                   |
|  +-------------+---------------------------+                   |
|                |                                                |
|  Output: Structured Goal Representation                        |
|  {                                                              |
|    primary_goal: {...},                                        |
|    sub_goals: [...],                                           |
|    constraints: {...},                                         |
|    context: {...},                                             |
|    complexity: "...",                                          |
|    strategy_hint: "..."                                        |
|  }                                                              |
+-----------------------------------------------------------------+
```

1.3 Semantic Parser

Approach: Hybrid (LLM reasoning + structured extraction)

Architecture:

```
Input Text
    |
+----------------------------------+
|  Intent Classification           |
|  +----------------------------+  |
|  | Questioning vs Requesting  |  |
|  | Analysis vs Action         |  |
|  | Creation vs Modification   |  |
|  | Research vs Implementation |  |
|  +----------------------------+  |
+--------------+-------------------+
               |
+----------------------------------+
|  Entity Extraction               |
|  +----------------------------+  |
|  | Objects (files, services)  |  |
|  | Actions (verbs)            |  |
|  | Attributes (modifiers)     |  |
|  | Quantities (numbers, time) |  |
|  +----------------------------+  |
+--------------+-------------------+
               |
+----------------------------------+
|  Relation Identification         |
|  +----------------------------+  |
|  | Dependencies               |  |
|  | Sequencing                 |  |
|  | Constraints                |  |
|  | Hierarchies                |  |
|  +----------------------------+  |
+--------------+-------------------+
               |
    Semantic Structure
```

Example Flow:

```
Input: "Refactor the authentication module to use JWT tokens,
        ensure backward compatibility, and update tests"

Semantic Parser Output:
{
  intent: "MODIFICATION",
  entities: {
    objects: ["authentication module", "JWT tokens", "tests"],
```

```
    actions: ["refactor", "ensure", "update"],
    constraints: ["backward compatibility"]
  },
  relations: {
    primary_action: "refactor authentication module",
    method: "use JWT tokens",
    constraint: "maintain backward compatibility",
    follow_up: "update tests"
  }
}
```

1.4 Goal Extractor

Purpose: Convertire semantic structure in goal hierarchy esplicita.

Goal Representation:

```
Goal {
  id: unique_identifier,
  type: PRIMARY | SECONDARY | CONSTRAINT,
  description: human_readable,
  success_criteria: [measurable_conditions],
  priority: 1-10,
  dependencies: [goal_ids],
  estimated_complexity: LOW | MEDIUM | HIGH,
  verification_method: how_to_verify_success
}
```

Extraction Process:

```
Semantic Structure
    |
+---------------------------------------+
|  Primary Goal Identification          |
|  * Main intent -> Primary goal         |
|  * Core action -> Goal description     |
|  * Expected outcome -> Success         |
+--------------+------------------------+
               |
+---------------------------------------+
|  Sub-Goal Decomposition               |
|  * Identify preparatory steps         |
|  * Extract follow-up actions          |
|  * Recognize parallel tasks           |
+--------------+------------------------+
               |
+---------------------------------------+
|  Success Criteria Extraction          |
|  * Explicit conditions from input     |
```

```
|  * Implicit quality standards        |
|  * Domain-specific requirements      |
+--------------+----------------------+
               |
+-------------------------------------+
|  Dependency Analysis                 |
|  * Which goals must precede others   |
|  * Which can run in parallel         |
|  * Critical path identification      |
+--------------+----------------------+
               |
    Goal Hierarchy
```

Example Flow:

```
Input (from parser):
{
  intent: "MODIFICATION",
  entities: {
    objects: ["authentication module", "JWT tokens", "tests"],
    actions: ["refactor", "ensure", "update"],
    constraints: ["backward compatibility"]
  }
}

Goal Extractor Output:
{
  primary_goal: {
    id: "G1",
    type: "PRIMARY",
    description: "Refactor authentication to use JWT",
    success_criteria: [
      "Authentication uses JWT tokens",
      "All auth flows functional",
      "Tests passing"
    ],
    priority: 10,
    dependencies: [],
    estimated_complexity: "HIGH"
  },
  sub_goals: [
    {
      id: "G1.1",
      type: "SECONDARY",
      description: "Implement JWT token generation",
      success_criteria: ["JWT library integrated", "Token generation working"],
      priority: 9,
      dependencies: [],
      estimated_complexity: "MEDIUM"
```

```
    },
    {
      id: "G1.2",
      type: "SECONDARY",
      description: "Update authentication flow",
      success_criteria: ["Auth flow uses JWT", "Old flow still works"],
      priority: 8,
      dependencies: ["G1.1"],
      estimated_complexity: "HIGH"
    },
    {
      id: "G1.3",
      type: "SECONDARY",
      description: "Update test suite",
      success_criteria: ["All tests pass", "New tests for JWT"],
      priority: 7,
      dependencies: ["G1.2"],
      estimated_complexity: "MEDIUM"
    }
  ],
  constraints: [
    {
      id: "C1",
      type: "CONSTRAINT",
      description: "Maintain backward compatibility",
      verification_method: "Old clients still authenticate",
      priority: 10
    }
  ]
}
```

1.5 Constraint Analyzer

Purpose: Identificare tutti i vincoli che limitano spazio di soluzioni accettabili.

Constraint Types:

```
+--------------------------------------------------------------+
|                      CONSTRAINT TAXONOMY                     |
|                                                              |
|  1. FUNCTIONAL CONSTRAINTS                                   |
|     * Must-have features                                     |
|     * Compatibility requirements                             |
|     * Interface contracts                                    |
|                                                              |
|  2. NON-FUNCTIONAL CONSTRAINTS                               |
|     * Performance (latency, throughput)                      |
|     * Scalability requirements                               |
|     * Security requirements                                  |
```

```
|                                                              |
|  3. RESOURCE CONSTRAINTS                                     |
|     * Time budget                                            |
|     * Cost budget                                            |
|     * Token/API call limits                                  |
|                                                              |
|  4. SAFETY CONSTRAINTS                                       |
|     * Actions prohibited                                     |
|     * Data protection requirements                           |
|     * Approval gates needed                                  |
|                                                              |
|  5. CONTEXTUAL CONSTRAINTS                                   |
|     * Environment limitations                                |
|     * Available tools/capabilities                           |
|     * Dependencies on external systems                       |
+--------------------------------------------------------------+
```

Analysis Flow:

```
Input: Task + Context + History
    |
+--------------------------------------+
|  Explicit Constraint Extraction      |
|  * Parse stated requirements         |
|  * Identify "must", "never", etc.    |
+--------------+-----------------------+
               |
+--------------------------------------+
|  Implicit Constraint Inference       |
|  * Domain conventions                |
|  * Safety standards                  |
|  * Quality expectations              |
+--------------+-----------------------+
               |
+--------------------------------------+
|  Constraint Validation               |
|  * Check for conflicts               |
|  * Assess feasibility                |
|  * Prioritize if conflicting         |
+--------------+-----------------------+
               |
+--------------------------------------+
|  Bound Generation                    |
|  * Convert to enforceable bounds     |
|  * Define verification methods       |
+--------------+-----------------------+
               |
    Constraint Set
```

1.6 Context Assessor

Purpose: Raccogliere e valutare informazioni contestuali rilevanti per il task.

Context Dimensions:

```
+-----------------------------------------------------------+
|                  CONTEXT ASSESSMENT                       |
|                                                           |
|  +-----------------------------------------------------+  |
|  |  HISTORICAL CONTEXT                                 |  |
|  |  * Similar past tasks                               |  |
|  |  * Previous failures/successes                      |  |
|  |  * Learned patterns                                 |  |
|  |  * User preferences                                 |  |
|  +-----------------------------------------------------+  |
|                          |                                |
|  +-----------------------------------------------------+  |
|  |  ENVIRONMENTAL CONTEXT                              |  |
|  |  * Current working directory                        |  |
|  |  * Available files/resources                        |  |
|  |  * System state                                     |  |
|  |  * External service status                          |  |
|  +-----------------------------------------------------+  |
|                          |                                |
|  +-----------------------------------------------------+  |
|  |  CAPABILITY CONTEXT                                 |  |
|  |  * Available tools                                  |  |
|  |  * API access                                       |  |
|  |  * Model capabilities                               |  |
|  |  * Resource budgets                                 |  |
|  +-----------------------------------------------------+  |
|                          |                                |
|  +-----------------------------------------------------+  |
|  |  DOMAIN CONTEXT                                     |  |
|  |  * Technical domain (web, ML, data)                 |  |
|  |  * Language/framework                               |  |
|  |  * Architecture patterns                            |  |
|  |  * Best practices                                   |  |
|  +-----------------------------------------------------+  |
+-----------------------------------------------------------+
```

Assessment Process:

```
Task Input
    |
+----------------------------------+
|  Query Historical Memory         |
|  * Semantic search similar tasks|
|  * Retrieve outcomes             |
```

```
|  * Extract relevant patterns    |
+--------------+-----------------+
               |
+--------------------------------+
|  Assess Environment            |
|  * Probe file system           |
|  * Check service availability  |
|  * Verify resource access      |
+--------------+-----------------+
               |
+--------------------------------+
|  Inventory Capabilities        |
|  * Available tools registry    |
|  * Model routing options       |
|  * Budget remaining            |
+--------------+-----------------+
               |
+--------------------------------+
|  Identify Domain               |
|  * Classify technical area     |
|  * Load domain knowledge       |
|  * Apply conventions           |
+--------------+-----------------+
               |
    Context Structure
```

1.7 Complexity Classifier

Purpose: Stimare difficoltà problema e selezionare strategia appropriata.

Classification Dimensions:

```
+---------------------------------------------------------------+
|                  COMPLEXITY ASSESSMENT                        |
|                                                               |
|  Dimension 1: DECOMPOSITION DEPTH                             |
|  +- Simple (1 step)                                           |
|  +- Moderate (2-5 steps)                                      |
|  +- Complex (6-15 steps)                                      |
|  +- Very Complex (>15 steps)                                  |
|                                                               |
|  Dimension 2: UNCERTAINTY                                     |
|  +- Deterministic (clear path)                                |
|  +- Low uncertainty (few unknowns)                            |
|  +- Moderate uncertainty (some exploration needed)            |
|  +- High uncertainty (research required)                      |
|                                                               |
|  Dimension 3: INTERDEPENDENCIES                               |
|  +- Independent (no dependencies)                             |
```

```
|  +- Sequential (linear chain)                                |
|  +- Tree (branching dependencies)                            |
|  +- Graph (complex interdependencies)                        |
|                                                              |
|  Dimension 4: RISK                                           |
|  +- No-risk (easily reversible)                              |
|  +- Low-risk (can rollback)                                  |
|  +- Medium-risk (some irreversible)                          |
|  +- High-risk (critical operations)                          |
|                                                              |
|  Dimension 5: NOVELTY                                        |
|  +- Routine (done many times)                                |
|  +- Familiar (similar to past tasks)                         |
|  +- Novel (first time, but similar domain)                   |
|  +- Unprecedented (no prior examples)                        |
+--------------------------------------------------------------+
```

Classification Decision Tree:

```
Task
  |
Is it single-step?
  +- YES -> SIMPLE
  |         +- Routing: Direct execution
  |         +- Strategy: Single LLM call
  |
  +- NO -> Multi-step needed
            |
            Uncertainty level?
            +- LOW -> STRUCTURED_COMPLEX
            |         +- Routing: Planning Engine
            |         +- Strategy: HTN planning
            |
            +- HIGH -> EXPLORATORY_COMPLEX
                      |
                      Risk level?
                      +- LOW/MEDIUM -> ADAPTIVE
                      |                +- Routing: Planning + Reflection
                      |                +- Strategy: Iterative refinement
                      |
                      +- HIGH -> VERIFIED
                                 +- Routing: Planning + Verification
                                 +- Strategy: Formal verification
```

Output Mapping:

```
Complexity Classification -> Strategy Selection

SIMPLE:
  - No planning phase needed
```

```
  - Direct execution with single tool call
  - Minimal memory usage
  - Fast path

STRUCTURED_COMPLEX:
  - Hierarchical task decomposition
  - Clear dependency management
  - Systematic execution
  - Standard planning approach

EXPLORATORY_COMPLEX:
  - Iterative planning
  - Frequent replanning based on discoveries
  - Heavy memory usage (episodic)
  - Reflection after each major step

VERIFIED_COMPLEX:
  - Extensive pre-execution planning
  - Verification gates before critical actions
  - Human approval for high-risk operations
  - Comprehensive logging and traceability
```

1.8 Goal Analysis Output Schema

Complete Output Structure:

```
GoalAnalysisResult {
  // Core Task Understanding
  task_id: string,
  original_input: string,
  timestamp: datetime,

  // Semantic Understanding
  semantic: {
    intent: Intent,
    entities: Entity[],
    relations: Relation[]
  },

  // Goal Structure
  goals: {
    primary: Goal,
    secondary: Goal[],
    constraints: Constraint[]
  },

  // Context
  context: {
```

```
    historical: {
      similar_tasks: Episode[],
      relevant_patterns: Pattern[],
      user_preferences: Preference[]
    },
    environmental: {
      working_directory: string,
      available_resources: Resource[],
      system_state: State
    },
    capabilities: {
      available_tools: Tool[],
      model_options: Model[],
      budget: Budget
    },
    domain: {
      identified_domain: Domain,
      conventions: Convention[],
      best_practices: Practice[]
    }
  },

  // Complexity Assessment
  complexity: {
    overall: ComplexityLevel,
    dimensions: {
      decomposition_depth: int,
      uncertainty: UncertaintyLevel,
      interdependencies: DependencyStructure,
      risk: RiskLevel,
      novelty: NoveltyLevel
    },
    recommended_strategy: Strategy
  },

  // Metadata
  confidence: float,
  ambiguities: Ambiguity[],
  assumptions: Assumption[]
}
```

2. Planning Engine

2.1 Purpose & Responsibilities

Core Function: Generare execution plan ottimale che trasforma goal structure in sequenza di azioni concrete ed eseguibili.

Responsibilities: 1. **Task Decomposition**: Scomporre goal complessi in subtask atomici 2. **Dependency Management**: Identificare e gestire dipendenze tra subtask 3. **Resource Allocation**: Stimare e allocare risorse (tempo, token, tools) 4. **Strategy Selection**: Scegliere approccio di planning appropriato 5. **Plan Optimization**: Ottimizzare per latenza, costo, o altri obiettivi 6. **Contingency Planning**: Preparare fallback per potenziali failure

2.2 Architecture Interna

```
+-------------------------------------------------------------------+
|                       PLANNING ENGINE                             |
|                                                                   |
|  Input: GoalAnalysisResult                                        |
|    |                                                              |
|  +---------------------------------------------+                 |
|  |  1. STRATEGY SELECTOR                       |                 |
|  |  * Map complexity -> planning approach       |                |
|  |  * Select decomposition method              |                 |
|  +----------------+----------------------------+                 |
|                   |                                               |
|  +---------------------------------------------+                 |
|  |  2. TASK DECOMPOSER                         |                 |
|  |  * Hierarchical decomposition               |                 |
|  |  * Atomic action identification             |                 |
|  |  * Subtask parameterization                 |                 |
|  +----------------+----------------------------+                 |
|                   |                                               |
|  +---------------------------------------------+                 |
|  |  3. DEPENDENCY ANALYZER                     |                 |
|  |  * Build dependency graph                   |                 |
|  |  * Identify critical path                   |                 |
|  |  * Find parallelizable subtasks             |                 |
|  +----------------+----------------------------+                 |
|                   |                                               |
|  +---------------------------------------------+                 |
|  |  4. RESOURCE ESTIMATOR                      |                 |
|  |  * Estimate time per subtask                |                 |
|  |  * Estimate cost (tokens, API calls)        |                 |
|  |  * Check against budgets                    |                 |
|  +----------------+----------------------------+                 |
|                   |                                               |
|  +---------------------------------------------+                 |
|  |  5. PLAN OPTIMIZER                          |                 |
|  |  * Sequence optimization                    |                 |
|  |  * Parallel execution opportunities         |                 |
|  |  * Risk minimization                        |                 |
|  +----------------+----------------------------+                 |
|                   |                                               |
```

```
|  +-----------------------------------------------+                  |
|  |  6. CONTINGENCY PLANNER                        |                  |
|  |  * Identify failure points                     |                  |
|  |  * Generate fallback strategies                |                  |
|  |  * Define recovery procedures                  |                  |
|  +-----------------+-----------------------------+                   |
|                    |                                                  |
|  Output: ExecutionPlan                                                |
|  {                                                                    |
|    subtasks: [...],                                                   |
|    dependencies: DAG,                                                 |
|    execution_order: [...],                                            |
|    resource_estimates: {...},                                         |
|    contingencies: [...]                                               |
|  }                                                                    |
+--------------------------------------------------------------------+
```

2.3 Strategy Selector

Purpose: Selezionare metodo di planning appropriato basato su caratteristiche task.

Available Strategies:

```
+-----------------------------------------------------------------+
|                     PLANNING STRATEGIES                         |
|                                                                 |
|  1. DIRECT EXECUTION                                            |
|     When: Simple, single-step tasks                             |
|     Approach: No decomposition, direct tool invocation          |
|     Overhead: Minimal                                           |
|                                                                 |
|  2. LINEAR DECOMPOSITION                                        |
|     When: Sequential tasks, clear path                          |
|     Approach: Simple step-by-step breakdown                     |
|     Overhead: Low                                               |
|                                                                 |
|  3. HIERARCHICAL TASK NETWORK (HTN)                             |
|     When: Complex tasks, known domain                           |
|     Approach: Recursive decomposition with methods              |
|     Overhead: Medium                                            |
|                                                                 |
|  4. MEANS-ENDS ANALYSIS                                         |
|     When: Goal-oriented, reduce difference iteratively          |
|     Approach: Identify gaps, find operators to reduce           |
|     Overhead: Medium                                            |
|                                                                 |
|  5. CASE-BASED PLANNING                                         |
|     When: Similar past tasks available                          |
|     Approach: Retrieve, adapt previous plans                    |
```

```
|     Overhead: Low (if cache hit)                                |
|                                                                 |
|  6. EXPLORATORY PLANNING                                        |
|     When: High uncertainty, research needed                     |
|     Approach: Iterative: plan -> execute -> learn -> replan        |
|     Overhead: High                                              |
|                                                                 |
|  7. VERIFIED PLANNING                                           |
|     When: Safety-critical, high-risk operations                 |
|     Approach: Formal specification, verification gates          |
|     Overhead: Very High                                         |
+-----------------------------------------------------------------+
```

Selection Decision Tree:

```
Complexity Assessment
  |
Is task atomic (1 step)?
  +- YES -> DIRECT EXECUTION
  |
  +- NO -> Is path clear & sequential?
            +- YES -> LINEAR DECOMPOSITION
            |
            +- NO -> Is domain well-known?
                      +- YES -> Do we have similar past tasks?
                      |         +- YES -> CASE-BASED PLANNING
                      |         |
                      |         +- NO -> HTN PLANNING
                      |
                      +- NO -> Is it high-risk?
                                +- YES -> VERIFIED PLANNING
                                |
                                +- NO -> EXPLORATORY PLANNING
```

2.4 Task Decomposer (HTN Focus)

HTN Planning Approach: Inspired by classical HTN but adapted for LLM context.

Core Concepts:

```
+-------------------------------------------------------------+
|               HIERARCHICAL TASK NETWORK (HTN)               |
|                                                             |
|  Compound Task: High-level goal that needs decomposition    |
|  Primitive Task: Atomic action directly executable          |
|  Method: Way to decompose compound task into subtasks       |
|  Ordering: Constraints on subtask execution sequence        |
+-------------------------------------------------------------+
```

Decomposition Process:

```
Goal: "Refactor authentication to use JWT"
  |
+----------------------------------------------+
|  Level 0: Goal (Compound Task)               |
|  * Refactor authentication to JWT            |
+-----------------+----------------------------+
                  |
+----------------------------------------------+
|  Level 1: Major Phases (Compound Tasks)      |
|  1. Research & Design                        |
|  2. Implementation                           |
|  3. Testing & Validation                     |
|  4. Deployment                               |
+-----------------+----------------------------+
                  |
+----------------------------------------------+
|  Level 2: Concrete Steps (Mix)               |
|  1.1 Research JWT libraries (Compound)       |
|  1.2 Design token flow (Compound)            |
|  2.1 Install JWT library (Primitive)         |
|  2.2 Implement token generation (Compound)   |
|  2.3 Update auth middleware (Compound)       |
|  3.1 Write unit tests (Compound)             |
|  3.2 Run test suite (Primitive)              |
|  4.1 Deploy to staging (Primitive)           |
|  4.2 Verify functionality (Compound)         |
+-----------------+----------------------------+
                  |
+----------------------------------------------+
|  Level 3: Atomic Actions (Primitives)        |
|  1.1.1 Search for "jwt library python"       |
|  1.1.2 Read library documentation            |
|  1.1.3 Compare options                       |
|  2.2.1 Create jwt_utils.py file              |
|  2.2.2 Import jwt library                    |
|  2.2.3 Write encode_token function           |
|  2.2.4 Write decode_token function           |
|  ... (continues)                             |
+----------------------------------------------+
```

Decomposition Algorithm (Conceptual):

```
Function DECOMPOSE(task, depth):
  IF task is primitive:
    RETURN [task]

  IF depth > MAX_DEPTH:
    TREAT as primitive (avoid infinite recursion)
    RETURN [task]
```

```
  # Query LLM for decomposition
  methods = LLM.generate_methods(task, context)

  # Select best method
  method = SELECT_BEST(methods, criteria)

  # Apply method to get subtasks
  subtasks = APPLY_METHOD(method, task)

  # Recursively decompose compound subtasks
  plan = []
  FOR subtask IN subtasks:
    plan.extend(DECOMPOSE(subtask, depth + 1))

  RETURN plan
```

Method Selection Criteria:

```
+------------------------------------------+
|  METHOD SELECTION FACTORS                |
|                                          |
|  1. FEASIBILITY                          |
|     * Are required resources available? |
|     * Are dependencies satisfiable?      |
|                                          |
|  2. EFFICIENCY                           |
|     * Estimated time                     |
|     * Estimated cost                     |
|     * Complexity overhead                |
|                                          |
|  3. RISK                                 |
|     * Probability of failure             |
|     * Impact of failure                  |
|     * Reversibility                      |
|                                          |
|  4. QUALITY                              |
|     * Expected outcome quality           |
|     * Maintainability                    |
|     * Robustness                         |
|                                          |
|  5. PAST PERFORMANCE                     |
|     * Success rate of similar methods    |
|     * Learned preferences                |
+------------------------------------------+
```

2.5 Dependency Analyzer

Purpose: Costruire grafo delle dipendenze e identificare vincoli di ordinamento.

Dependency Types:

```
+-----------------------------------------------------------+
|                  DEPENDENCY TYPES                         |
|                                                           |
|  1. PREREQUISITE (Must-Before)                            |
|     Task B requires output of Task A                      |
|     Example: "Install library" before "Import library"    |
|                                                           |
|  2. ORDERING (Should-Before)                              |
|     Task B logically follows A (but not strict)           |
|     Example: "Design" before "Implement"                  |
|                                                           |
|  3. RESOURCE (Shared-Resource)                            |
|     Tasks compete for same resource, can't run parallel   |
|     Example: Two tasks modifying same file                |
|                                                           |
|  4. CONFLICT (Mutual-Exclusion)                           |
|     Tasks logically conflict, must not overlap            |
|     Example: "Backup DB" and "Modify DB"                  |
+-----------------------------------------------------------+
```

Dependency Graph Construction:

```
Subtasks: [T1, T2, T3, T4, T5]
  |
+---------------------------------------+
|  Analyze Each Pair (Ti, Tj)           |
|  * Does Ti produce input for Tj?      |
|  * Do they access same resources?     |
|  * Are they logically ordered?        |
|  * Do they conflict?                  |
+--------------+------------------------+
               |
+---------------------------------------+
|  Build Dependency Graph (DAG)         |
|                                       |
|         T1                            |
|        / \                           |
|       /   \                          |
|      T2   T3                          |
|       \   /                          |
|        \ /                           |
|         T4                            |
|         |                             |
|         T5                            |
|                                       |
|  Edges represent dependencies         |
+--------------+------------------------+
```

```
              |
+-------------------------------------+
|  Identify Parallelizable Tasks       |
|  * T2 and T3 can run in parallel     |
|    (no dependencies between them)    |
+-------------+-----------------------+
              |
+-------------------------------------+
|  Compute Critical Path               |
|  * Longest path through graph        |
|  * Determines minimum execution time |
|  * Example: T1 -> T2 -> T4 -> T5        |
+-------------------------------------+
```

Topological Sort for Execution Order:

```
Algorithm: Kahn's Algorithm (adapted)

Input: Dependency DAG
Output: Valid execution order

1. Compute in-degree for each task
   (number of dependencies)

2. Queue all tasks with in-degree 0
   (no dependencies)

3. While queue not empty:
   a. Dequeue task T
   b. Add T to execution order
   c. For each task T' dependent on T:
      - Decrease in-degree of T'
      - If in-degree becomes 0, enqueue T'

4. If any tasks remain (not in order):
   -> Cycle detected! Error.

5. Return execution order
```

Example:

```
Tasks: [Install_JWT, Import_JWT, Create_Encoder, Create_Decoder,
        Update_Auth, Write_Tests, Run_Tests]

Dependencies:
  Import_JWT -> Install_JWT
  Create_Encoder -> Import_JWT
  Create_Decoder -> Import_JWT
  Update_Auth -> Create_Encoder, Create_Decoder
  Write_Tests -> Update_Auth
```

```
  Run_Tests -> Write_Tests

Execution Order (one valid ordering):
  1. Install_JWT
  2. Import_JWT
  3. Create_Encoder    } Can run in parallel
  4. Create_Decoder    }
  5. Update_Auth
  6. Write_Tests
  7. Run_Tests

Critical Path:
  Install_JWT -> Import_JWT -> Create_Encoder -> Update_Auth
  -> Write_Tests -> Run_Tests
```

2.6 Resource Estimator

Purpose: Stimare risorse necessarie per ogni subtask e piano complessivo.

Resource Types:

```
+----------------------------------------------------------+
|                  RESOURCE DIMENSIONS                     |
|                                                          |
|  1. TIME                                                 |
|     * Wall-clock time (latency)                          |
|     * LLM inference time                                 |
|     * Tool execution time                                |
|                                                          |
|  2. COMPUTE                                              |
|     * LLM calls (count, size)                            |
|     * Token usage (input + output)                       |
|     * Model tier required (small/large)                  |
|                                                          |
|  3. COST                                                 |
|     * LLM API cost                                       |
|     * External API cost                                  |
|     * Compute infrastructure cost                        |
|                                                          |
|  4. MEMORY                                               |
|     * Context window usage                               |
|     * Working memory size                                |
|     * Storage for intermediate results                   |
|                                                          |
|  5. EXTERNAL DEPENDENCIES                                |
|     * API rate limits                                    |
|     * Service availability                               |
|     * Network bandwidth                                  |
+----------------------------------------------------------+
```

Estimation Process:

```
For each subtask:
  |
+--------------------------------------+
|  Classify Task Type                  |
|  * LLM reasoning                     |
|  * Tool execution                    |
|  * External API call                 |
|  * File I/O                          |
+--------------+-----------------------+
               |
+--------------------------------------+
|  Look up Historical Data             |
|  * Query similar tasks from memory   |
|  * Get actual resource usage         |
|  * Compute statistics (p50, p95)     |
+--------------+-----------------------+
               |
+--------------------------------------+
|  Apply Heuristics                    |
|  * Task complexity multiplier        |
|  * Context size factor               |
|  * Uncertainty buffer                |
+--------------+-----------------------+
               |
+--------------------------------------+
|  Generate Estimate                   |
|  * Point estimate                    |
|  * Confidence interval               |
|  * Risk factors                      |
+--------------+-----------------------+
               |
    Per-Task Resource Estimate
```

Aggregation for Overall Plan:

```
Individual Estimates -> Overall Plan Estimate

Sequential Tasks:
  Time = sum(task_times)
  Cost = sum(task_costs)
  Tokens = sum(task_tokens)

Parallel Tasks:
  Time = max(parallel_group_times)
  Cost = sum(parallel_group_costs)
  Tokens = sum(parallel_group_tokens)
```

```
With Uncertainty:
  Time_estimate = expected_time * (1 + uncertainty_factor)
  Cost_budget = expected_cost * (1 + buffer_factor)
```

Example Estimation:

```
Task: "Refactor authentication to JWT"

Subtask Estimates:
+-----------------------------------------------------------+
| Subtask              Time    LLM Calls  Tokens   Cost    |
+-----------------------------------------------------------+
| Research JWT libs    30s     2          8K       $0.02   |
| Design token flow    45s     3          12K      $0.03   |
| Install library      10s     1          2K       $0.01   |
| Create encoder       60s     4          15K      $0.04   |
| Create decoder       60s     4          15K      $0.04   |
| Update auth          90s     5          20K      $0.05   |
| Write tests          120s    6          25K      $0.06   |
| Run tests            15s     0          0        $0.00   |
+-----------------------------------------------------------+
| TOTAL (sequential)   430s    25         97K      $0.25   |
| TOTAL (optimized)    ~300s   25         97K      $0.25   |
|   (encoder+decoder in parallel)                          |
+-----------------------------------------------------------+

Budget Check:
  [v] Time: 300s < 600s budget
  [v] Cost: $0.25 < $0.50 budget
  [v] Tokens: 97K < 200K budget
  -> Plan is feasible
```

2.7 Plan Optimizer

Purpose: Ottimizzare piano di esecuzione per obiettivi specificati.

Optimization Objectives:

```
+-------------------------------------------------------+
|              OPTIMIZATION OBJECTIVES                  |
|                                                       |
|  1. MINIMIZE LATENCY                                  |
|     * Maximize parallelization                        |
|     * Use faster models for non-critical tasks        |
|     * Cache aggressively                              |
|                                                       |
|  2. MINIMIZE COST                                     |
|     * Use smaller models when possible                |
|     * Batch operations                                |
```

```
|     * Reuse computations                                    |
|                                                             |
|  3. MAXIMIZE QUALITY                                        |
|     * Use best models                                       |
|     * Add verification steps                                |
|     * Multi-pass refinement                                 |
|                                                             |
|  4. MINIMIZE RISK                                           |
|     * Add checkpoints                                       |
|     * Prefer reversible operations early                    |
|     * Add approval gates                                    |
|                                                             |
|  5. BALANCE (Default)                                       |
|     * Trade-off between above                               |
|     * Context-appropriate weighting                         |
+-------------------------------------------------------------+
```

Optimization Techniques:

```
1. PARALLELIZATION
   Before:  T1 -> T2 -> T3 -> T4 (sequential)
   After:   T1 -> {T2, T3} -> T4 (parallel)
            +-----+------+
   Speedup: 2x for T2+T3 phase

2. REORDERING
   Before:  Expensive -> Cheap -> Verify
   After:   Cheap -> Verify -> Expensive
   Benefit: Fail fast, save expensive work

3. BATCHING
   Before:  API_call(x1), API_call(x2), API_call(x3)
   After:   API_batch_call([x1, x2, x3])
   Benefit: Reduce overhead, cost

4. CACHING
   Before:  Compute F(x) multiple times
   After:   Compute once, reuse result
   Benefit: Eliminate redundant work

5. MODEL ROUTING
   Before:  Use large model for all tasks
   After:   Use small model for simple, large for complex
   Benefit: Reduce cost without quality loss

6. INCREMENTAL EXECUTION
   Before:  Build entire solution, then test
   After:   Build incrementally, test each piece
   Benefit: Earlier feedback, easier debugging
```

Optimization Algorithm (Conceptual):

```
Function OPTIMIZE_PLAN(plan, objective):

  # 1. Identify parallelization opportunities
  parallel_groups = FIND_PARALLEL_TASKS(plan.dag)
  plan = APPLY_PARALLELIZATION(plan, parallel_groups)

  # 2. Reorder for fail-fast
  IF objective prioritizes risk_minimization:
    plan = REORDER_FOR_EARLY_VALIDATION(plan)

  # 3. Apply caching
  cacheable = IDENTIFY_CACHEABLE_COMPUTATIONS(plan)
  plan = INSERT_CACHE_LOOKUPS(plan, cacheable)

  # 4. Model routing optimization
  FOR task IN plan.tasks:
    IF task is simple AND objective prioritizes cost:
      task.model = SMALL_MODEL
    ELSE IF task is critical AND objective prioritizes quality:
      task.model = LARGE_MODEL
    ELSE:
      task.model = MEDIUM_MODEL

  # 5. Batch operations
  batchable = FIND_BATCHABLE_OPERATIONS(plan)
  plan = APPLY_BATCHING(plan, batchable)

  RETURN plan
```

2.8 Contingency Planner

Purpose: Preparare strategie di fallback per gestire failure previsti.

Failure Taxonomy:

```
+---------------------------------------------------------+
|                    FAILURE TYPES                        |
|                                                         |
|  1. TRANSIENT FAILURES                                  |
|     * Network timeout                                   |
|     * Rate limit hit                                    |
|     * Temporary service unavailable                     |
|     Recovery: Retry with backoff                        |
|                                                         |
|  2. RESOURCE FAILURES                                   |
|     * Budget exhausted                                  |
|     * Timeout exceeded                                  |
```

```
|     * Memory limit reached                             |
|     Recovery: Abort gracefully, report                 |
|                                                        |
|  3. LOGIC FAILURES                                     |
|     * Task prerequisites not met                       |
|     * Invalid input to tool                            |
|     * Assertion violation                              |
|     Recovery: Replan, adjust approach                  |
|                                                        |
|  4. EXTERNAL FAILURES                                  |
|     * Tool unavailable                                 |
|     * API breaking change                              |
|     * File not found                                   |
|     Recovery: Use alternative tool/approach            |
|                                                        |
|  5. QUALITY FAILURES                                   |
|     * Output doesn't meet criteria                     |
|     * Test failures                                    |
|     * Validation rejection                             |
|     Recovery: Refine, iterate, or escalate             |
+--------------------------------------------------------+
```

Contingency Planning Process:

```
For each subtask in plan:
  |
+---------------------------------------+
|  Identify Potential Failures          |
|  * What can go wrong?                 |
|  * Historical failure patterns        |
|  * Uncertainty indicators             |
+--------------+------------------------+
               |
+---------------------------------------+
|  Assess Failure Impact                |
|  * Blocks other tasks?                |
|  * Partial vs complete failure        |
|  * Reversibility of side effects      |
+--------------+------------------------+
               |
+---------------------------------------+
|  Generate Recovery Strategies         |
|  * Retry with modifications           |
|  * Alternative approaches             |
|  * Fallback to simpler solution       |
|  * Human escalation                   |
+--------------+------------------------+
               |
+---------------------------------------+
```

```
|  Attach to Plan                      |
|  * Each task has contingency list    |
|  * Ordered by preference             |
|  * Conditions for triggering         |
+--------------------------------------+
```

Contingency Structure:

```
Contingency {
  failure_type: FailureType,
  detection: {
    symptoms: [observable_conditions],
    confidence_threshold: float
  },
  recovery_strategies: [
    {
      strategy_id: string,
      description: string,
      preconditions: [conditions_to_apply],
      actions: [recovery_steps],
      expected_success_rate: float,
      cost: ResourceEstimate
    }
  ],
  escalation: {
    max_retries: int,
    escalate_to: "human" | "alternative_plan",
    escalation_message: string
  }
}
```

Example Contingencies:

```
Task: "Install JWT library"

Contingency 1:
  Failure: "Package not found in pip"
  Recovery Strategies:
    1. Try alternative package name (e.g., PyJWT vs python-jwt)
    2. Search PyPI for similar packages
    3. Install from source (GitHub)
    4. Ask user for guidance

Contingency 2:
  Failure: "Permission denied"
  Recovery Strategies:
    1. Retry with sudo (if appropriate context)
    2. Install in virtual environment
    3. Install with --user flag
    4. Request user to fix permissions
```

```
Contingency 3:
  Failure: "Network timeout"
  Recovery Strategies:
    1. Retry with exponential backoff (3 attempts)
    2. Try alternative package index
    3. Defer to later (maybe network recovers)
    4. Report failure, request manual installation
```

2.9 Planning Engine Output Schema

```
ExecutionPlan {
  // Metadata
  plan_id: string,
  created_at: datetime,
  goal_analysis_id: string,
  strategy_used: PlanningStrategy,

  // Task Structure
  tasks: [
    {
      task_id: string,
      description: string,
      type: "primitive" | "compound",
      parent_task_id: string | null,
      depth: int,

      // Execution Details
      action: {
        type: "llm_call" | "tool_invocation" | "composite",
        parameters: {...},
        model: ModelSpec,
        tools: [ToolSpec]
      },

      // Dependencies
      prerequisites: [task_id],
      produces: [output_id],
      consumes: [input_id],

      // Resource Estimates
      estimates: {
        time: {expected: float, confidence_interval: [float, float]},
        cost: {expected: float, max: float},
        tokens: {input: int, output: int}
      },

      // Contingencies
```

```
      contingencies: [Contingency],

      // Verification
      success_criteria: [Criterion],
      verification_method: string
    }
  ],

  // Dependency Structure
  dependency_graph: {
    nodes: [task_id],
    edges: [{from: task_id, to: task_id, type: DependencyType}]
  },

  // Execution Order
  execution_sequence: [
    {
      phase: int,
      parallel_groups: [[task_id]]  // Tasks in same group can run in parallel
    }
  ],

  // Overall Estimates
  overall_estimates: {
    min_time: float,
    expected_time: float,
    max_time: float,
    expected_cost: float,
    max_cost: float,
    total_tokens: int
  },

  // Optimization Info
  optimization: {
    objective: OptimizationObjective,
    parallelization_opportunities: int,
    cached_computations: int
  },

  // Risk Assessment
  risk: {
    overall_risk: RiskLevel,
    high_risk_tasks: [task_id],
    failure_probability: float,
    mitigation_strategies: [string]
  }
}
```

3. Execution Engine

3.1 Purpose & Responsibilities

Core Function: Eseguire il plan generato, monitorando progresso, verificando risultati, e adattando dinamicamente quando necessario.

Responsibilities: 1. **Task Execution**: Invocare azioni (LLM calls, tool usage) 2. **State Management**: Tracciare stato corrente, output intermedi 3. **Monitoring**: Osservare progresso, resource usage, problemi 4. **Verification**: Validare output contro success criteria 5. **Adaptation**: Replan o adjust quando si incontrano ostacoli 6. **Safety**: Enforce bounds, prevent unsafe actions

3.2 Architecture Interna

```
+----------------------------------------------------------------+
|                     EXECUTION ENGINE                           |
|                                                                |
|  Input: ExecutionPlan + Context                                |
|    |                                                           |
|  +------------------------------------------+                 |
|  |  1. EXECUTION COORDINATOR                |                 |
|  |  * Manage execution flow                 |                 |
|  |  * Dispatch tasks                        |                 |
|  |  * Handle parallelization                |                 |
|  +----------------+-------------------------+                 |
|                   |                                            |
|  +------------------------------------------+                 |
|  |  2. TASK EXECUTOR                        |                 |
|  |  * Invoke LLM                            |                 |
|  |  * Call tools                            |                 |
|  |  * Manage context                        |                 |
|  +----------------+-------------------------+                 |
|                   |                                            |
|  +------------------------------------------+                 |
|  |  3. STATE MANAGER                        |                 |
|  |  * Track execution state                 |                 |
|  |  * Store intermediate results            |                 |
|  |  * Maintain working memory               |                 |
|  +----------------+-------------------------+                 |
|                   |                                            |
|  +------------------------------------------+                 |
|  |  4. MONITORING SYSTEM                    |                 |
|  |  * Track resource usage                  |                 |
|  |  * Detect anomalies                      |                 |
|  |  * Log execution trace                   |                 |
|  +----------------+-------------------------+                 |
|                   |                                            |
|  +------------------------------------------+                 |
```

```
|  |  5. VERIFICATION LAYER                   |                 |
|  |  * Validate outputs                      |                 |
|  |  * Check success criteria                |                 |
|  |  * Safety verification                   |                 |
|  +----------------+-------------------------+                 |
|                   |                                            |
|  +----------------------------------------+                   |
|  |  6. ADAPTATION ENGINE                    |                 |
|  |  * Detect need for replanning            |                 |
|  |  * Apply contingencies                   |                 |
|  |  * Dynamic strategy adjustment           |                 |
|  +----------------+-------------------------+                 |
|                   |                                            |
|  Output: ExecutionResult                                      |
|  {                                                             |
|    status: SUCCESS | FAILURE | PARTIAL,                       |
|    outputs: {...},                                            |
|    trace: [...],                                              |
|    stats: {...}                                               |
|  }                                                             |
+--------------------------------------------------------------+
```

3.3 Execution Coordinator

Purpose: Orchestrare esecuzione di piano, gestendo dipendenze e parallelizzazione.

Execution Loop (Main Algorithm):

```
Function EXECUTE_PLAN(plan):

  # Initialize
  state = INITIALIZE_STATE(plan)
  execution_trace = []

  # Main execution loop
  FOR phase IN plan.execution_sequence:

    # Execute parallel groups in this phase
    FOR parallel_group IN phase.parallel_groups:

      # Check if all prerequisites satisfied
      IF NOT ALL_PREREQUISITES_MET(parallel_group, state):
        HANDLE_BLOCKED_EXECUTION(parallel_group, state)
        CONTINUE

      # Execute tasks in parallel (if multiple)
      IF len(parallel_group) > 1:
        results = PARALLEL_EXECUTE(parallel_group, state)
      ELSE:
```

```
      results = [EXECUTE_TASK(parallel_group[0], state)]

    # Process results
    FOR task, result IN zip(parallel_group, results):

      # Verify result
      verification = VERIFY_RESULT(task, result, state)

      IF verification.success:
        # Update state with successful result
        state = UPDATE_STATE(state, task, result)
        execution_trace.append({task, result, "SUCCESS"})
      ELSE:
        # Handle failure
        recovery = APPLY_CONTINGENCY(task, result, verification)

        IF recovery.success:
          state = UPDATE_STATE(state, task, recovery.result)
          execution_trace.append({task, recovery.result, "RECOVERED"})
        ELSE:
          # Failure not recoverable at task level
          RETURN PARTIAL_RESULT(state, execution_trace, task)

# All phases completed
RETURN SUCCESS_RESULT(state, execution_trace)
```

State Machine:

```
+--------------------------------------------------------------+
|                  EXECUTION STATE MACHINE                     |
|                                                              |
|                      +----------+                            |
|                      |  START   |                            |
|                      +----+-----+                            |
|                           |                                  |
|                  +-----------------+                         |
|                  | READY_TO_START |                          |
|                  +--------+-------+                          |
|                           |                                  |
|         +-----------------+-----------------+               |
|         |                                   |               |
|  +---------------+                  +---------------+       |
|  |  EXECUTING    |                  |   BLOCKED     |       |
|  |  (Running)    |                  | (Waiting Dep)|       |
|  +------+-------+                   +------+-------+       |
|         |                                   |               |
|  +--------------+                          |              |
|  |  VERIFYING   | <-------------------------+              |
|  | (Check Result)|                                         |
```

```
|  +------+-------+                                            |
|         |                                                    |
|  +----------------------+                                   |
|  |  Verification Pass?  |                                   |
|  +--+----------------+--+                                   |
|     | YES            | NO                                    |
|     |                |                                       |
| +---------+      +-------------+                           |
| |SUCCESS |      |  FAILED    |                             |
| +---------+      +-----+------+                            |
|                         |                                    |
|              +-----------------+                             |
|              | Contingency OK? |                             |
|              +---+----------+--+                             |
|                  | YES      | NO                             |
|                  |          |                                |
|           +------------+  +-----------+                     |
|           |RECOVERING |  | ABORTED  |                       |
|           +-----+-----+  +-----------+                      |
|                 |                                            |
|          (Back to EXECUTING)                                 |
+---------------------------------------------------------------+
```

3.4 Task Executor

Purpose: Eseguire singoli task (primitive actions).

Execution Types:

```
+------------------------------------------------------------+
|                     TASK EXECUTION TYPES                     |
|                                                            |
|  1. LLM_CALL                                               |
|     * Prepare context (system + user prompt)               |
|     * Route to appropriate model                           |
|     * Invoke with tools if needed                          |
|     * Extract structured output                            |
|                                                            |
|  2. TOOL_INVOCATION                                        |
|     * Retrieve tool from registry                          |
|     * Validate inputs                                      |
|     * Execute tool with safety bounds                      |
|     * Capture output                                       |
|                                                            |
|  3. COMPOUND_ACTION                                        |
|     * Execute sub-plan recursively                         |
|     * Aggregate results                                    |
|                                                            |
|  4. VERIFICATION_STEP                                      |
```

```
|     * Run validation logic                                 |
|     * Check assertions                                     |
|     * Produce verification report                          |
+------------------------------------------------------------+
```

LLM Call Execution Flow:

```
Task: "Generate JWT token encoding function"
  |
+----------------------------------------+
|  1. Context Preparation                |
|                                        |
|  System Prompt:                        |
|    "You are expert Python developer"   |
|                                        |
|  User Prompt:                          |
|    "Generate JWT encoding function     |
|     Requirements: [...]"               |
|                                        |
|  Working Memory:                       |
|    - JWT library choice (PyJWT)        |
|    - Previous implementation context   |
+---------------+------------------------+
               |
+----------------------------------------+
|  2. Model Routing                      |
|  * Task complexity: MEDIUM             |
|  * Context size: 5K tokens             |
|  * Budget remaining: OK                |
|  -> Route to: claude-sonnet-3.5         |
+---------------+------------------------+
               |
+----------------------------------------+
|  3. LLM Invocation                     |
|  * Model: claude-sonnet-3.5            |
|  * Max tokens: 2000                    |
|  * Temperature: 0.2                    |
|  * Tools: [write_file, read_file]      |
+---------------+------------------------+
               |
+----------------------------------------+
|  4. Output Extraction                  |
|  * Parse LLM response                  |
|  * Extract function code               |
|  * Extract any metadata                |
+---------------+------------------------+
               |
    Task Result
```

Tool Invocation Flow:

```
Tool: "write_file"
Parameters: {path: "jwt_utils.py", content: "..."}
  |
+---------------------------------------+
|  1. Safety Verification               |
|  * Check path against whitelist       |
|  * Verify not overwriting critical    |
|  * Check size limits                  |
+---------------+-----------------------+
                |
+---------------------------------------+
|  2. Input Validation                  |
|  * Schema validation                  |
|  * Type checking                      |
|  * Constraint verification            |
+---------------+-----------------------+
                |
+---------------------------------------+
|  3. Tool Execution                    |
|  * Invoke tool with parameters        |
|  * Capture stdout/stderr              |
|  * Monitor for errors                 |
+---------------+-----------------------+
                |
+---------------------------------------+
|  4. Output Capture                    |
|  * Parse tool output                  |
|  * Structure as ToolResult            |
|  * Attach metadata                    |
+---------------+-----------------------+
                |
    Tool Result
```

3.5 State Manager

Purpose: Mantenere stato corrente di esecuzione e risultati intermedi.

State Structure:

```
ExecutionState {
  // Task Status
  completed_tasks: {task_id: Result},
  in_progress_tasks: {task_id: StartTime},
  pending_tasks: [task_id],
  failed_tasks: {task_id: FailureInfo},

  // Working Memory (context for execution)
```

```
  working_memory: {
    task_outputs: {output_id: Value},
    intermediate_results: {key: Value},
    context_variables: {var_name: Value}
  },

  // Resource Tracking
  resources_used: {
    time_elapsed: float,
    tokens_consumed: int,
    cost_accumulated: float,
    llm_calls: int,
    tool_invocations: int
  },

  // Dependencies
  dependency_satisfaction: {
    task_id: boolean  // Are all prerequisites met?
  },

  // Errors & Warnings
  errors: [Error],
  warnings: [Warning],

  // Metadata
  execution_start: datetime,
  last_update: datetime
}
```

State Transitions:

```
State Update Operations:

1. TASK_STARTED(task_id, timestamp):
   pending_tasks.remove(task_id)
   in_progress_tasks[task_id] = timestamp

2. TASK_COMPLETED(task_id, result):
   in_progress_tasks.remove(task_id)
   completed_tasks[task_id] = result
   working_memory.task_outputs[task_id] = result.output
   UPDATE_DEPENDENCY_SATISFACTION(task_id)

3. TASK_FAILED(task_id, error):
   in_progress_tasks.remove(task_id)
   failed_tasks[task_id] = error
   errors.append(error)

4. UPDATE_RESOURCES(resource_delta):
```

```
   resources_used += resource_delta

5. SET_CONTEXT(key, value):
   working_memory.context_variables[key] = value
```

Context Management:

```
Working Memory has limited size -> Must manage carefully

+------------------------------------------------------+
|              CONTEXT MANAGEMENT                      |
|                                                      |
|  Hot Context (always in working memory):             |
|  * Current task parameters                           |
|  * Recent task outputs (last 3-5)                    |
|  * Global variables                                  |
|                                                      |
|  Warm Context (retrievable on demand):               |
|  * Older task outputs (stored in episodic memory)    |
|  * Large intermediate results                        |
|                                                      |
|  Cold Context (compressed/summarized):               |
|  * Very old outputs                                  |
|  * Detailed traces (summarized)                      |
+------------------------------------------------------+

Context Size Budget: ~20K tokens
  - If approaching limit:
    1. Summarize old outputs
    2. Move to episodic memory
    3. Keep only essential info
```

3.6 Monitoring System

Purpose: Osservare esecuzione in real-time, detect anomalies, collect telemetry.

Monitoring Dimensions:

```
+------------------------------------------------------------+
|                  MONITORING DASHBOARD                      |
|                                                            |
|  PERFORMANCE METRICS                                       |
|  +---------------------------------------------------+ |
|  | * Latency per task (p50, p95, p99)                | |
|  | * Throughput (tasks/minute)                       | |
|  | * Resource utilization (% of budget)              | |
|  +---------------------------------------------------+ |
|                                                            |
|  QUALITY METRICS                                           |
```

```
|  +--------------------------------------------------+ |
|  | * Task success rate                              | |
|  | * Verification pass rate                         | |
|  | * Error frequency                                | |
|  +--------------------------------------------------+ |
|                                                        |
|  ANOMALY DETECTION                                     |
|  +--------------------------------------------------+ |
|  | * Task taking unusually long                     | |
|  | * Repeated failures on same task                 | |
|  | * Resource usage spike                           | |
|  | * Unexpected error patterns                      | |
|  +--------------------------------------------------+ |
|                                                        |
|  EXECUTION TRACE                                       |
|  +--------------------------------------------------+ |
|  | * Detailed log of all actions                    | |
|  | * Inputs/outputs for each task                   | |
|  | * Decision points and reasoning                  | |
|  | * Timestamps and durations                       | |
|  +--------------------------------------------------+ |
+-------------------------------------------------------+
```

Anomaly Detection:

```
Monitor continuously for:

1. LATENCY ANOMALY
   IF task_duration > expected_duration * 3:
     WARN: "Task taking unusually long"
     CONSIDER: Timeout, interrupt, or investigate

2. REPEATED FAILURE
   IF same_task_failed > 3 times:
     WARN: "Repeated failures detected"
     CONSIDER: Skip task, try alternative, escalate

3. RESOURCE SPIKE
   IF resource_rate > budget_rate * 2:
     WARN: "Resource consumption too high"
     CONSIDER: Throttle, optimize, or abort

4. ERROR PATTERN
   IF similar_errors > threshold:
     WARN: "Systematic error detected"
     CONSIDER: Diagnose root cause, replan

5. STUCK EXECUTION
   IF no_progress_for > timeout:
```

```
    WARN: "Execution appears stuck"
    CONSIDER: Interrupt, diagnose
```

Logging:

```
Execution Trace Entry {
  timestamp: datetime,
  task_id: string,
  event_type: "START" | "COMPLETE" | "FAIL" | "VERIFY",

  // Context
  inputs: {...},
  working_memory_snapshot: {...},

  // Execution Details
  action_taken: string,
  llm_prompt: string (if applicable),
  llm_response: string (if applicable),
  tool_calls: [ToolCall],

  // Results
  output: {...},
  verification_result: {...},

  // Metadata
  duration: float,
  tokens_used: int,
  cost: float,
  model_used: string,

  // Decisions
  decisions_made: [Decision],
  reasoning: string
}
```

3.7 Verification Layer

Purpose: Validare output di ogni task contro success criteria e safety bounds.

Verification Types:

```
+-----------------------------------------------------------+
|                   VERIFICATION HIERARCHY                  |
|                                                           |
|  Level 1: SCHEMA VERIFICATION                             |
|  * Output structure matches expected format               |
|  * Required fields present                                |
|  * Type correctness                                       |
|  -> Fast, automatic, always run                             |
```

```
|                                                          |
|  Level 2: CONSTRAINT VERIFICATION                        |
|  * Output satisfies explicit constraints                 |
|  * Bounds checking (numeric ranges, lengths)             |
|  * Format validation (regex patterns)                    |
|  -> Fast, automatic, always run                          |
|                                                          |
|  Level 3: SEMANTIC VERIFICATION                          |
|  * Output semantically correct                           |
|  * Meaningful in context                                 |
|  * Logically consistent                                  |
|  -> Moderate cost, LLM-based, selective                  |
|                                                          |
|  Level 4: FUNCTIONAL VERIFICATION                        |
|  * Output achieves intended goal                         |
|  * Side effects as expected                              |
|  * Integration with system works                         |
|  -> Expensive, testing required, critical tasks          |
|                                                          |
|  Level 5: SAFETY VERIFICATION                            |
|  * No prohibited actions taken                           |
|  * No security vulnerabilities introduced                |
|  * Compliance requirements met                           |
|  -> Critical, always run for sensitive operations        |
+----------------------------------------------------------+
```

Verification Flow:

```
Task Result
    |
+--------------------------------------+
|  1. Schema Verification              |
|  * Check structure                   |
|  * Validate types                    |
|  IF fail: REJECT immediately         |
+--------------+-----------------------+
               | PASS
+--------------------------------------+
|  2. Constraint Verification          |
|  * Check bounds                      |
|  * Validate format                   |
|  IF fail: REJECT or REQUEST_RETRY    |
+--------------+-----------------------+
               | PASS
+--------------------------------------+
|  3. Safety Verification              |
|  * Scan for prohibited patterns      |
|  * Check against safety rules        |
|  IF fail: REJECT + ALERT             |
```

```
+--------------+----------------------+
               | PASS
+-------------------------------------+
|  4. Semantic Verification (Optional) |
|  * LLM-based quality check           |
|  * Context appropriateness           |
|  IF fail: FLAG for review            |
+--------------+----------------------+
               | PASS
+-------------------------------------+
|  5. Functional Verification (Select) |
|  * Execute tests                     |
|  * Verify side effects               |
|  IF fail: RETRY with fixes           |
+--------------+----------------------+
               | PASS
    ACCEPT Result
```

Success Criteria Evaluation:

```
Task has success_criteria: [Criterion]

Criterion {
  type: "OUTPUT_PROPERTY" | "SIDE_EFFECT" | "TEST_PASS" | "CUSTOM",
  description: string,
  verification_method: {
    type: "AUTOMATIC" | "LLM_CHECK" | "TOOL_EXECUTION",
    parameters: {...}
  },
  required: boolean  // Must pass vs nice-to-have
}

Evaluation Process:
FOR criterion IN task.success_criteria:
  result = VERIFY_CRITERION(criterion, task_output)

  IF criterion.required AND NOT result.passed:
    RETURN VERIFICATION_FAILED(criterion, result.reason)

  IF NOT criterion.required AND NOT result.passed:
    LOG_WARNING(criterion, result.reason)

RETURN VERIFICATION_PASSED
```

Example Verification:

```
Task: "Write JWT encoding function"
Output: (Python function code)

Success Criteria:
```

```
1. Function is valid Python (REQUIRED)
   Method: Parse with ast module
   Result: [v] PASS

2. Function accepts correct parameters (REQUIRED)
   Method: Check signature: encode_token(payload, secret)
   Result: [v] PASS

3. Function returns string (REQUIRED)
   Method: Check return type hint
   Result: [v] PASS

4. Function uses PyJWT library (REQUIRED)
   Method: Check for "import jwt" and "jwt.encode"
   Result: [v] PASS

5. Function handles errors gracefully (OPTIONAL)
   Method: Check for try-except block
   Result: [v] PASS

6. Function has docstring (OPTIONAL)
   Method: Check for docstring
   Result: [x] FAIL (but optional, so warning only)

Overall: PASS (all required criteria met)
```

3.8 Adaptation Engine

Purpose: Detect quando execution devia da plan e adattarsi dinamicamente.

Adaptation Triggers:

```
+-----------------------------------------------------------+
|                   ADAPTATION TRIGGERS                     |
|                                                           |
|  1. TASK FAILURE                                          |
|     * Task execution failed                               |
|     * Verification rejected output                        |
|     -> Apply contingency or replan                         |
|                                                           |
|  2. ASSUMPTION VIOLATION                                  |
|     * Expected precondition not met                       |
|     * Resource not available                              |
|     -> Update assumptions, replan affected subtasks        |
|                                                           |
|  3. OPPORTUNITY DISCOVERY                                 |
|     * Better approach discovered during execution         |
|     * Shortcut found                                      |
|     -> Optimize plan opportunistically                     |
```

```
|                                                          |
|  4. RESOURCE PRESSURE                                    |
|     * Budget running low                                 |
|     * Time pressure increased                            |
|     -> Switch to more efficient strategy                 |
|                                                          |
|  5. CONTEXT CHANGE                                       |
|     * External environment changed                       |
|     * Requirements updated                               |
|     -> Replan with new context                           |
+----------------------------------------------------------+
```

Adaptation Strategies:

```
+------------------------------------------------------------+
|                  ADAPTATION STRATEGIES                     |
|                                                            |
|  STRATEGY 1: CONTINGENCY APPLICATION                       |
|  When: Anticipated failure                                 |
|  Action: Apply pre-planned fallback                        |
|  Cost: Low (already planned)                               |
|  Example: Package not found -> Try alternative name        |
|                                                            |
|  STRATEGY 2: LOCAL REPLANNING                              |
|  When: Unanticipated failure, affects single task          |
|  Action: Replan just the failed task                       |
|  Cost: Medium (limited replanning)                         |
|  Example: API changed -> Find new way to call it           |
|                                                            |
|  STRATEGY 3: SUBTREE REPLANNING                            |
|  When: Failure affects multiple dependent tasks            |
|  Action: Replan failed task and all descendants            |
|  Cost: Medium-High (more extensive)                        |
|  Example: Library choice wrong -> Redo all library-specific |
|                                                            |
|  STRATEGY 4: GLOBAL REPLANNING                             |
|  When: Fundamental assumption violated                     |
|  Action: Restart planning from current state               |
|  Cost: High (redo most work)                               |
|  Example: Requirements completely changed                  |
|                                                            |
|  STRATEGY 5: ESCALATION                                    |
|  When: Adaptation not possible within bounds               |
|  Action: Request human intervention                        |
|  Cost: Blocks progress until human responds                |
|  Example: Ambiguity can't be resolved automatically        |
+------------------------------------------------------------+
```

Adaptation Decision Tree:

```
Failure Detected
    |
Has pre-planned contingency?
+- YES -> Apply Contingency
|        |
|        Success?
|        +- YES -> Continue Execution
|        +- NO -> Proceed to replanning
|
+- NO -> Is failure isolated?
          +- YES (single task) -> Local Replan
          |                      |
          |                      Replan one task
          |                      Continue execution
          |
          +- NO (affects multiple) -> Dependency Analysis
                                      |
                                      How many tasks affected?
                                      +- Few (<5) -> Subtree Replan
                                      |             |
                                      |             Replan affected subtree
                                      |
                                      +- Many (>5) -> Global Replan?
                                                     |
                                                     Check if viable
                                                     +- YES -> Global Replan
                                                     |        (restart from current)
                                                     |
                                                     +- NO -> Escalate to Human
```

Adaptation Algorithm:

```
Function ADAPT(failure, state, plan):

  # Classify failure severity
  severity = CLASSIFY_FAILURE(failure)

  # Check for contingency
  IF plan.has_contingency_for(failure):
    contingency = plan.get_contingency(failure)
    result = APPLY_CONTINGENCY(contingency, state)

    IF result.success:
      RETURN CONTINUE_EXECUTION(state)

  # Analyze impact
  affected_tasks = FIND_AFFECTED_TASKS(failure, plan)

  # Select adaptation strategy
```

```
  IF len(affected_tasks) == 1:
    # Local replanning
    new_task_plan = REPLAN_SINGLE_TASK(affected_tasks[0], state)
    plan = REPLACE_TASK(plan, affected_tasks[0], new_task_plan)
    RETURN CONTINUE_EXECUTION(state)

  ELSE IF len(affected_tasks) <= 5:
    # Subtree replanning
    new_subtree_plan = REPLAN_SUBTREE(affected_tasks, state)
    plan = REPLACE_SUBTREE(plan, affected_tasks, new_subtree_plan)
    RETURN CONTINUE_EXECUTION(state)

  ELSE IF severity < CRITICAL AND resources_allow_replan:
    # Global replanning
    new_plan = REPLAN_FROM_SCRATCH(original_goals, state)
    RETURN RESTART_EXECUTION(new_plan, state)

  ELSE:
    # Escalate
    RETURN ESCALATE_TO_HUMAN(failure, state, reason)
```

Example Adaptation:

```
Scenario: Installing JWT library

Original Plan:
  1. Run: pip install PyJWT
  2. Import jwt in code
  3. Use jwt.encode()

Execution:
  Step 1: pip install PyJWT
  -> FAILURE: "Package not found"

Adaptation:

1. Check Contingency:
   [v] Found: "Try alternative package names"

2. Apply Contingency:
   Attempt 1: pip install python-jwt
   -> FAILURE: "Package not found"

   Attempt 2: Search PyPI for "jwt python"
   -> SUCCESS: Found packages [PyJWT, python-jose, authlib]

   Attempt 3: pip install python-jose
   -> SUCCESS: Installed
```

```
3. Local Replan:
   Affected task: Step 2 (Import statement)
   Old: import jwt
   New: from jose import jwt

   Affected task: Step 3 (API usage)
   Old: jwt.encode(...)
   New: jwt.encode(...) # (API compatible)

4. Continue Execution:
   Execute updated steps 2 and 3
   -> SUCCESS

Outcome: Task completed successfully via adaptation
```

3.9 Execution Engine Output Schema

```
ExecutionResult {
  // Status
  status: "SUCCESS" | "PARTIAL_SUCCESS" | "FAILURE",
  completion_percentage: float,  // 0-100%

  // Outputs
  primary_output: {...},  // Main result
  intermediate_outputs: {task_id: output},
  side_effects: [SideEffect],

  // Execution Trace
  trace: [
    {
      task_id: string,
      timestamp: datetime,
      event: "START" | "COMPLETE" | "FAIL" | "ADAPT",
      details: {...},
      duration: float,
      resources_used: ResourceDelta
    }
  ],

  // Task Results
  completed_tasks: [
    {
      task_id: string,
      result: {...},
      verification: VerificationResult,
      duration: float
    }
  ],
```

```
failed_tasks: [
  {
    task_id: string,
    error: Error,
    attempted_contingencies: [Contingency],
    reason_for_failure: string
  }
],

// Adaptations
adaptations_applied: [
  {
    trigger: AdaptationTrigger,
    strategy: AdaptationStrategy,
    tasks_affected: [task_id],
    outcome: "SUCCESS" | "FAILURE"
  }
],

// Resource Usage
resources: {
  total_time: float,
  llm_calls: int,
  tokens_consumed: {input: int, output: int},
  cost: float,
  tool_invocations: {tool_name: count}
},

// Quality Metrics
metrics: {
  plan_adherence: float,  // % of original plan executed
  adaptation_frequency: float,  // # adaptations / # tasks
  verification_pass_rate: float,  // % tasks passed verification
  efficiency: float  // actual_time / estimated_time
},

// Errors & Warnings
errors: [Error],
warnings: [Warning],

// Context for Reflection
learning_insights: [
  {
    insight: string,
    category: "STRATEGY" | "FAILURE_PATTERN" | "OPTIMIZATION",
    confidence: float
  }
],
```

```
  // Metadata
  execution_id: string,
  plan_id: string,
  start_time: datetime,
  end_time: datetime
}
```

4. Reflection Module

4.1 Purpose & Responsibilities

Core Function: Analizzare episodi di esecuzione completati per estrarre learning, identificare pattern, e migliorare performance future.

Key Distinction: Reflection è **post-execution**, non real-time. Avviene DOPO task completion per informare future executions.

Responsibilities: 1. **Episode Analysis**: Analizzare execution trace per successi/fallimenti 2. **Pattern Extraction**: Identificare pattern ricorrenti 3. **Strategy Learning**: Capire quali strategie funzionano in quali contesti 4. **Performance Analysis**: Valutare efficiency vs expectations 5. **Knowledge Distillation**: Convertire esperienze in reusable knowledge 6. **Memory Update**: Aggiornare episodic memory e pattern cache

4.2 Architecture Interna

```
+----------------------------------------------------------------+
|                    REFLECTION MODULE                            |
|                                                                 |
|  Input: ExecutionResult + Context                               |
|    |                                                            |
|  +------------------------------------------+                  |
|  |  1. EPISODE ANALYZER                     |                  |
|  |  * Parse execution trace                 |                  |
|  |  * Identify key events                   |                  |
|  |  * Compute metrics                       |                  |
|  +-----------------+------------------------+                  |
|                    |                                            |
|  +------------------------------------------+                  |
|  |  2. PATTERN EXTRACTOR                    |                  |
|  |  * Find recurring structures             |                  |
|  |  * Identify success patterns             |                  |
|  |  * Identify failure patterns             |                  |
|  +-----------------+------------------------+                  |
|                    |                                            |
|  +------------------------------------------+                  |
|  |  3. STRATEGY EVALUATOR                   |                  |
|  |  * Assess strategy effectiveness         |                  |
```

```
|  |  * Compare alternatives                |                 |
|  |  * Update strategy preferences         |                 |
|  +----------------+-----------------------+                 |
|                   |                                           |
|  +----------------------------------------+                 |
|  |  4. PERFORMANCE ANALYZER               |                 |
|  |  * Actual vs estimated comparison      |                 |
|  |  * Bottleneck identification           |                 |
|  |  * Efficiency opportunities            |                 |
|  +----------------+-----------------------+                 |
|                   |                                           |
|  +----------------------------------------+                 |
|  |  5. KNOWLEDGE DISTILLER                |                 |
|  |  * Generalize from specific            |                 |
|  |  * Create reusable heuristics          |                 |
|  |  * Update beliefs                      |                 |
|  +----------------+-----------------------+                 |
|                   |                                           |
|  +----------------------------------------+                 |
|  |  6. MEMORY UPDATER                     |                 |
|  |  * Store episode in episodic memory    |                 |
|  |  * Update pattern cache                |                 |
|  |  * Consolidate knowledge               |                 |
|  +----------------+-----------------------+                 |
|                   |                                           |
|  Output: ReflectionInsights + Memory Updates                |
+-------------------------------------------------------------+
```

4.3 Episode Analyzer

Purpose: Processare execution trace per estrarre eventi chiave e metriche.

Analysis Dimensions:

```
+-------------------------------------------------------+
|                  EPISODE ANALYSIS                     |
|                                                       |
|  TEMPORAL ANALYSIS                                    |
|  * Total duration                                     |
|  * Time per phase                                     |
|  * Time per task                                      |
|  * Critical path actual time                          |
|                                                       |
|  SUCCESS/FAILURE ANALYSIS                             |
|  * Which tasks succeeded                              |
|  * Which tasks failed                                 |
|  * Failure reasons taxonomy                           |
|  * Recovery success rate                              |
|                                                       |
```

```
|  RESOURCE ANALYSIS                                          |
|  * Tokens consumed vs estimated                             |
|  * Cost actual vs budget                                    |
|  * Tool usage patterns                                      |
|  * Model usage patterns                                     |
|                                                             |
|  QUALITY ANALYSIS                                           |
|  * Verification pass rates                                  |
|  * Output quality scores                                    |
|  * Rework frequency                                         |
|                                                             |
|  ADAPTATION ANALYSIS                                        |
|  * # of adaptations                                         |
|  * Adaptation triggers                                      |
|  * Adaptation success rate                                  |
|  * Plan adherence percentage                                |
+-------------------------------------------------------------+
```

Key Events Identification:

```
Execution Trace -> Key Events

Event Types to Extract:

1. DECISION POINTS
   * Where agent chose between alternatives
   * Reasoning for choice
   * Outcome of choice

2. FAILURES
   * What failed
   * Why it failed
   * How it was (or wasn't) recovered

3. ADAPTATIONS
   * What triggered adaptation
   * How agent adapted
   * Effectiveness of adaptation

4. BOTTLENECKS
   * Tasks that took unexpectedly long
   * Resource constraints hit
   * Blocking dependencies

5. SURPRISES
   * Unexpected successes
   * Unexpected failures
   * Assumption violations
```

Metrics Computation:

```
Computed Metrics:

EFFICIENCY METRICS:
  efficiency_ratio = actual_time / estimated_time
  cost_efficiency = actual_cost / estimated_cost
  resource_utilization = resources_used / resources_available

QUALITY METRICS:
  success_rate = completed_tasks / total_tasks
  first_time_success = tasks_passed_verification_first_try / total_tasks
  rework_rate = tasks_requiring_retry / total_tasks

ADAPTATION METRICS:
  adaptation_rate = adaptations / total_tasks
  adaptation_success = successful_adaptations / total_adaptations
  plan_stability = tasks_from_original_plan / total_tasks_executed

LEARNING METRICS:
  novel_situations = situations_not_in_memory / total_situations
  pattern_reuse = patterns_applied_from_cache / opportunities
```

4.4 Pattern Extractor

Purpose: Identificare pattern ricorrenti che possono informare future decisioni.

Pattern Types:

```
+-----------------------------------------------------------+
|                    PATTERN TAXONOMY                       |
|                                                           |
|  1. SUCCESS PATTERNS                                      |
|     * Task sequences that work well                       |
|     * Effective strategies for problem types              |
|     * Optimal tool combinations                           |
|     Example: "For Python refactoring, always run tests    |
|              before and after"                            |
|                                                           |
|  2. FAILURE PATTERNS                                      |
|     * Common failure modes                                |
|     * Error co-occurrences                                |
|     * Problematic assumptions                             |
|     Example: "Installing package X often fails on first   |
|              try, but succeeds after pip upgrade"         |
|                                                           |
|  3. OPTIMIZATION PATTERNS                                 |
|     * Tasks that can be parallelized                      |
|     * Caching opportunities                               |
```

```
|     * Shortcuts and simplifications                      |
|     Example: "Can cache library search results for 1hr"  |
|                                                          |
|  4. CONTEXT PATTERNS                                     |
|     * Problem characteristics -> best approach            |
|     * Environment signals -> strategy selection           |
|     Example: "If codebase is TypeScript, verification    |
|              must include type checking"                 |
|                                                          |
|  5. TEMPORAL PATTERNS                                    |
|     * Time-of-day effects                                |
|     * Sequence effects (order matters)                   |
|     Example: "Database operations faster in mornings"    |
+----------------------------------------------------------+
```

Pattern Extraction Process:

```
Episodes in Memory
    |
+--------------------------------------+
|  1. Cluster Similar Episodes         |
|  * Group by task type                |
|  * Group by problem domain           |
|  * Group by strategies used          |
+--------------+-----------------------+
               |
+--------------------------------------+
|  2. Compare Within Clusters          |
|  * What's common to successes?       |
|  * What's common to failures?        |
|  * What differentiates success/fail? |
+--------------+-----------------------+
               |
+--------------------------------------+
|  3. Identify Recurring Structures    |
|  * Frequent subsequences             |
|  * Repeated decision patterns        |
|  * Common adaptation triggers        |
+--------------+-----------------------+
               |
+--------------------------------------+
|  4. Generalize Patterns              |
|  * Abstract from specific instances  |
|  * Identify context conditions       |
|  * Estimate pattern confidence       |
+--------------+-----------------------+
               |
+--------------------------------------+
|  5. Validate Patterns                |
```

```
|  * Statistical significance?         |
|  * Consistent across episodes?       |
|  * Explanable causality?             |
+--------------+----------------------+
               |
    Validated Patterns
```

Pattern Representation:

```
Pattern {
  pattern_id: string,
  type: PatternType,
  name: string,
  description: string,

  // Context where pattern applies
  context: {
    problem_type: string,
    domain: string,
    conditions: [Condition]
  },

  // Pattern structure
  structure: {
    type: "SEQUENCE" | "CHOICE" | "CONDITION" | "OPTIMIZATION",
    elements: [...]
  },

  // Evidence
  evidence: {
    supporting_episodes: [episode_id],
    counter_episodes: [episode_id],
    confidence: float,  // 0-1
    statistical_significance: float
  },

  // Application
  application: {
    when_to_apply: string,
    expected_benefit: string,
    estimated_improvement: {
      time_saving: float,
      cost_saving: float,
      success_rate_increase: float
    }
  },

  // Metadata
  discovered_at: datetime,
```

```
  last_validated: datetime,
  usage_count: int,
  success_when_applied: int
}
```

Example Pattern:

```
Pattern: "Install-Import-Verify Sequence"

Type: SUCCESS_PATTERN
Context: Package installation tasks in Python

Structure:
  SEQUENCE [
    1. Try pip install
    2. If fail, check for alternatives
    3. Import package to verify
    4. Run simple test to confirm functionality
  ]

Evidence:
  * Applied in 45 episodes
  * Success rate: 92% (vs 70% without pattern)
  * Statistical significance: p < 0.01

Application:
  When: Any Python package installation task
  Benefit: Higher success rate, earlier failure detection
  Improvement: ~30% time saving, ~20% higher success

Last Validated: 2024-01-15
Usage Count: 45
Success Rate: 92%
```

4.5 Strategy Evaluator

Purpose: Valutare effectiveness di diverse strategie e aggiornare preferenze.

Strategy Comparison:

```
+-----------------------------------------------------------+
|                STRATEGY EVALUATION MATRIX                 |
|                                                           |
|  Strategy: Planning Approach for Code Refactoring         |
|                                                           |
|  +-----------------------------------------------------+ |
|  |          | Episodes | Success | Avg Time | Avg Cost| |
|  +-----------------------------------------------------+ |
|  | HTN Plan |    25    |   88%   |   180s   | $0.15  | |
```

```
|  | Linear   |    18    |   78%   |    120s   | $0.10  | |
|  | Explore  |    12    |   83%   |    240s   | $0.25  | |
|  +-----------------------------------------------------+ |
|                                                          |
|  Analysis:                                               |
|  * HTN: Best success rate, moderate cost                 |
|  * Linear: Fastest but lower success                     |
|  * Exploratory: High cost, not justified by success      |
|                                                          |
|  Recommendation:                                         |
|  * Default to HTN for code refactoring                   |
|  * Use Linear only for simple, well-understood cases     |
|  * Avoid Exploratory unless high uncertainty             |
+----------------------------------------------------------+
```

Strategy Learning Process:

```
For each strategy type:
    |
+---------------------------------------+
|  1. Collect Usage Data                |
|  * # times used                       |
|  * Contexts used                      |
|  * Outcomes                           |
+--------------+------------------------+
               |
+---------------------------------------+
|  2. Compute Performance Metrics       |
|  * Success rate                       |
|  * Average time                       |
|  * Average cost                       |
|  * Quality of results                 |
+--------------+------------------------+
               |
+---------------------------------------+
|  3. Identify Contexts                 |
|  * Where does it work best?           |
|  * Where does it fail?                |
|  * What conditions affect it?         |
+--------------+------------------------+
               |
+---------------------------------------+
|  4. Compare to Alternatives           |
|  * Which strategy is best when?       |
|  * Trade-offs between strategies      |
+--------------+------------------------+
               |
+---------------------------------------+
|  5. Update Strategy Preferences       |
```

```
|  * Adjust default choices           |
|  * Update selection heuristics      |
|  * Refine context-strategy mapping  |
+-------------------------------------+
```

4.6 Performance Analyzer

Purpose: Analizzare performance effettiva vs aspettative, identificare bottleneck.

Analysis Types:

```
+-----------------------------------------------------------+
|              PERFORMANCE ANALYSIS TYPES                   |
|                                                           |
|  1. ESTIMATION ACCURACY                                   |
|     Compare estimated vs actual:                          |
|     * Time per task                                       |
|     * Cost per task                                       |
|     * Resource usage                                      |
|     -> Improve estimation models                           |
|                                                           |
|  2. BOTTLENECK IDENTIFICATION                             |
|     Find tasks that:                                      |
|     * Took disproportionately long                        |
|     * Blocked other tasks                                 |
|     * Consumed excessive resources                        |
|     -> Target for optimization                             |
|                                                           |
|  3. EFFICIENCY OPPORTUNITIES                              |
|     Identify:                                             |
|     * Redundant work                                      |
|     * Missed parallelization                              |
|     * Suboptimal tool choices                             |
|     -> Apply optimizations                                 |
|                                                           |
|  4. RESOURCE UTILIZATION                                  |
|     Analyze:                                              |
|     * Token efficiency (output quality per token)         |
|     * Model selection appropriateness                     |
|     * Tool usage patterns                                 |
|     -> Optimize resource allocation                        |
+-----------------------------------------------------------+
```

Bottleneck Detection:

```
Analysis Process:

1. IDENTIFY SLOW TASKS
   FOR task IN executed_tasks:
```

```
    IF task.actual_time > task.estimated_time * 2:
      FLAG as bottleneck candidate

2. ANALYZE ROOT CAUSE
   FOR bottleneck IN candidates:
     * Was it LLM inference time?
     * Was it tool execution time?
     * Was it waiting for dependencies?
     * Was it due to retries/failures?

3. ASSESS IMPACT
   * Did it block other tasks?
   * How much did it extend total time?
   * Could it be optimized?

4. GENERATE RECOMMENDATIONS
   * Use smaller model if quality acceptable
   * Cache results if repeated
   * Parallelize if dependencies allow
   * Choose different tool/approach
```

Example Performance Analysis:

```
Episode: "Refactor Authentication Module"

Estimated Total Time: 300s
Actual Total Time: 450s
Efficiency: 0.67 (below target of 0.8)

Breakdown:
+--------------------------------------------------------+
| Task                  | Est | Actual | Variance      |
+--------------------------------------------------------+
| Research JWT libs     | 30s |  28s   |  -7% [v]       |
| Design token flow     | 45s |  52s   | +16%          |
| Install library       | 10s |  35s   | +250%      |
| Create encoder        | 60s |  65s   |  +8% [v]       |
| Create decoder        | 60s |  58s   |  -3% [v]       |
| Update auth flow      | 90s | 145s   | +61%        |
| Write tests           |120s | 115s   |  -4% [v]       |
| Run tests             | 15s |  12s   | -20% [v]       |
+--------------------------------------------------------+

BOTTLENECKS IDENTIFIED:

1. Install Library (+25s)
   Root Cause: Package name issues, multiple retries
   Impact: 25s delay, 17% of overrun
   Recommendation: Improve package search, better contingencies
```

```
2. Update Auth Flow (+55s)
   Root Cause: Integration complexity underestimated
   Impact: 55s delay, 37% of overrun
   Recommendation: Better complexity estimation for integration tasks

EFFICIENCY OPPORTUNITIES:

1. Encoder & Decoder could be parallelized
   Current: 65s + 58s = 123s sequential
   Potential: max(65s, 58s) = 65s parallel
   Saving: 58s (13% of total time)

2. Library research could be cached
   If similar task recurs, save 28s

UPDATED ESTIMATES:
Next similar task estimate adjusted:
- Install library: 30s (was 10s)
- Update auth flow: 120s (was 90s)
- Apply parallelization pattern
-> New estimate: 280s (vs original 300s, actual 450s)
```

4.7 Knowledge Distiller

Purpose: Generalizzare da episodi specifici a conoscenza riusabile.

Distillation Process:

```
Specific Experiences -> General Knowledge

+-----------------------------------------------------+
|  LEVEL 0: RAW EPISODES                              |
|  * Specific task executions                         |
|  * Concrete actions and results                     |
|  * Full context and details                         |
|  Example: "Installed PyJWT in project X on date Y"  |
+--------------+--------------------------------------+
               | GENERALIZE
+-----------------------------------------------------+
|  LEVEL 1: PATTERNS                                  |
|  * Recurring structures across episodes             |
|  * Common success/failure modes                     |
|  * Context-strategy associations                    |
|  Example: "PyJWT installation works via pip"        |
+--------------+--------------------------------------+
               | ABSTRACT
+-----------------------------------------------------+
|  LEVEL 2: HEURISTICS                                |
```

```
|  * Rules of thumb                                 |
|  * Decision guidelines                            |
|  * Optimization principles                        |
|  Example: "For JWT in Python, prefer PyJWT"       |
+--------------+-----------------------------------+
               | FORMALIZE
+--------------------------------------------------+
|  LEVEL 3: STRATEGIES                              |
|  * Reusable approaches                            |
|  * Template solutions                             |
|  * Meta-knowledge                                 |
|  Example: "Token-based auth pattern for APIs"     |
+--------------+-----------------------------------+
               | INTERNALIZE
+--------------------------------------------------+
|  LEVEL 4: INTUITIONS                              |
|  * Implicit preferences                           |
|  * Automatic choices                              |
|  * Expertise                                      |
|  Example: "Auth refactoring -> security first"     |
+--------------------------------------------------+
```

Heuristic Generation:

```
Heuristic {
  heuristic_id: string,
  category: "SELECTION" | "ORDERING" | "OPTIMIZATION" | "SAFETY",
  description: string,

  // When to apply
  applicability: {
    problem_types: [string],
    conditions: [Condition],
    confidence_required: float
  },

  // What to do
  guidance: {
    type: "PREFER" | "AVOID" | "ALWAYS" | "NEVER" | "IF_THEN",
    action: string,
    rationale: string
  },

  // Supporting evidence
  evidence: {
    patterns: [pattern_id],
    success_rate: float,
    episodes: [episode_id]
  }
```

```
}
```

Example Heuristics:

```
Heuristic 1: "Verify Before Commit"
Category: SAFETY
Applicability: Code modification tasks
Guidance: ALWAYS run tests before committing code changes
Rationale: Prevents broken commits, catches issues early
Evidence: 95% of successful code changes included pre-commit testing
          80% of failed deployments lacked pre-commit testing

Heuristic 2: "Small Model for Simple Tasks"
Category: OPTIMIZATION
Applicability: Tasks with complexity=LOW, no creativity needed
Guidance: PREFER small/fast model over large model
Rationale: Cost/latency optimization without quality loss
Evidence: 89% of simple tasks succeeded with small model
          Cost saving: 90%, Time saving: 70%

Heuristic 3: "Parallel Independence"
Category: OPTIMIZATION
Applicability: Tasks with no data dependencies
Guidance: IF no dependencies THEN parallelize
Rationale: Linear speedup for independent tasks
Evidence: 2-4x speedup observed in 78% of opportunities

Heuristic 4: "Backup Before Destructive"
Category: SAFETY
Applicability: Tasks with irreversible operations
Guidance: ALWAYS create backup/checkpoint before destructive ops
Rationale: Enables rollback if things go wrong
Evidence: 100% of recovered failures had backups
          0% recovery rate without backups
```

4.8 Memory Updater

Purpose: Persistere insights dalla reflection in memory systems.

Update Operations:

```
+------------------------------------------------------------+
|                  MEMORY UPDATE OPERATIONS                  |
|                                                            |
|  1. EPISODIC MEMORY UPDATE                                 |
|     * Store complete episode                               |
|     * Add metadata (tags, category, outcome)               |
|     * Create embeddings for semantic search                |
|     * Link to related episodes                             |
```

```
|                                                           |
|  2. PATTERN CACHE UPDATE                                  |
|     * Add newly discovered patterns                       |
|     * Update existing pattern statistics                  |
|     * Retire low-confidence patterns                      |
|     * Consolidate similar patterns                        |
|                                                           |
|  3. STRATEGY PREFERENCES UPDATE                           |
|     * Adjust strategy selection weights                   |
|     * Update context-strategy mappings                    |
|     * Refine estimation models                            |
|                                                           |
|  4. HEURISTICS UPDATE                                     |
|     * Add new heuristics                                  |
|     * Strengthen validated heuristics                     |
|     * Weaken contradicted heuristics                      |
|     * Remove obsolete heuristics                          |
+-----------------------------------------------------------+
```

Memory Consolidation:

```
Periodic Consolidation Process:

1. MERGE SIMILAR EPISODES
   * Find highly similar episodes
   * Abstract common structure
   * Keep representative + summary
   * Save storage space

2. STRENGTHEN VALIDATED PATTERNS
   * Patterns confirmed by new episodes
   * Increase confidence score
   * Promote to higher priority

3. PRUNE LOW-VALUE CONTENT
   * Patterns with low confidence
   * Episodes very old and never retrieved
   * Heuristics never applied
   -> Archive or delete

4. RESOLVE CONFLICTS
   * Patterns that contradict
   * Heuristics that conflict
   -> Keep higher-evidence version

5. GENERALIZE WHERE POSSIBLE
   * Multiple specific patterns
   -> Combine into general pattern
```

4.9 Reflection Output Schema

```
ReflectionInsights {
  // Episode Summary
  episode_id: string,
  episode_summary: {
    task_type: string,
    outcome: "SUCCESS" | "PARTIAL" | "FAILURE",
    key_events: [Event],
    duration: float,
    cost: float
  },

  // Performance Analysis
  performance: {
    efficiency_ratio: float,
    bottlenecks: [
      {
        task_id: string,
        delay: float,
        root_cause: string,
        recommendation: string
      }
    ],
    estimation_errors: {
      time_error: float,  // actual/estimated
      cost_error: float
    }
  },

  // Patterns Discovered
  new_patterns: [Pattern],
  pattern_validations: [
    {
      pattern_id: string,
      outcome: "VALIDATED" | "CONTRADICTED",
      evidence: string
    }
  ],

  // Strategy Insights
  strategy_evaluations: [
    {
      strategy: string,
      effectiveness: float,
      context: string,
      recommendation: "INCREASE_USE" | "DECREASE_USE" | "REFINE"
    }
```

```
],

// Learned Heuristics
new_heuristics: [Heuristic],
heuristic_updates: [
  {
    heuristic_id: string,
    confidence_delta: float,
    reason: string
  }
],

// Improvement Opportunities
opportunities: [
  {
    type: "OPTIMIZATION" | "QUALITY" | "COST_REDUCTION" | "SPEED",
    description: string,
    potential_impact: string,
    implementation_suggestion: string
  }
],

// Failures & Lessons
failures: [
  {
    what_failed: string,
    why_failed: string,
    how_to_prevent: string,
    generalizability: float
  }
],

// Questions & Uncertainties
open_questions: [
  {
    question: string,
    why_uncertain: string,
    how_to_resolve: string
  }
],

// Memory Updates
memory_updates: {
  episodic_memory: "STORED",
  patterns_added: int,
  patterns_updated: int,
  heuristics_added: int,
  heuristics_updated: int
```

```
  },

  // Meta-Learning
  meta_insights: [
    {
      insight: string,
      category: "LEARNING_STRATEGY" | "ADAPTATION" | "LIMITS",
      implication: string
    }
  ]
}
```

5. Cognitive Layer Integration

5.1 Complete Flow Example

Scenario: User requests "Add rate limiting to API endpoints"

```
+--------------------------------------------------------------+
|  PHASE 1: GOAL ANALYSIS                                      |
|                                                              |
|  Input: "Add rate limiting to API endpoints"                 |
|    |                                                         |
|  Semantic Parsing:                                           |
|    Intent: MODIFICATION (adding feature)                     |
|    Entities: ["rate limiting", "API endpoints"]              |
|    Actions: ["add", "implement"]                             |
|    |                                                         |
|  Goal Extraction:                                            |
|    Primary: Implement rate limiting on API                   |
|    Sub-goals:                                                |
|      1. Choose rate limiting strategy                        |
|      2. Select rate limiting library/approach                |
|      3. Implement rate limiter                               |
|      4. Apply to API endpoints                               |
|      5. Test rate limiting behavior                          |
|    |                                                         |
|  Constraints:                                                |
|    * Must not break existing endpoints                       |
|    * Should be configurable                                  |
|    * Must handle distributed systems (implied)               |
|    |                                                         |
|  Context:                                                    |
|    * Codebase: Python FastAPI application                    |
|    * Similar past task: "Add authentication" (retrieved)     |
|    * Pattern: Middleware pattern applicable                  |
|    |                                                         |
|  Complexity: MODERATE (5 subtasks, moderate uncertainty)     |
```

```
|  Strategy: HTN Planning recommended                          |
+--------------------------------------------------------------+
              |
+--------------------------------------------------------------+
|  PHASE 2: PLANNING                                           |
|                                                              |
|  Strategy Selected: HTN Planning                             |
|    |                                                         |
|  Task Decomposition:                                         |
|    L0: Add rate limiting                                     |
|    L1:                                                       |
|      1.1 Research rate limiting approaches                   |
|      1.2 Design rate limiting architecture                   |
|      1.3 Implement rate limiter                              |
|      1.4 Integrate with FastAPI                              |
|      1.5 Test and validate                                   |
|    L2:                                                       |
|      1.1.1 Search for FastAPI rate limiting libraries        |
|      1.1.2 Compare options (slowapi, fastapi-limiter)        |
|      1.1.3 Choose best fit                                   |
|      1.3.1 Install chosen library                            |
|      1.3.2 Configure rate limits                             |
|      1.4.1 Add middleware to app                             |
|      1.4.2 Apply decorators to endpoints                     |
|      1.5.1 Write test cases                                  |
|      1.5.2 Run tests                                         |
|      1.5.3 Manual verification                               |
|    |                                                         |
|  Dependency Analysis:                                        |
|    1.1 -> 1.2 -> 1.3 -> 1.4 -> 1.5 (mostly sequential)          |
|    1.5.1 -> 1.5.2 (sequential within test phase)             |
|    |                                                         |
|  Resource Estimates:                                         |
|    Total: ~240s, $0.18, 85K tokens                           |
|    |                                                         |
|  Contingencies:                                              |
|    * If library install fails -> try alternative              |
|    * If tests fail -> debug and retry                         |
+--------------------------------------------------------------+
              |
+--------------------------------------------------------------+
|  PHASE 3: EXECUTION                                          |
|                                                              |
|  Task 1.1.1: Search for FastAPI rate limiting libraries      |
|    -> Execute: Web search + document read                     |
|    -> Result: Found [slowapi, fastapi-limiter, ...]           |
|    -> Verification: [v] PASS                                    |
|    |                                                         |
```

```
|  Task 1.1.2: Compare options                                  |
|    -> Execute: LLM analysis of options                         |
|    -> Result: "slowapi recommended (simple, Redis-backed)"     |
|    -> Verification: [v] PASS                                    |
|    |                                                          |
|  Task 1.1.3: Choose best fit                                  |
|    -> Execute: Decision based on analysis                      |
|    -> Result: "slowapi selected"                               |
|    -> Verification: [v] PASS                                    |
|    |                                                          |
|  Task 1.3.1: Install chosen library                           |
|    -> Execute: pip install slowapi                             |
|    -> Result: FAILURE "Package not found"                      |
|    -> Apply Contingency: Try "pip install slowapi"             |
|    -> Result: SUCCESS (typo correction worked)                 |
|    -> Verification: [v] PASS                                    |
|    |                                                          |
|  Task 1.3.2: Configure rate limits                            |
|    -> Execute: Write configuration code                        |
|    -> Result: Configuration created                            |
|    -> Verification: [v] PASS                                    |
|    |                                                          |
|  ... (continues through all tasks)                            |
|    |                                                          |
|  Final Result: SUCCESS                                        |
|    * All tests passing                                        |
|    * Rate limiting working as expected                        |
|    * Documentation updated                                    |
+--------------------------------------------------------------+
              |
+--------------------------------------------------------------+
|  PHASE 4: REFLECTION                                          |
|                                                              |
|  Episode Analysis:                                            |
|    * Total time: 265s (vs estimated 240s) = 1.1x              |
|    * Total cost: $0.19 (vs estimated $0.18) = 1.05x           |
|    * Success rate: 100% (1 retry needed)                      |
|    * Plan adherence: 95%                                      |
|    |                                                          |
|  Patterns Discovered:                                         |
|    * "slowapi" is preferred library for FastAPI               |
|      rate limiting (add to pattern cache)                     |
|    * Middleware pattern worked well                           |
|      (validate existing pattern)                              |
|    |                                                          |
|  Performance Insights:                                        |
|    * Installation phase took longer due to typo               |
|    * Testing was faster than estimated                        |
```

```
|    * Overall efficiency: 0.9 (good)                            |
|    |                                                           |
|  Learned Heuristic:                                            |
|    "For FastAPI features, check if middleware                  |
|     approach is applicable before custom implementation"       |
|    |                                                           |
|  Memory Updates:                                               |
|    * Episode stored in episodic memory                         |
|    * Pattern "FastAPI-rate-limiting" added                     |
|    * Estimation model for API features refined                 |
|    |                                                           |
|  Opportunities Identified:                                     |
|    * Could cache library search results                        |
|    * Could create template for API feature additions           |
+---------------------------------------------------------------+
```

5.2 Caratteristiche del Cognitive Layer

Proprietà Chiave:

```
MODULARITÀ
  * Ogni componente ha responsabilità chiare
  * Interfacce pulite tra componenti
  * Componenti migliorabili indipendentemente

BOUNDED EMERGENCE
  * Reasoning LLM entro framework strutturato
  * Validazione esplicita ad ogni stadio
  * Bounds di sicurezza enforced ovunque

APPRENDIMENTO & ADATTAMENTO
  * Apprende da ogni esecuzione
  * Adatta strategie basate sull'esperienza
  * Costruisce conoscenza riusabile nel tempo

TRASPARENZA
  * Trace di esecuzione complete
  * Reasoning esplicito catturato
  * Decisioni spiegabili

ROBUSTEZZA
  * Multipli layer di validazione
  * Degradazione graduale
  * Recovery automatico da errori quando possibile
```

5.3 Caratteristiche di Performance

Breakdown Latenza (task complesso tipico):

```
Analisi Goal:      15-30s  (10-15%)
Pianificazione:    30-60s  (20-30%)
Esecuzione:       120-240s (50-70%)
Reflection:        20-40s  (10-15%)
--------------------------------
Totale:           185-370s

Nota: Reflection è spesso asincrona,
      non blocca risposta utente
```

Utilizzo Risorse:

```
Token:
  * Analisi Goal: 5-10K
  * Pianificazione: 10-20K
  * Esecuzione: 40-80K (varia per task)
  * Reflection: 8-15K
  Totale: 63-125K token per task complesso

Chiamate LLM:
  * Analisi Goal: 2-3
  * Pianificazione: 3-5
  * Esecuzione: 10-30 (dipende da task)
  * Reflection: 2-4
  Totale: 17-42 chiamate per task complesso

Costo:
  * $0.15-0.35 per task complesso (dipende da modello)
```

Prossimo: `03-memory-system.md` -> Specifiche dettagliate dell'architettura di memoria