Contents

Roadmap di Implementazione: Reflective Adaptive Agent

Guida Pratica per Software Engineer con Agenti AI

Versione: 1.0 **Ultimo aggiornamento**: 2025-11-11 **Target**: Software Engineer che lavorano con agenti AI multipli (Claude Code, Cursor, GitHub Copilot, ecc.) **Durata stimata**: 12-16 settimane per MVP + Production Ready

---

[LIST] Indice

[TARGET] Filosofia di Implementazione con AI Agents

Principi Guida

1. Divide et Impera: Scomponi task complessi in subtask piccoli e ben definiti - [OK] Ogni task dovrebbe essere completabile in una singola sessione di codifica (1-3 ore) - [OK] Ogni file dovrebbe avere responsabilità chiare e limitate (<500 righe) - [OK] Ogni componente dovrebbe avere interfacce ben definite

2. Test-First Development: Scrivi test prima dell'implementazione - [OK] Gli AI agents sono eccellenti nel generare test da specifiche - [OK] I test forniscono guardrail per l'implementazione - [OK] I test documentano il comportamento atteso

3. Iterazione Rapida: MVP prima, ottimizzazione dopo - [OK] Implementa la versione più semplice che funziona - [OK] Misura le performance reali prima di ottimizzare - [OK] Rafforza solo ciò che è necessario

4. Observable-First: Strumentazione dal giorno 1 - [OK] Logging, tracing, metrics integrate sin dall'inizio - [OK] Facilita debugging e troubleshooting - [OK] Permette di capire cosa sta realmente accadendo

Come Lavorare con AI Agents

Prompt Efficaci:

```
[NO] BAD: "Implementa il planning engine"
[OK] GOOD: "Implementa una classe PlanningEngine con questi requisiti:
- Input: TaskGoal (tipo, vincoli, success criteria)
- Output: ExecutionPlan (lista di steps con dipendenze)
- Usa algoritmo HTN semplificato
- Includi unit tests per scenari: task semplice, task con dipendenze, task impossibile
- Max 300 righe di codice"
```

Workflow Consigliato: 1. **Specifica**: Definisci interfacce e contratti 2. **Test**: Scrivi test cases con AI agent 3. **Implementazione**: Genera codice con AI agent 4. **Review**: Verifica manualmente logica e edge cases 5. **Refactoring**: Ottimizza con AI agent 6. **Integrazione**: Collega componenti e verifica end-to-end

Agents Raccomandati per Task: | Task | Agent Consigliato | Rationale | |——|———————|———–| | Architettura & Design | Claude (Opus/Sonnet) | Ragionamento complesso, trade-offs | | Implementazione Codice | Claude Code, Cursor | Contesto codebase, refactoring | | Testing | GitHub Copilot | Generazione test ripetitivi | | Documentazione | Claude | Linguaggio naturale, chiarezza | | Debugging | Claude Code | Analisi trace, log investigation | | Code Review | Claude | Analisi critica, best practices |

Prerequisiti e Setup

Tecnologie Core

Stack Consigliato: - **Runtime**: Node.js 20+ o Python 3.11+ - **TypeScript**: 5.3+ (type safety cruciale per AI agents) - **Testing**: Vitest/Jest (TS) o pytest (Python) - **Observability**: OpenTelemetry SDK - **Storage**: PostgreSQL 15+ (episodic memory) + Redis 7+ (cache) - **Vector DB**: Pinecone/Weaviate/Qdrant (semantic memory) - **LLM Provider**: Anthropic Claude API (primario)

Setup Iniziale

```
# 1. Inizializza progetto
mkdir reflective-agent
cd reflective-agent
git init

# TypeScript Setup
npm init -y
npm install -D typescript @types/node vitest
npm install opentelemetry-api opentelemetry-sdk-node
npm install @anthropic-ai/sdk zod dotenv

# Python Setup (alternativa)
poetry init
poetry add anthropic opentelemetry-api opentelemetry-sdk pydantic python-dotenv
poetry add --group dev pytest pytest-cov black ruff mypy

# 2. Configura TypeScript
npx tsc --init
# Modifica tsconfig.json:
# - strict: true
# - target: ES2022
# - module: NodeNext
# - moduleResolution: NodeNext

# 3. Setup struttura directory
mkdir -p src/{core,memory,planning,reflection,tools,infrastructure}
mkdir -p tests/{unit,integration,e2e}
mkdir -p docs
```

Configurazione AI Development Environment

VS Code Extensions: - Claude Code (primary) - GitHub Copilot (supplementare) - Cursor (alternativa)

Settings:

```
{
  "editor.formatOnSave": true,
  "editor.codeActionsOnSave": {
    "source.fixAll": true,
    "source.organizeImports": true
  },
  "typescript.preferences.importModuleSpecifier": "relative"
}
```

□ Fase 0: Foundation Setup (Settimana 1)

Goal: Infrastruttura base + primo execution loop funzionante

Task 0.1: Project Structure & Config

Con AI Agent:

```
"Crea una struttura di progetto TypeScript per un agente AI con:
- src/types/core.ts: Definisci interface Task, ExecutionResult, AgentState
- src/config/index.ts: Configuration management con zod validation
- src/infrastructure/logger.ts: Logger con pino, livelli: debug/info/warn/error
- src/infrastructure/telemetry.ts: OpenTelemetry setup con console exporter
- tests/setup.ts: Test environment setup
Genera anche package.json scripts per: dev, test, build, lint"
```

Deliverables: - [x] Struttura directory completa - [x] Config management con validation - [x] Logger funzionante - [x] Telemetry base configurata - [x] Test runner funzionante

Test di Validazione:

```typescript
// tests/unit/infrastructure/logger.test.ts
describe('Logger', () => {
  it('should log at different levels', () => {
    const logger = createLogger({ level: 'debug' });
    expect(() => logger.debug('test')).not.toThrow();
    expect(() => logger.info('test')).not.toThrow();
  });
});
```

Task 0.2: Basic Types & Interfaces

Con AI Agent:

```
"In src/types/core.ts, definisci TypeScript interfaces per:

1. Task: { id, type, input, constraints, metadata }
2. ExecutionResult: { taskId, status, output, error?, metrics }
3. AgentState: { currentTask?, history, memory }
4. Tool: { name, description, parameters schema, execute function }
5. Plan: { steps: Step[], dependencies: Map<string, string[]> }
6. Step: { id, type, tool, input, status }

Usa Zod per runtime validation. Genera unit tests che verificano validation."
```

Deliverables: - [x] Core types definiti con Zod schemas - [x] Runtime validation funzionante - [x] Type exports organizzati - [x] Unit tests per validation

Task 0.3: Minimal Execution Loop (MVP)

Con AI Agent:

```
"Implementa src/core/agent.ts con classe Agent che:

1. Metodo execute(task: Task): Promise<ExecutionResult>
2. Loop interno:
   - Log task ricevuto
   - Chiama LLM per analisi task
   - Esegue azione (per ora solo log)
   - Ritorna risultato

3. Usa Anthropic Claude API
4. Includi error handling basico
5. Aggiungi tracing OpenTelemetry
6. Genera tests con mock LLM

Mantieni sotto 200 righe. Focus su semplicità."
```

Esempio Test:

```typescript
// tests/unit/core/agent.test.ts
describe('Agent', () => {
  it('should execute simple task', async () => {
    const agent = new Agent({ llm: mockLLM });
    const result = await agent.execute({
      type: 'simple',
      input: 'Hello world'
    });
    expect(result.status).toBe('completed');
  });

  it('should handle errors gracefully', async () => {
    const agent = new Agent({ llm: failingMockLLM });
    const result = await agent.execute({
      type: 'failing',
      input: 'Fail'
    });
    expect(result.status).toBe('failed');
    expect(result.error).toBeDefined();
  });
});
```

Deliverables: - [x] Agent class con execute method - [x] Integrazione Claude API - [x] Error handling - [x] Telemetry integration - [x] Unit & integration tests - [x] **Milestone**: Primo task completato end-to-end!

Success Criteria:

```
npm test          # Tutti test passano
```

```
npm run dev       # Agent esegue task di esempio
```

⚙ Fase 1: Core Loop (Settimane 2-3)

Goal: Loop OODA completo (Observe, Orient, Decide, Act) con tool execution

Task 1.1: Tool Registry

Con AI Agent:

```
"Implementa src/tools/registry.ts con classe ToolRegistry:

1. register(tool: Tool): void
2. get(name: string): Tool | undefined
3. list(): Tool[]
4. execute(toolName: string, input: unknown): Promise<unknown>

Implementa anche src/tools/builtin/ con 5 tools essenziali:
- filesystem: { read, write, list }
- web: { fetch, search }
- shell: { execute }
- math: { calculate }
- text: { extract, summarize }

Ogni tool deve:
- Avere schema Zod per input validation
- Gestire errori con try/catch
- Loggare execution con telemetry
- Avere unit tests

Genera tests che verificano validation e execution."
```

Deliverables: - [x] ToolRegistry implementato - [x] 5 built-in tools funzionanti - [x] Input validation con Zod - [x] Tests completi (unit + integration) - [x] Documentazione per ogni tool

Task 1.2: Action Execution Engine

Con AI Agent:

```
"Implementa src/core/executor.ts con classe Executor:

1. executeAction(action: Action): Promise<ActionResult>
2. Supporta action types:
   - 'tool': esegue tool dal registry
   - 'llm': chiama LLM
   - 'composite': esegue azioni multiple in sequenza
```

```
3. Features:
   - Retry logic (3 tentativi con exponential backoff)
   - Timeout handling (default 30s)
   - Result validation
   - Telemetry per ogni execution

4. Genera tests per:
   - Tool execution success/failure
   - Retry mechanism
   - Timeout scenarios
   - Composite actions

Max 300 righe."
```

Deliverables: - [x] Executor class - [x] Retry logic con backoff - [x] Timeout handling - [x] Action type handlers - [x] Comprehensive tests

Task 1.3: Enhanced Agent Loop (OODA)

Con AI Agent:

```
"Refactor src/core/agent.ts per implementare OODA loop:

OBSERVE:
- Carica task context
- Recupera relevant memories (per ora solo history recente)

ORIENT:
- Analizza task con LLM
- Identifica goal e constraints
- Determina available tools

DECIDE:
- Genera plan (per ora: singolo step)
- Seleziona tool appropriato

ACT:
- Esegue action tramite Executor
- Valida risultato
- Aggiorna state

Ogni fase deve:
- Avere span OpenTelemetry separato
- Loggare inputs/outputs
- Gestire errors gracefully

Refactor tests esistenti + aggiungi test per ogni fase OODA."
```

Esempio Test:

```
describe('Agent OODA Loop', () => {
  it('should complete full loop', async () => {
    const agent = new Agent({
      llm: mockLLM,
      executor: mockExecutor
    });

    const result = await agent.execute({
      type: 'web_search',
      input: 'Latest AI news'
    });

    expect(result.status).toBe('completed');
    expect(result.metadata?.phases).toEqual([
      'observe', 'orient', 'decide', 'act'
    ]);
  });
});
```

Deliverables: - [x] OODA loop implementato - [x] Ogni fase tracciata con telemetry - [x] Logging dettagliato - [x] Tests per ogni fase - [x] **Milestone**: Agent esegue tools reali!

Success Criteria:

```
# Test che agent esegue tool reale
npm test -- agent.integration.test.ts

# Esempio: agent legge file e lo riassume
node scripts/test-agent.js \
  --task "Read README.md and summarize it"
```

□ Fase 2: Memory System (Settimane 4-5)

Goal: Sistema di memoria multi-layer (Working, Episodic, Pattern Cache)

Task 2.1: Memory Interfaces & Storage

Con AI Agent:

```
"Implementa src/memory/interfaces.ts con:

1. Interface Memory<T>:
   - store(key: string, value: T, metadata?): Promise<void>
   - retrieve(key: string): Promise<T | null>
   - search(query: string, limit?): Promise<Array<T>>
```

```
   - delete(key: string): Promise<void>

2. Types:
   - Episode: { taskId, input, actions, result, timestamp, metrics }
   - Pattern: { name, frequency, successRate, averageTime, context }
   - WorkingMemoryItem: { key, value, expiresAt, priority }

3. Implementazioni in src/memory/stores/:
   - InMemoryStore (per tests)
   - PostgresStore (episodic memory)
   - RedisStore (working memory)

Genera tests con fixtures per ogni store type."
```

Deliverables: - [x] Memory interfaces definiti - [x] 3 store implementations - [x] Migration scripts per Postgres - [x] Tests con fixtures - [x] Connection pooling configurato

Task 2.2: Working Memory

Con AI Agent:

```
"Implementa src/memory/working.ts con classe WorkingMemory:

1. Context window management:
   - addContext(item: ContextItem): void
   - getContext(): ContextItem[]
   - Mantiene solo ultimo N items (N configurabile, default 50)

2. Priority-based eviction:
   - High priority items persisted longer
   - LRU eviction per low priority

3. Serialization per LLM:
   - toPrompt(): string
   - Formatta context in modo leggibile per LLM

4. Features:
   - Configurabile max size (token count)
   - Compression old items (summarization)

Genera tests che verificano eviction policy e serialization."
```

Deliverables: - [x] WorkingMemory class - [x] Priority-based eviction - [x] Token counting - [x] Serialization formats - [x] Tests completi

Task 2.3: Episodic Memory

Con AI Agent:

```
"Implementa src/memory/episodic.ts con classe EpisodicMemory:

1. Storage:
   - storeEpisode(episode: Episode): Promise<void>
   - Salva in Postgres con indici su: taskId, timestamp, status

2. Retrieval:
   - getRecent(limit: number): Promise<Episode[]>
   - getByTaskType(type: string): Promise<Episode[]>
   - getSuccessful(limit: number): Promise<Episode[]>

3. Analysis:
   - getSuccessRate(taskType?: string): Promise<number>
   - getAverageTime(taskType?: string): Promise<number>

4. Cleanup:
   - Retention policy (default: 30 giorni)
   - Archiving old episodes

Genera tests con multiple episodes e query complesse."
```

Deliverables: - [x] EpisodicMemory class - [x] Postgres schema & migrations - [x] Query methods optimized - [x] Retention policy - [x] Tests con fixtures complessi

Task 2.4: Pattern Cache

Con AI Agent:

```
"Implementa src/memory/patterns.ts con classe PatternCache:

1. Pattern Detection:
   - extractPattern(episodes: Episode[]): Pattern
   - Identifica pattern ricorrenti (es: task type + tool sequence)

2. Pattern Storage:
   - Usa Redis con TTL
   - Key: hash(taskType + context)
   - Value: Pattern con metrics

3. Pattern Matching:
   - findSimilar(task: Task): Promise<Pattern[]>
   - Usa similarity scoring (es: Jaccard per tools)

4. Auto-Learning:
   - Aggiorna pattern dopo ogni task
   - Increment frequency, update success rate

Genera tests con pattern matching scenarios."
```

Deliverables: - [x] PatternCache class - [x] Pattern detection algorithm - [x] Similarity matching - [x] Auto-learning logic - [x] Tests con scenarios realistici - [x] **Milestone**: Agent ricorda e impara!

Success Criteria:

```
# Test memoria funzionante
npm test -- memory.integration.test.ts

# Verifica che agent usa pattern appreso
node scripts/test-learning.js
# Esegui stesso task 3 volte, verifica che:
# - 1st execution: lento, esplora
# - 2nd execution: più veloce, usa episode memory
# - 3rd execution: molto veloce, usa pattern cache
```

[LIST] Fase 3: Planning Engine (Settimane 6-7)

Goal: Decomposizione task gerarchica con gestione dipendenze

Task 3.1: Planning Data Structures

Con AI Agent:

```
"Implementa src/planning/types.ts con:

1. Interface Plan:
   - steps: Step[]
   - dependencies: Map<stepId, stepId[]>
   - estimatedTime: number
   - estimatedCost: number

2. Interface Step:
   - id: string
   - type: 'tool' | 'llm' | 'human_input'
   - action: Action
   - dependencies: string[]
   - status: 'pending' | 'running' | 'completed' | 'failed'
   - result?: unknown
   - error?: Error

3. Utility functions:
   - topologicalSort(plan: Plan): Step[]
   - findExecutableSteps(plan: Plan): Step[]
   - isComplete(plan: Plan): boolean

Genera tests per dependency resolution e ordering."
```

Deliverables: - [x] Plan types definiti - [x] Dependency graph utilities - [x] Topological sort - [x] Tests per edge cases (cicli, missing deps)

Task 3.2: Simple Planning Engine

Con AI Agent:

```
"Implementa src/planning/simple-planner.ts:

1. Classe SimplePlanner:
   - plan(task: Task, context: Context): Promise<Plan>

2. Planning strategy (LLM-based):
   - Prompt LLM con task e available tools
   - Parse risposta LLM in Plan structure
   - Validazione: check tool availability, dependency graph

3. Features:
   - Max steps limit (default: 20)
   - Cycle detection
   - Invalid tool detection

4. Error handling:
   - LLM parsing errors
   - Invalid plans
   - Retry con different prompts

Genera tests con vari task types (semplici, complessi, impossibili)."
```

Prompt Template Esempio:

```
const PLANNING_PROMPT = `
You are a task planning assistant.

TASK: ${task.description}
AVAILABLE TOOLS: ${tools.map(t => t.name).join(', ')}
CONSTRAINTS: ${task.constraints}

Generate a step-by-step plan as JSON:
{
  "steps": [
    { "id": "1", "action": "tool_name", "input": {...}, "dependencies": [] },
    { "id": "2", "action": "tool_name", "input": {...}, "dependencies": ["1"] }
  ]
}

Keep plan simple and efficient. Max 20 steps.
`;
```

Deliverables: - [x] SimplePlanner implementato - [x] LLM prompt engineering - [x] Plan validation - [x] Tests con mock LLM - [x] Error handling robusto

Task 3.3: Hierarchical Planning (HTN-inspired)

Con AI Agent:

```
"Implementa src/planning/htn-planner.ts:

1. Classe HTNPlanner:
   - Supporta task decomposition gerarchica
   - Methods con preconditions e effects

2. Method Templates:
   - CompositePlan: tasks che si decompongono in subtasks
   - Primitive: tasks che mappano direttamente a tools

3. Planning algorithm:
   - Decompose high-level task in subtasks
   - Ricorsivamente decompone subtasks
   - Termina quando tutto è primitivo

4. Example methods:
   - research_topic -> [web_search, extract_info, summarize]
   - analyze_code -> [read_files, find_patterns, report]

Implementa solo 3-5 methods per MVP. Genera tests."
```

Deliverables: - [x] HTNPlanner class - [x] Method library (5 methods) - [x] Decomposition algorithm - [x] Tests per decomposition - [x] **Milestone**: Agent pianifica task complessi!

Success Criteria:

```bash
# Test planning
npm test -- planning.integration.test.ts

# Test con task complesso reale
node scripts/test-planning.js \
  --task "Research latest papers on LLM agents and create summary report"

# Verifica che genera plan multi-step con:
# - Web search
# - Content extraction
# - Summarization
# - Report generation
```

🔄 Fase 4: Reflection & Learning (Settimane 8-9)

Goal: Auto-miglioramento tramite analisi episodi e pattern extraction

Task 4.1: Reflection Module Base

Con AI Agent:

```
"Implementa src/reflection/reflector.ts:

1. Classe Reflector:
   - reflect(episode: Episode): Promise<Reflection>

2. Reflection types:
   - SuccessAnalysis: cosa ha funzionato bene
   - FailureAnalysis: cosa è andato storto, root cause
   - ImprovementSuggestions: come migliorare

3. LLM-based analysis:
   - Prompt con episode details
   - Parse structured output
   - Store insights in episodic memory

4. Metrics extraction:
   - Time per step
   - Cost per step
   - Success rate per tool
   - Error patterns

Genera tests con episode fixtures (success/failure)."
```

Prompt Example:

```
const REFLECTION_PROMPT = `
Analyze this task execution:

TASK: ${episode.task}
ACTIONS: ${episode.actions}
RESULT: ${episode.result}
TIME: ${episode.metrics.totalTime}ms
STATUS: ${episode.status}

Provide analysis as JSON:
{
  "whatWorked": ["point1", "point2"],
  "whatFailed": ["point1", "point2"],
  "improvements": ["suggestion1", "suggestion2"],
  "patterns": ["pattern1", "pattern2"]
}
```

`;
```

**Deliverables**: - [x] Reflector class - [x] LLM-based analysis - [x] Structured output parsing - [x] Metrics extraction - [x] Tests con fixtures

### Task 4.2: Strategy Learning

**Con AI Agent**:

```
"Implementa src/reflection/strategy-learner.ts:

1. Classe StrategyLearner:
   - learn(reflections: Reflection[]): Strategy[]

2. Strategy extraction:
   - Cluster similar reflections
   - Identifica pattern ricorrenti
   - Genera strategy templates

3. Strategy application:
   - match(task: Task): Strategy | null
   - Trova strategy applicabile a task
   - Usa per guidare planning

4. Strategy types:
   - ToolSequence: sequenza tools che funziona bene
   - ErrorRecovery: come gestire errori specifici
   - Optimization: shortcut per task comuni

Genera tests con multiple reflections -> strategy extraction."
```

Deliverables: - [x] StrategyLearner class - [x] Clustering algorithm - [x] Strategy templates - [x] Matching logic - [x] Tests end-to-end

Task 4.3: Adaptive Planning

Con AI Agent:

```
"Refactor src/planning per integrare learned strategies:

1. Update Planner:
   - checkStrategies(task: Task): Strategy[]
   - Se strategy match, usa come base per plan
   - Altrimenti, genera nuovo plan

2. Strategy-guided planning:
   - Inizia con strategy template
   - Adatta a specific task context
   - Fall back a standard planning se necessario
```
```

```
3. Feedback loop:
   - Dopo execution, compare plan vs actual
   - Update strategy success rate
   - Deprecate ineffective strategies

Refactor tests esistenti + nuovi tests per adaptive behavior."
```

Deliverables: - [x] Strategy integration in Planner - [x] Adaptive planning logic - [x] Feedback loop - [x] Tests per adaptive behavior - [x] **Milestone**: Agent impara e migliora!

Success Criteria:

```bash
# Test learning
npm test -- learning.integration.test.ts

# Benchmark learning curve
node scripts/benchmark-learning.js \
  --task "web_search and summarize" \
  --iterations 10

# Verifica che:
# - Iteration 1: ~30s
# - Iteration 5: ~20s (usa pattern)
# - Iteration 10: ~15s (usa strategy ottimizzata)
```

□ Fase 5: Production Hardening (Settimane 10-12)

Goal: Affidabilità, sicurezza, osservabilità production-grade

Task 5.1: Safety Verifier

Con AI Agent:

```
"Implementa src/safety/verifier.ts con 6-layer verification:

LAYER 1 - Input Validation:
- Schema validation con Zod
- Injection detection (SQL, command, prompt)
- Size limits

LAYER 2 - Action Validation:
- Tool whitelist checking
- Permission verification
- Prohibited action detection

LAYER 3 - Resource Bounds:
```

```
- Token budget enforcement
- Time limits
- Cost tracking

LAYER 4 - Output Validation:
- Format checking
- Sensitive data detection
- Size limits

LAYER 5 - Behavioral Bounds:
- Loop detection
- Stuck state detection
- Anomaly detection

LAYER 6 - Audit Logging:
- Log tutte validations
- Alert su violations
- Compliance tracking

Genera tests per ogni layer con violation scenarios."
```

Deliverables: - [x] SafetyVerifier class (6 layers) - [x] Per-layer verification methods - [x] Violation handling - [x] Audit logging - [x] Tests completi per ogni layer

Task 5.2: Error Handling & Recovery

Con AI Agent:

```
"Implementa src/infrastructure/error-handler.ts:

1. Error Classification:
   - Transient (retry): network, timeout, rate limit
   - Permanent (fail): invalid input, auth error
   - Recoverable (fallback): tool failure

2. Recovery Strategies:
   - Retry con exponential backoff
   - Fallback a tool alternativo
   - Graceful degradation
   - Human escalation

3. Circuit Breaker:
   - Track failure rate per tool/service
   - Open circuit dopo N failures
   - Auto-reset dopo timeout

4. Error Context:
   - Capture full context per debugging
   - Store in episodic memory
```

```
   - Use per reflection

Genera tests con vari error scenarios."
```

Deliverables: - [x] ErrorHandler class - [x] Retry strategies - [x] Circuit breaker - [x] Fallback logic - [x] Tests per error scenarios - [x] Error documentation

Task 5.3: Observability Complete

Con AI Agent:

```
"Completa observability stack:

1. Structured Logging (src/infrastructure/logger.ts):
   - JSON logging con structured fields
   - Correlation IDs per request
   - Log levels dinamici
   - Log sampling per high volume

2. Distributed Tracing (src/infrastructure/tracing.ts):
   - OpenTelemetry setup completo
   - Span per ogni operation
   - Trace context propagation
   - Export a Jaeger/Zipkin

3. Metrics (src/infrastructure/metrics.ts):
   - Counters: task_total, task_success, task_failure
   - Histograms: task_duration, llm_latency
   - Gauges: active_tasks, memory_usage
   - Export a Prometheus

4. Dashboards:
   - Grafana dashboard templates
   - Alert rules per anomalies

Genera load tests per verificare observability overhead."
```

Deliverables: - [x] Structured logging completo - [x] Distributed tracing funzionante - [x] Metrics collection - [x] Grafana dashboards - [x] Alert rules - [x] Documentation

Task 5.4: Configuration & Secrets Management

Con AI Agent:

```
"Implementa src/config/manager.ts:

1. Hierarchical Configuration:
   - Defaults hardcoded
   - Environment variables override
```

```
   - Config files override (YAML/JSON)
   - Runtime overrides via API

2. Secrets Management:
   - Integrazione con vault (HashiCorp/AWS Secrets Manager)
   - Encryption at rest
   - Auto-rotation support
   - Never log secrets

3. Feature Flags:
   - Toggle features runtime
   - A/B testing support
   - Gradual rollouts

4. Validation:
   - Zod schemas per config
   - Fail fast on invalid config
   - Clear error messages

Genera tests con varie config sources."
```

Deliverables: - [x] ConfigManager class - [x] Secrets integration - [x] Feature flags - [x] Validation completa - [x] Tests - [x] **Milestone**: Production-ready!

Success Criteria:

```bash
# Test safety
npm test -- safety.test.ts

# Test error handling
npm test -- error-handling.test.ts

# Test observability
npm run test:observability

# Load test
npm run test:load
# - 100 concurrent tasks
# - Success rate >95%
# - p95 latency <10s
# - Zero crashes
```

□ Fase 6: Optimization & Scale (Settimane 13-16)

Goal: Performance optimization, caching, scalabilità orizzontale

Task 6.1: Model Router & Cost Optimization

Con AI Agent:

```
"Implementa src/models/router.ts:

1. Model Registry:
   - Register models: gpt-4, claude-3, llama-70b, etc.
   - Metadata: cost per token, latency, capability score

2. Routing Strategy:
   - Simple tasks -> cheap models (gpt-3.5, claude-haiku)
   - Complex tasks -> capable models (gpt-4, claude-opus)
   - Real-time -> fast models

3. Routing Algorithm:
   - Classify task complexity
   - Check budget remaining
   - Select optimal model
   - Fall back se failure

4. Caching:
   - Cache LLM responses (keyed by prompt hash)
   - TTL based (1h - 24h)
   - Invalidation strategy

Genera tests con vari task complexities."
```

Deliverables: - [x] ModelRouter class - [x] Complexity classifier - [x] Cost tracking - [x] Response caching - [x] Tests + benchmarks

Task 6.2: Performance Optimization

Con AI Agent:

```
"Ottimizza performance:

1. Parallelization:
   - Execute independent steps in parallel
   - Parallel tool execution where safe
   - Batch LLM requests

2. Caching Strategy:
   - L1: In-memory (hot data)
   - L2: Redis (warm data)
   - L3: Postgres (cold data)
   - Cache hit rate target: 60-80%

3. Database Optimization:
```

```
   - Index optimization
   - Query optimization
   - Connection pooling
   - Read replicas

4. Streaming:
   - Stream LLM responses
   - Progressive results
   - Real-time feedback

Genera benchmarks pre/post optimization."
```

Deliverables: - [x] Parallel execution - [x] Multi-layer caching - [x] DB optimizations - [x] Streaming support - [x] Benchmark results

Task 6.3: Horizontal Scalability

Con AI Agent:

```
"Prepara per horizontal scaling:

1. Stateless Design:
   - State in external stores (Redis/Postgres)
   - No local file system dependency
   - Session affinity non richiesta

2. Load Balancing:
   - Health check endpoint
   - Graceful shutdown
   - Request draining

3. Message Queue:
   - Task queue (BullMQ/RabbitMQ)
   - Worker pool architecture
   - Dead letter queue

4. Deployment:
   - Docker containerization
   - Kubernetes manifests
   - Auto-scaling policies

Genera load tests multi-instance."
```

Deliverables: - [x] Stateless refactoring - [x] Task queue integration - [x] Docker setup - [x] K8s manifests - [x] Load tests - [x] **Milestone**: Production scale!

Success Criteria:

```
# Performance benchmarks
npm run benchmark
```

```
# Target metrics:
# - Simple task: <2s (p95)
# - Complex task: <30s (p95)
# - Success rate: >90%
# - Cost: <$0.10 per task (average)
# - Cache hit rate: >60%

# Scale test
npm run test:scale
# - Deploy 5 instances
# - 500 tasks/min sustained
# - Linear scaling verified
```

Best Practices con AI Agents

1. Prompt Engineering per Codice

[OK] Buone Pratiche:

```
TEMPLATE:
"Implementa [COMPONENT] in [FILE]:

REQUIREMENTS:
- [REQ 1]
- [REQ 2]
- [REQ 3]

CONSTRAINTS:
- Max [N] lines
- Use [PATTERN/LIBRARY]
- Follow [STYLE GUIDE]

TESTS:
- [TEST SCENARIO 1]
- [TEST SCENARIO 2]

Return: implementation + tests"
```

[NO] Anti-Pattern:

```
"Fai una classe per gestire la memoria"  # Troppo vago
"Implementa tutto il sistema"             # Troppo ampio
"Scrivi codice perfetto"                  # Aspettative irrealistiche
```

2. Iterazione Efficace

Workflow Consigliato:

```
1. SPECIFICA (5 min)
   -> Definisci interfacce e contratti con AI

2. TESTS (10 min)
   -> Genera test cases con AI
   -> Review manuale test quality

3. IMPLEMENTAZIONE (20 min)
   -> Genera codice con AI
   -> Review critico logica e edge cases

4. REFACTORING (10 min)
   -> Ottimizza con AI
   -> Verifica leggibilità

5. INTEGRAZIONE (15 min)
   -> Collega componenti
   -> E2E testing manuale

TOTAL: ~1 ora per componente medio
```

3. Code Review con AI

Checklist: - [ ] Logica corretta? (Review manuale critico) - [ ] Edge cases gestiti? (Chiedi ad AI: “quali edge cases mancano?”) - [ ] Error handling robusto? (AI suggerisce scenari) - [ ] Tests completi? (AI genera test mancanti) - [ ] Performance accettabile? (AI identifica bottlenecks) - [ ] Security ok? (AI scan per vulnerabilità comuni) - [ ] Documentation chiara? (AI genera/migliora docs)

4. Testing Strategy

Pyramid:

```
        /\
       /E2\     E2E (5%): Full workflow tests
      /----\
     /INTEG\    Integration (15%): Component interaction
    /--------\
   /   UNIT   \ Unit (80%): Function/method tests
  /------------\
```

Con AI Agents: - **Unit**: AI genera la maggior parte - **Integration**: AI genera skeleton, human define scenarios - **E2E**: Human-written, AI assist con test data

5. Documentation

Auto-Generate: - JSDoc/docstrings per functions - README per ogni module - API documentation - Architecture diagrams (con Mermaid)

Human-Written: - High-level design decisions - Trade-off rationale - Complex algorithms explanation - Getting started guides

6. Git Workflow con AI

Commit Strategy:

```
# AI genera implementazione
git add src/new-component.ts tests/new-component.test.ts

# Human scrive commit message semantico
git commit -m "feat(memory): Add episodic memory with Postgres storage

- Implements EpisodicMemory class with CRUD operations
- Adds Postgres schema with migrations
- Includes comprehensive tests with fixtures
- Closes #123"

# AI review prima di push
git diff HEAD~1 | ai-review
```

Troubleshooting Guide

Issue: Agent Stuck in Loop

Symptoms: Agent ripete stesse azioni indefinitamente

Diagnosis:

```
// Check logs
grep "LOOP_DETECTION" logs/agent.log

// Check working memory
curl http://localhost:3000/debug/memory
```

Solutions: 1. Implementa loop detection nel Safety Verifier 2. Aggiungi max iterations limit (default: 50) 3. Migliora reflection per detect stuck patterns

Issue: High LLM Costs

Symptoms: Budget esaurito rapidamente

Diagnosis:

```
# Check metrics
curl http://localhost:3000/metrics | grep llm_cost

# Analyze per task type
npm run analyze-costs
```

Solutions: 1. Abilita aggressive caching 2. Usa model router per downgrade tasks semplici 3. Batch multiple LLM requests 4. Optimize prompts (riduci tokens)

Issue: Low Success Rate

Symptoms: <80% task success rate

Diagnosis:

```
# Check failure types
npm run analyze-failures

# Review failed episodes
curl http://localhost:3000/api/episodes?status=failed
```

Solutions: 1. Migliora error handling e retry logic 2. Aggiungi fallback strategies 3. Improve planning quality (più context) 4. Expand tool capabilities

Issue: Slow Performance

Symptoms: p95 latency >30s per task medio

Diagnosis:

```
# Profile performance
npm run profile

# Check traces
# Apri Jaeger: http://localhost:16686
```

Solutions: 1. Abilita parallel execution 2. Optimize slow tools (profiling) 3. Increase cache hit rate 4. Use faster models per simple tasks

Issue: Memory Leaks

Symptoms: Memory usage crescente, OOM crashes

Diagnosis:

```
# Monitor memory
node --inspect src/index.ts
# Apri chrome://inspect

# Heap snapshot
npm run heap-snapshot
```

Solutions: 1. Fix WorkingMemory eviction policy 2. Clear episodic memory old entries 3. Close database connections properly 4. Review event listener cleanup

---

[CHART] Metrics & Success Criteria

Key Performance Indicators

Latency: | Task Type | Target p50 | Target p95 | Target p99 | |———-|———-|———-|———-| | Simple Q&A | <1s | <3s | <5s | | Tool Execution | <3s | <8s | <15s | | Multi-step Planning | <5s | <20s | <40s | | Complex Research | <10s | <60s | <120s |

Reliability: - **Success Rate**: >90% overall, >95% per task type - **Error Rate**: <5% transient errors, <1% permanent errors - **Recovery Rate**: >95% auto-recovery da transient errors - **Availability**: >99.9% uptime

Cost Efficiency: - **Cost per Task**: <$0.10 average, <$0.50 p95 - **Cache Hit Rate**: >60% (target: 80%) - **Token Efficiency**: <5000 tokens per task average

Learning: - **Improvement Rate**: 20% faster dopo 100 episodes stesso task type - **Pattern Extraction**: >10 useful patterns dopo 1000 tasks - **Strategy Success**: >80% success rate per learned strategies

Monitoring Dashboard

Real-time Metrics (Grafana):

```
ROW 1: Overview
- Total Tasks (24h): Counter
- Success Rate: Gauge (target: >90%)
- Active Tasks: Gauge
- Error Rate: Gauge (alert: >5%)

ROW 2: Latency
- Task Duration: Histogram (p50/p95/p99)
- LLM Latency: Histogram
- Tool Latency: Histogram per tool

ROW 3: Cost
- Total Cost (24h): Counter
- Cost per Task: Histogram
- Cost by Model: Pie chart
- Budget Remaining: Gauge

ROW 4: Learning
- Episodes Stored: Counter
- Patterns Extracted: Counter
- Cache Hit Rate: Gauge
- Strategy Applications: Counter

ROW 5: Errors
- Errors by Type: Bar chart
- Failed Tasks Timeline: Time series
- Circuit Breaker Status: Gauge per service
```

Alerts:

```yaml
alerts:
  - name: HighErrorRate
    condition: error_rate > 0.05
    for: 5m
    severity: warning

  - name: LowSuccessRate
    condition: success_rate < 0.85
    for: 10m
    severity: critical

  - name: HighLatency
    condition: p95_latency > 60s
    for: 5m
    severity: warning

  - name: BudgetExhausted
    condition: budget_remaining < 0.1
    for: 1m
    severity: critical
```

Weekly Review Checklist

```markdown
## Week [N] Review

### Metrics
- [ ] Success rate: _____% (target: >90%)
- [ ] p95 latency: _____s (target: <30s)
- [ ] Total cost: $_____ (budget: $_____)
- [ ] Cache hit rate: _____% (target: >60%)

### Learning
- [ ] New patterns extracted: _____
- [ ] Strategies validated: _____
- [ ] Performance improvements: _____%

### Issues
- [ ] Critical bugs: _____
- [ ] Performance bottlenecks: _____
- [ ] User feedback: _____

### Next Week Focus
- [ ] Priority 1: _____
- [ ] Priority 2: _____
- [ ] Priority 3: _____
```

[TARGET] Conclusione

Questa roadmap fornisce un percorso strutturato per implementare un Reflective Adaptive Agent production-ready in 12-16 settimane.

Principi Chiave: 1. [OK] **Iterazione rapida**: MVP prima, ottimizzazione dopo 2. [OK] **Test-first**: Scrivi test prima di implementare 3. [OK] **Observable-first**: Strumentazione dal giorno 1 4. [OK] **AI-augmented**: Sfrutta agenti AI per velocizzare 5. [OK] **Human-guided**: Review critico umano sempre

Prossimi Passi: 1. Setup progetto (Fase 0) 2. Inizia con Core Loop (Fase 1) 3. Itera velocemente con feedback continuo 4. Misura tutto, ottimizza ciò che conta 5. Deploy in production con confidence

Risorse: - Reference Architecture - Framework Documentation - Italian Specifications

Supporto: - GitHub Issues per bug reports - Discussions per domande - Wiki per best practices condivise

---

Happy Building!