

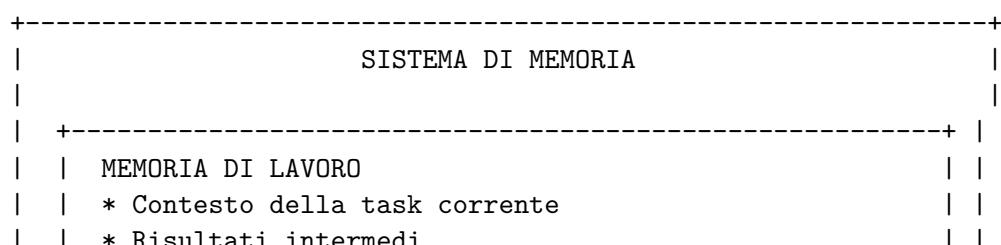
Contents

Sistema di Memoria: Archiviazione e Recupero della Conoscenza	1
Panoramica	1
1. Memoria di Lavoro	2
1.1 Scopo e Responsabilità	2
1.2 Struttura della Memoria di Lavoro	3
1.3 Strategie di Gestione del Contesto	4
1.4 Archiviazione Variabili	5
1.5 Operazioni della Memoria di Lavoro	7
2. Memoria Episodica	8
2.1 Scopo e Responsabilità	8
2.2 Struttura dell'Episodio	8
2.3 Architettura di Archiviazione	10
2.4 Strategie di Recupero	10
2.5 Gestione del Lifecycle degli Episodi	12
2.6 Operazioni della Memoria Episodica	14
3. Cache dei Pattern	15
3.1 Scopo e Responsabilità	15
3.2 Tassonomia dei Pattern	16
3.3 Struttura del Pattern	17
3.4 Archiviazione e Indicizzazione dei Pattern	19
3.5 Matching e Selezione dei Pattern	20
3.6 Validazione ed Evoluzione dei Pattern	22
3.7 Operazioni della Cache dei Pattern	24
4. Integrazione del Sistema di Memoria	25
4.1 Operazioni Cross-Memoria	25
4.2 Pipeline di Consolidamento Memoria	26
4.3 Caratteristiche di Performance	28
4.4 API del Sistema di Memoria	29

Sistema di Memoria: Archiviazione e Recupero della Conoscenza

Panoramica

Il Sistema di Memoria è la componente che permette all'agente di mantenere contesto, ricordare esperienze passate e accumulare conoscenza nel tempo. È strutturato in tre sottosistemi specializzati, ciascuno con caratteristiche e scopi diversi.



```

|   |   * Stato di reasoning attivo
|   |   Archiviazione: In-memory, ~20K token
|   |   Durata: Singola esecuzione di task
|   +-----+
|   |
|   +-----+
|   |   MEMORIA EPISODICA
|   |   * Esecuzioni storiche di task
|   |   * Tracce complete di esecuzione
|   |   * Risultati e apprendimenti
|   |   Archiviazione: Vector DB, milioni di episodi
|   |   Durata: Permanente (con invecchiamento)
|   +-----+
|   |
|   +-----+
|   |   CACHE DEI PATTERN
|   |   * Strategie apprese
|   |   * Euristiche validate
|   |   * Template riusabili
|   |   Archiviazione: Structured DB, migliaia di pattern
|   |   Durata: Permanente (con validazione)
|   +-----+
|
|   Flussi di Dati:
|   * Inizio Task -> Carica contesto in Memoria di Lavoro
|   * Durante Esecuzione -> Interroga Episodica per casi simili
|   * Durante Pianificazione -> Recupera Pattern da Cache
|   * Fine Task -> Memorizza episodio in Memoria Episodica
|   * Riflessione -> Aggiorna Cache dei Pattern
+-----+

```

1. Memoria di Lavoro

1.1 Scopo e Responsabilità

Funzione Principale: Mantenere contesto immediato necessario per reasoning ed execution della task corrente.

Caratteristiche: - **Volatile:** Esiste solo per durata di un'esecuzione - **Dimensione Limitata:** ~20K token max (vincolo context window LLM) - **Accesso Veloce:** In-memory, latenza <1ms - **Strutturato:** Organizzato per facilitare accesso

Responsabilità: 1. **Gestione Contesto:** Mantenere informazioni rilevanti per task corrente 2. **Tracciamento Stato:** Tracciare stato di esecuzione 3. **Caching Risultati:** Memorizzare output intermedi 4. **Archiviazione Variabili:** Gestire variabili temporanee 5. **Potatura Contesto:** Gestire limite di size, rimuovere info non più rilevanti

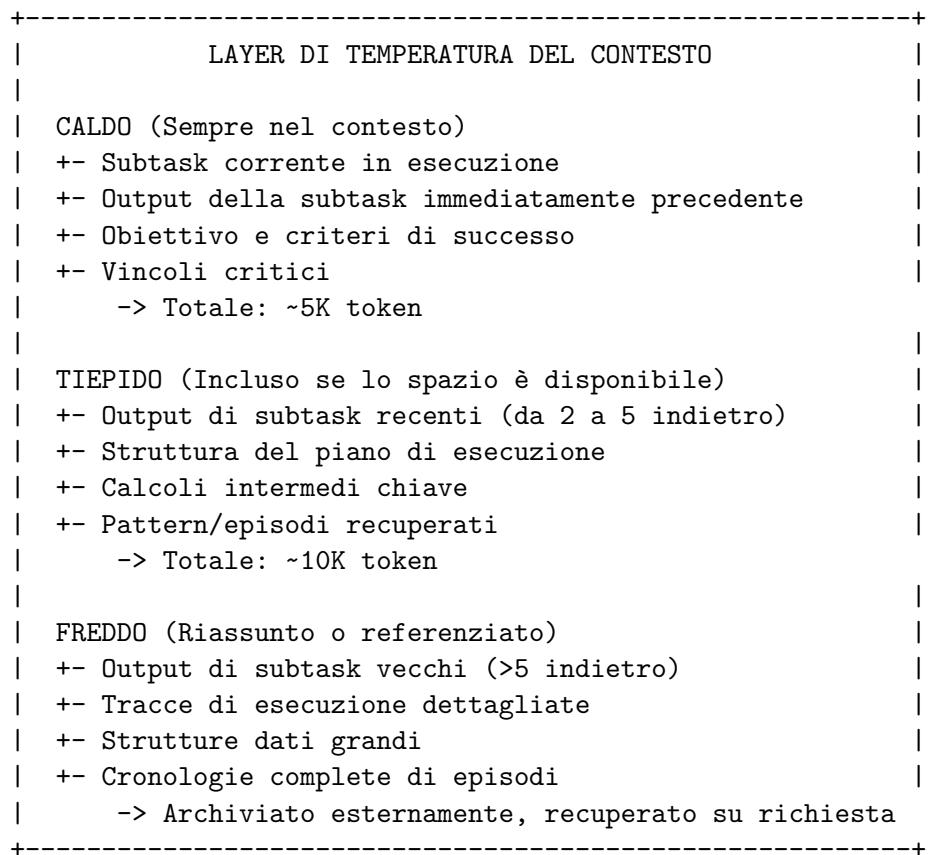
1.2 Struttura della Memoria di Lavoro

```
+-----+
|           LAYOUT DELLA MEMORIA DI LAVORO
|
| +-----+
| | CONTESTO DI SISTEMA (Statico, ~2K token)
| | * Capacità dell'agente
| | * Strumenti disponibili
| | * Configurazione
| | * Limiti di sicurezza
| +-----+
|
| +-----+
| | CONTESTO TASK (~5K token, Caldo)
| | * Descrizione originale della task
| | * Obiettivi e vincoli analizzati
| | * Criteri di successo
| | * Piano di esecuzione
| +-----+
|
| +-----+
| | STATO DI ESECUZIONE (~3K token, Dinamico)
| | * Fase corrente (analisi, pianificazione, esecuzione)
| | * Subtask completate
| | * Subtask in corso
| | * Subtask fallite con errori
| +-----+
|
| +-----+
| | RISULTATI INTERMEDI (~8K token, Tiepido)
| | * Output di subtask recenti (ultimi 5-7)
| | * Valori calcolati chiave
| | * Riferimenti a file/dati temporanei
| +-----+
|
| +-----+
| | CONTESTO RECUPERATO (~2K token, Freddo)
| | * Episodi rilevanti (riassunti)
| | * Pattern applicabili
| | * Frammenti di conoscenza di dominio
| +-----+
|
| | Totale: ~20K token (rientra in contesto LLM)
+-----+
```

1.3 Strategie di Gestione del Contesto

Sfida: Le context window degli LLM sono limitate, ma le task possono generare molto più contesto di quanto possa essere contenuto.

Soluzione: Gestione gerarchica del contesto con layer di “temperatura”.



Algoritmo di Potatura del Contesto:

```
Funzione GESTISCI_CONTESTO(memoria_di_lavoro, nuovo_contenuto):

    # Controlla se aggiungere nuovo contenuto supera il limite
    SE memoria_di_lavoro.dimensione + nuovo_contenuto.dimensione > LIMITE:

        # PASSO 1: Identifica ciò che può essere potato
        potabile = []

        # Candidato 1: Vecchi risultati intermedi
        risultati_vecchi = memoria_di_lavoro.risultati_intermedi.più_vecchi_di(5_subtask)
        PER OGNI risultato IN risultati_vecchi:
            SE NON risultato.è_referenziato_da_task_future:
                potabile.aggiungi(risultato)

        # Candidato 2: Tracce dettagliate (mantieni riassunti)
        tracce_dettagliate = memoria_di_lavoro.tracce_esecuzione
```

```

PER OGNI traccia IN tracce_dettagliate:
    riassunto = RIASSUMI(traccia)
    potabile.aggiungi({originale: traccia, sostituzione: riassunto})

# Candidato 3: Contesto recuperato non ancora usato
recuperato_non_usato = memoria_di_lavoro.contesto_recuperato.non_accesso
potabile.aggiungi(recuperato_non_usato)

# PASSO 2: Sposta in archiviazione esterna
PER OGNI elemento IN potabile:
    SE elemento.tipo == "risultato_intermedio":
        ARCHIVIA_IN_MEMORY_EPISODICA(elemento)
        memoria_di_lavoro.aggiungi_riferimento(elemento.id, "episodica")
    ALTRIMENTI SE elemento.tipo == "traccia":
        SOSTITUISCI(elemento.originale, elemento.sostituzione)

# PASSO 3: Ricalcola spazio
SE memoria_di_lavoro.dimensione + nuovo_contenuto.dimensione <= LIMITE:
    memoria_di_lavoro.aggiungi(nuovo_contenuto)
ALTRIMENTI:
    # Escalation: contesto veramente insufficiente
    AVVISA("Limite di contesto raggiunto nonostante la potatura")
    APPLICA_RIASSUNTO_AGGRESSIVO()

```

1.4 Archiviazione Variabili

Scopo: Mantenere variabili e valori computati durante esecuzione.

Tipi di Variabili:

TIPI DI VARIABILI	
1. VARIABILI TASK	
* Estratte dall'analisi dell'obiettivo	
* Esempio: file_target = "auth.py"	
* Durata: Intera task	
2. VALORI INTERMEDI	
* Risultati di subtask	
* Esempio: libreria_jwt = "PyJWT"	
* Durata: Fino a quando non più referenziato	
3. VARIABILI ACCUMULATORE	
* Aggregate attraverso subtask	
* Esempio: file_modificati = ["auth.py", "test.py"]	
* Durata: Intera task	
4. VARIABILI DI CONTROLLO DI FLUSSO	

```

|   * Contatori di loop, conteggi di retry
|   * Esempio: conteggio_retry = 2
|   * Durata: Limitato allo scope
|
| 5. VARIABILI METADATA
|   * Statistiche di esecuzione
|   * Esempio: token_usati = 15420
|   * Durata: Intera task
+-----+

```

Binding e Scope delle Variabili:

Schema di Archiviazione Variabili:

```
{
  "globale": {
    // Disponibile per tutta la task
    "task_id": "task_12345",
    "user_id": "user_789",
    "directory_lavoro": "/project/src"
  },

  "scope_task": {
    // Dall'analisi dell'obiettivo
    "obiettivo_principale": "Aggiungi autenticazione JWT",
    "moduli_target": ["auth.py", "middleware.py"],
    "vincoli": {...}
  },

  "scope_esecuzione": {
    // Stato di esecuzione corrente
    "fase_corrente": "implementazione",
    "subtask_corrente": "crea_encoder_jwt",
    "task_genitore": "implementa_jwt"
  },

  "valori_intermedi": {
    // Indicizzati per subtask_id
    "subtask_001": {
      "output": "PyJWT",
      "tipo": "scelta_libreria"
    },
    "subtask_002": {
      "output": {...},
      "tipo": "codice_generato"
    }
  },

  "metadata": {

```

```

    "ora_inizio": "2024-01-15T10:30:00Z",
    "token_consumati": 15420,
    "chiamate_llm": 8,
    "costo": 0.12
  }
}

```

1.5 Operazioni della Memoria di Lavoro

Operazioni Principali:

1. INIZIALIZZA()
 - * Inizializza memoria di lavoro vuota per nuova task
 - * Carica contesto di sistema
 - * Alloca spazio iniziale
2. CARICA_CONTESTO_TASK(task)
 - * Analizza task in rappresentazione strutturata
 - * Memorizza obiettivi, vincoli, contesto
 - * Alloca variabili task
3. IMPOSTA(chiave, valore, scope)
 - * Memorizza valore nello scope appropriato
 - * Aggiorna tracciamento dimensione
 - * Attiva potatura se necessario
4. OTTIENI(chiave, scope)
 - * Recupera valore dallo scope
 - * Marca come accesso (per potatura)
 - * Restituisce valore o null
5. INSERISCI_STATO(stato)
 - * Salva stato di esecuzione corrente
 - * Aggiorna fase, progressi
 - * Registra timestamp
6. AGGIUNGI_INTERMEDIO(subtask_id, output)
 - * Memorizza output della subtask
 - * Aggiorna tracciamento recenza
 - * Può attivare potatura
7. RECUPERA_DA_EPISODICA(query)
 - * Interroga memoria episodica
 - * Carica episodi rilevanti in memoria di lavoro
 - * Riassume se necessario per adattarsi
8. POTA()
 - * Identifica contenuto freddo

- * Sposta in archiviazione esterna
 - * Mantieni riferimenti
9. SNAPSHOT()
 - * Crea checkpoint dello stato corrente
 - * Per debugging, rollback
 10. PULISCI()
 - * Resetta memoria di lavoro dopo completamento task
 - * Mantieni solo metadata per riflessione

2. Memoria Episodica

2.1 Scopo e Responsabilità

Funzione Principale: Archiviazione permanente di tutti gli episodi di esecuzione per learning e recupero futuro.

Caratteristiche: - **Persistente:** Dati sopravvivono tra sessioni - **Larga Scala:** Milioni di episodi - **Ricercabile:** Ricerca semantica e strutturale - **Solo Aggiunte:** Episodi non modificati dopo creazione (immutabili) - **Indicizzato:** Indicizzazione multi-dimensionale per recupero veloce

Responsabilità: 1. **Archiviazione Episodi:** Persistere tracce di esecuzione complete 2. **Ricerca Semantica:** Trovare episodi simili a query 3. **Query Strutturata:** Query su attributi specifici 4. **Invecchiamento e Archiviazione:** Gestire lifecycle di episodi vecchi 5. **Ottimizzazione Recupero:** Accesso veloce a episodi rilevanti

2.2 Struttura dell'Episodio

Schema Completo dell'Episodio:

```
Episodio {
    // Identificazione
    episodio_id: stringa (UUID),
    timestamp: datetime,
    sessione_id: stringa,

    // Descrizione Task
    task: {
        input_originale: stringa,
        obiettivi_analizzati: StrutturaObiettivo,
        vincoli: [Vincolo],
        contesto: Contesto,
        complessità: ValutazioneComplessità
    },
    // Traccia di Esecuzione
    esecuzione: {
```

```

piano: PianoEsecuzione,
traccia: [VoceTraccia],
adattamenti: [Adattamento],
durata: float,
uso_risorse: MetricheRisorse
},

// Risultato
risultato: {
    stato: "SUCCESSO" | "PARZIALE" | "FALLIMENTO",
    output_primario: {...},
    effetti_collaterali: [EffettoCollaterale],
    metriche_qualità: Metriche
},

// Apprendimento
riflessione: {
    intuizioni: [Intuizione],
    pattern_scoperti: [Pattern],
    fallimenti_analizzati: [Fallimento],
    analisi_performance: Analisi
},

// Metadata per Recupero
metadata: {
    dominio: stringa,
    tipo_task: stringa,
    strumenti_usati: [stringa],
    strategie_applicate: [stringa],
    tag: [stringa]
},

// Embedding per Ricerca Semantica
embedding: {
    embedding_task: vettore(768),
    embedding_risultato: vettore(768),
    embedding_episodio_completo: vettore(768)
},

// Lifecycle
lifecycle: {
    conteggio_accessi: int,
    ultimo_accesso: datetime,
    referenziato_da: [episodio_id],
    stato_archiviazione: "attivo" | "archiviato" | "eliminato"
}
}

```

2.3 Architettura di Archiviazione

+-----+ ARCHIVIAZIONE MEMORIA EPISODICA +-----+ VECTOR DATABASE (Ricerca Semantica) * Memorizza embedding degli episodi * Ricerca di similarità veloce (ANN) * Tecnologia: Pinecone, Weaviate, o Milvus * Indice: vettori ~768-dimensionalni * Tempo di query: <100ms per top-K +-----+ +-----+ DOCUMENT DATABASE (Archiviazione Strutturata) * Memorizza documenti episodi completi * Indicizzato per episodio_id * Tecnologia: MongoDB, PostgreSQL con JSONB * Indici: tipo_task, dominio, timestamp, stato * Tempo query: <50ms per ID, <200ms per attributi +-----+ +-----+ OBJECT STORAGE (Artefatti Grandi) * Memorizza output grandi, tracce * Referenziati dall'episodio, non embedded * Tecnologia: S3, MinIO * Esempio: File generati grandi, immagini, log +-----+ +-----+ CACHE LAYER (Episodi Caldi) * Cache in-memory di episodi accessi frequentemente * Tecnologia: Redis * Dimensione: Ultimi 1000 episodi o più accessi * Tasso di hit target: >70% +-----+ +-----+	
--	--

2.4 Strategie di Recupero

Recupero Multi-Modale: Combina diversi approcci di recupero per i migliori risultati.

+-----+ STRATEGIE DI RECUPERO 1. RICERCA DI SIMILARITÀ SEMANTICA +-----+	
---	--

```

| Input: Descrizione della task corrente | 
| Metodo: Similarità vettoriale (coseno, prodotto dot) | 
| Output: Top-K episodi più simili | 
| Uso: Trova task passate simili | 

| 2. QUERY STRUTTURATA | 
| Input: Filtri attributi (dominio, tipo_task, ecc) | 
| Metodo: Query database | 
| Output: Episodi che corrispondono ai criteri | 
| Uso: Trova tutti gli episodi di tipo specifico | 

| 3. RICERCA IBRIDA | 
| Input: Descrizione task + filtri attributi | 
| Metodo: Combina semantica + strutturata | 
| Output: Episodi rilevanti che soddisfano criteri | 
| Uso: "Simile a X nel dominio Y" | 

| 4. RICERCA TEMPORALE | 
| Input: Intervallo temporale | 
| Metodo: Indicizzazione basata su tempo | 
| Output: Episodi recenti | 
| Uso: "Cosa ho fatto in questa sessione?" | 

| 5. RICERCA SPECIFICA PER FALLIMENTI | 
| Input: Firma del fallimento corrente | 
| Metodo: Confronta pattern di fallimento | 
| Output: Episodi con fallimenti simili | 
| Uso: Impara dai fallimenti passati | 
+-----+

```

Algoritmo di Recupero:

Funzione RECUPERA_EPISODI_RILEVANTI(query, contesto, k=5):

```

# PASSO 1: Ricerca semantica
embedding_query = EMBEDDING(query)
corrispondenze_semantiche = RICERCA_VETTORIALE(embedding_query, top_k=20)

# PASSO 2: Applica filtri contestuali
filtrato = []
PER OGNI episodio IN corrispondenze_semantiche:
    # Filtra per dominio se specificato
    SE contesto.dominio E episodio.dominio != contesto.dominio:
        CONTINUA

    # Filtra per recenza se preferito
    SE contesto.preferisci_recenti E episodio.età > SOGLIA:
        CONTINUA

```

```

# Filtra per successo se impariamo da episodi riusciti
SE contesto.solo_riusciti E episodio.risultato != SUCCESSO:
    CONTINUA

filtrato.aggiungi(episodio)

# PASSO 3: Ri-ordina per rilevanza
con_punteggio = []
PER OGNI episodio IN filtrato:
    punteggio = CALCOLA_PUNTEGGIO_RILEVANZA(episodio, query, contesto)
    con_punteggio.aggiungi((episodio, punteggio))

ordinato = ORDINA_PER_PUNTEGGIO(con_punteggio, decrescente=Vero)

# PASSO 4: Restituisci top-K
RESTITUISCI ordinato[:k]

Funzione CALCOLA_PUNTEGGIO_RILEVANZA(episodio, query, contesto):
    punteggio = 0

    # Similarità semantica (primaria)
    punteggio += 0.5 * SIMILARITÀ_COSEN0(query.embedding, episodio.embedding)

    # Bonus recenza
    età_giorni = (ora - episodio.timestamp).giorni
    fattore_recenza = EXP(-età_giorni / 30) # Decadimento su ~mese
    punteggio += 0.2 * fattore_recenza

    # Bonus successo
    SE episodio.risultato == SUCCESSO:
        punteggio += 0.15

    # Bonus frequenza accesso (episodi popolari probabilmente utili)
    fattore_accesso = LOG(1 + episodio.conteggio_accessi) / 10
    punteggio += 0.1 * MIN(fattore_accesso, 1.0)

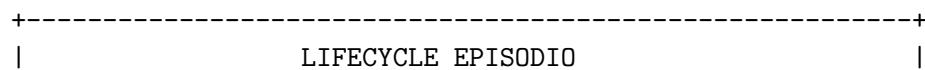
    # Bonus corrispondenza dominio
    SE episodio.dominio == contesto.dominio:
        punteggio += 0.05

RESTITUISCI punteggio

```

2.5 Gestione del Lifecycle degli Episodi

Politica di Invecchiamento: Vecchi episodi meno rilevanti nel tempo.



FASE 1: ATTIVO (0-30 giorni)	
* Completamente indicizzato e ricercabile	
* Alta priorità di recupero	
* Nessuna degradazione	
FASE 2: MATURO (30-180 giorni)	
* Ancora completamente ricercabile	
* Priorità di recupero più bassa vs recenti	
* Eleggibile per consolidamento con episodi simili	
FASE 3: INVECHIATO (180-365 giorni)	
* Tracce dettagliate archiviate in archiviazione fredda	
* Riassunti rimangono ricercabili	
* Recuperato solo se altamente rilevante	
FASE 4: ARCHIVIATO (>365 giorni)	
* Solo metadata e apprendimenti chiave mantenuti	
* Episodio completo in archiviazione fredda	
* Recuperato solo su richiesta esplicita	
FASE 5: ELIMINATO (Raro)	
* Episodi senza valore (duplicati, errori)	
* Esplicitamente contrassegnati per eliminazione	
* Eliminazione soft con periodo di ritenzione	

Processo di Consolidamento: Unisci episodi simili per risparmiare archiviazione.

Algoritmo di Consolidamento:

1. IDENTIFICA CLUSTER
 - * Raggruppa episodi per alta similarità (>0.95 coseno)
 - * Stesso tipo task, dominio, risultato
 - * Entro breve finestra temporale
2. ANALIZZA CLUSTER
 - * Estrai struttura comune
 - * Identifica variazioni
 - * Calcola statistiche aggregate
3. CREA EPISODIO CONSOLIDATO


```
EpisodioConsolidato {
    episodio_rappresentativo_id: stringa,
    num_istanze: int,
    struttura_comune: {...},
    variazioni: [{episodio_id, diff}],
    statistiche_aggregate: {
        durata_media: float,
```

```

        tasso_successo: float,
        pattern_comuni: [Pattern]
    }
}

```

4. ARCHIVIA EPISODI ORIGINALI
 - * Marca originali come consolidati
 - * Mantieni riferimenti per verificabilità
 - * Libera archiviazione primaria
5. AGGIORNA INDICI
 - * Episodio consolidato ottiene embedding combinato
 - * Mantiene ricercabilità

2.6 Operazioni della Memoria Episodica

1. MEMORIZZA_EPISODIO(episodio)
 - * Valida struttura episodio
 - * Genera embedding
 - * Inserisci in vector DB
 - * Inserisci in document DB
 - * Aggiorna indici
 - * Restituisci episodio_id
2. RECUPERA_PER_ID(episodio_id)
 - * Controlla cache
 - * Se miss, interroga document DB
 - * Se archiviato, recupera da archiviazione fredda
 - * Aggiorna metadata di accesso
 - * Restituisci episodio
3. RICERCA_SEMANTICA(query, k, filtri)
 - * Embedding della query
 - * Ricerca vettoriale con filtri
 - * Recupera episodi completi per top-K
 - * Ordina per rilevanza
 - * Restituisci risultati ordinati
4. RICERCA_STRUTTURATA(criteri)
 - * Costruisci query database
 - * Esegui su document DB
 - * Applica limiti
 - * Restituisci episodi corrispondenti
5. OTTIENI_RECENTI(n, filtri)
 - * Query per timestamp desc
 - * Applica filtri
 - * Limita a n

- * Restituisci episodi
6. OTTIENI_FALLIMENTI_SIMILI(firma_fallimento, k)
 - * Estrai caratteristiche fallimento
 - * Cerca pattern corrispondenti
 - * Restituisci episodi con fallimenti simili
 7. AGGIORNA_METADATA_ACCESSO(episodio_id)
 - * Incrementa conteggio_accessi
 - * Aggiorna ultimo_accesso
 - * Aggiorna cache
 8. ARCHIVIA_EPISODI_VECCHI(soglia_eta)
 - * Identifica episodi più vecchi della soglia
 - * Sposta tracce dettagliate in archiviazione fredda
 - * Aggiorna stato_archiviazione
 - * Mantieni riassunti ricercabili
 9. CONSOLIDA_SIMILI(soglia_similarita)
 - * Esegui clustering
 - * Identifica candidati per consolidamento
 - * Crea episodi consolidati
 - * Archivia originali
 10. ELIMINA_CANCELLATI(periodo_ritenzione)
 - * Elimina permanentemente episodi marcati per eliminazione
 - * Dopo scadenza periodo di ritenzione

3. Cache dei Pattern

3.1 Scopo e Responsabilità

Funzione Principale: Archiviare strategie, euristiche e pattern validati che possono essere riusati per migliorare performance e decision-making.

Caratteristiche: - **Curato:** Solo pattern validati con sufficiente evidenza - **Strutturato:** Organizzati per facilitare matching e applicazione - **Evolvibile:** Pattern aggiornati quando nuova evidenza emerge - **Azionabile:** Direttamente applicabili in pianificazione/esecuzione

Differenza dalla Memoria Episodica:

MEMORIA EPISODICA:

- * Esperienze grezze (cosa è accaduto)
- * Tracce complete
- * Milioni di istanze
- * Immutabile dopo creazione

CACHE DEI PATTERN:

- * Conoscenza distillata (lezioni apprese)
- * Strategie generalizzate
- * Migliaia di pattern
- * Aggiornato con nuova evidenza

3.2 Tassonomia dei Pattern

CATEGORIE DI PATTERN	
1. PATTERN STRATEGICI	
* Approcci di alto livello per classi di problemi	
* Quando usare quale strategia di pianificazione	
* Esempio: "HTN planning migliore per refactoring"	
* Applicabilità: Classificazione problema -> Strategia	
2. PATTERN DI DECOMPOSIZIONE	
* Come scomporre tipi specifici di task	
* Strutture template di subtask	
* Esempio: "Feature API -> progetta, implementa, testa"	
* Applicabilità: Tipo task -> Template decomposizione	
3. PATTERN DI SEQUENZA	
* Passi ordinati che funzionano bene insieme	
* Relazioni di dipendenza	
* Esempio: "Installa -> Importa -> Verifica -> Testa"	
* Applicabilità: Tipo subtask -> Passi successivi	
4. PATTERN DI SELEZIONE STRUMENTI	
* Quali strumenti funzionano meglio per quali task	
* Strategie di combinazione strumenti	
* Esempio: "Refactoring Python -> usa ast + black"	
* Applicabilità: Caratteristiche task -> Scelte strumenti	
5. PATTERN DI FALLIMENTO	
* Modalità comuni di fallimento e cause	
* Come riconoscere ed evitare	
* Esempio: "Install pacchetto fallisce senza pip upgrade"	
* Applicabilità: Segnale fallimento -> Diagnosi	

```

+-----+
| |
| +-----+
| | 6. PATTERN DI RECUPERO
| | * Strategie di contingenza efficaci
| | * Sequenze di recupero
| | * Esempio: "Timeout API -> retry con backoff"
| | * Applicabilità: Tipo fallimento -> Azione recupero
| +-----+
|
| +-----+
| | 7. PATTERN DI OTTIMIZZAZIONE
| | * Opportunità di miglioramento performance
| | * Ottimizzazioni uso risorse
| | * Esempio: "Task indipendenti -> parallelizza"
| | * Applicabilità: Struttura codice -> Ottimizzazione
| +-----+
|
| +-----+
| | 8. PATTERN DI DOMINIO
| | * Best practice specifiche del dominio
| | * Convenzioni e idiom
| | * Esempio: "FastAPI -> middleware per cross-cutting"
| | * Applicabilità: Dominio + obiettivo -> Approccio
| +-----+
+-----+

```

3.3 Struttura del Pattern

Schema Completo del Pattern:

```

Pattern {
    // Identificazione
    pattern_id: stringa (UUID),
    nome: stringa,
    categoria: CategoriaPattern,
    versione: int,
    creato_il: datetime,
    ultimo_aggiornamento: datetime,

    // Descrizione
    descrizione: stringa,
    riassunto: stringa (1-2 frasi),
    spiegazione_dettagliata: stringa,

    // Applicabilità
    applicabilità: {
        tipi_problema: [stringa],

```

```

caratteristiche_task: {
    dominio: [stringa],
    complessità: IntervalloComplessità,
    capacità_richieste: [stringa]
},
condizioni_contesto: [Condizione],
anti_condizioni: [Condizione] // Quando NON applicare
},

// Contenuto Pattern
contenuto: {
    tipo: "STRATEGIA" | "TEMPLATE" | "EURISTICA" | "PROCEDURA",
}

// Per pattern Strategia
strategia: {
    approccio: stringa,
    principi_chiave: [stringa],
    trade_off: stringa
},

// Per pattern Template
template: {
    struttura: {...}, // Template decomposizione subtask
    parametri: [Parametro],
    regole_instanziazione: [Regola]
},

// Per pattern Euristica
euristica: {
    condizione: Condizione,
    azione: Azione,
    razionale: stringa
},

// Per pattern Procedura
procedura: {
    passi: [Passo],
    logica_branching: {...},
    condizioni_terminazione: [Condizione]
}
},

// Evidenza & Validazione
evidenza: {
    // Episodi che supportano questo pattern
    episodi_supporto: [episodio_id],
    episodi_contrari: [episodio_id],
}

```

```

// Statistiche
num_applicazioni: int,
tasso_successo: float,
risparmio_tempo_medio: float,
risparmio_costo_medio: float,
miglioramento_tasso_successo: float,

// Confidenza
punteggio_confidenza: float, // 0-1
significatività_statistica: float, // p-value
ultima_validazione: datetime
},

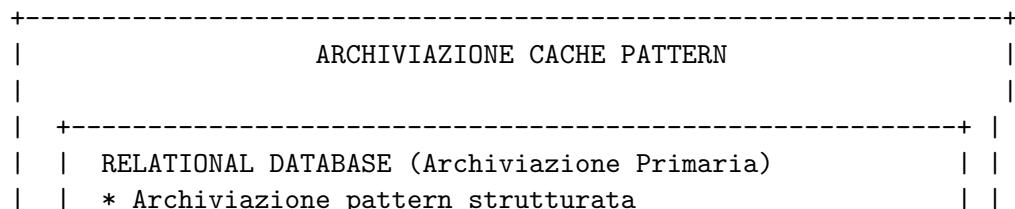
// Guida Applicazione
applicazione: {
    quando_applicare: stringa,
    come_applicare: stringa,
    benefici_attesi: [stringa],
    rischi_potenziali: [stringa],
    pattern_alternativi: [pattern_id]
},

// Lifecycle
lifecycle: {
    stato: "CANDIDATO" | "VALIDATO" | "DEPRECATO",
    conteggio_uso: int,
    ultimo_uso: datetime,
    sostituito_da: pattern_id | null,
    motivo_deprecazione: stringa | null
},

// Relazioni
relazioni: {
    specializza: pattern_id | null, // Versione più specifica di
    generalizza: [pattern_id], // Versioni più generali
    contraddice: [pattern_id], // Pattern incompatibili
    complementa: [pattern_id] // Funziona bene con
}
}

```

3.4 Archiviazione e Indicizzazione dei Pattern



```

|   |   * Query complesse su attributi           |
|   |   * Tecnologia: PostgreSQL                 |
|   |   * Indici: categoria, applicabilità, confidenza |
|   +-----+
|       |
|   +-----+
|   |   INVERTED INDEX (Lookup basato su caratteristiche)   |
|   |   * Mappa caratteristiche task -> pattern applicabili |
|   |   * Matching veloce dei pattern                   |
|   |   * Esempio: dominio="python" -> [pattern_ids]      |
|   +-----+
|       |
|   +-----+
|   |   DECISION TREE INDEX (Matching condizionale)    |
|   |   * Struttura ad albero per selezione pattern   |
|   |   * Naviga rispondendo a domande                |
|   |   * Esempio: È task codice? -> Sì -> Python? -> ... |
|   +-----+
|       |
|   +-----+
|   |   IN-MEMORY CACHE (Pattern caldi)          |
|   |   * Pattern usati più frequentemente        |
|   |   * Tecnologia: Redis                      |
|   |   * Dimensione: Top 100-200 pattern         |
|   +-----+
+-----+

```

3.5 Matching e Selezione dei Pattern

Algoritmo di Matching dei Pattern:

Funzione TROVA_PATTERN_APPLICABILI(task, contesto):

```

# PASSO 1: Estrai caratteristiche task
caratteristiche = ESTRAI_CARATTERISTICHE(task, contesto)
# Caratteristiche: {dominio, tipo_task, complessità, strumenti_richiesti, ...}

# PASSO 2: Query per categoria (se nota)
SE contesto.suggerimento_categoria_pattern:
    candidati = QUERY_PATTERN_PER_CATEGORIA(contesto.suggerimento_categoria_pattern)
ALTRIMENTI:
    # Ottieni tutti i pattern attivi
    candidati = QUERY_PATTERN_PER_STATO("VALIDATO")

# PASSO 3: Filtra per applicabilità
applicabili = []
PER OGNI pattern IN candidati:

```

```

# Controlla corrispondenza tipo problema
SE task.tipo NON IN pattern.applicabilità.tipi_problema:
    CONTINUA

# Controlla requisiti caratteristiche
SE NON CORRISPONDE(caratteristiche, pattern.applicabilità.caratteristiche_task):
    CONTINUA

# Controlla condizioni contesto
SE NON VALUTA_CONDIZIONI(pattern.applicabilità.condizioni_contesto, contesto):
    CONTINUA

# Controlla anti-condizioni (quando NON applicare)
SE VALUTA_CONDIZIONI(pattern.applicabilità.anti_condizioni, contesto):
    CONTINUA

applicabili.aggiungi(pattern)

# PASSO 4: Ordina per confidenza ed evidenza
ordinato = ORDINA_PATTERN(applicabili, task, contesto)

# PASSO 5: Restituisci top-K
RESTITUISCI ordinato[:5]

Funzione ORDINA_PATTERN(pattern, task, contesto):
    con_punteggio = []

    PER OGNI pattern IN pattern:
        punteggio = 0

        # Punteggio base: confidenza
        punteggio += 0.4 * pattern.evidenza.punteggio_confidenza

        # Bonus tasso successo
        punteggio += 0.3 * pattern.evidenza.tasso_successo

        # Frequenza uso (pattern popolari probabilmente buoni)
        fattore_uso = LOG(1 + pattern.lifecycle.conteggio_uso) / 10
        punteggio += 0.15 * MIN(fattore_uso, 1.0)

        # Bonus recenza (pattern validati recentemente preferiti)
        giorni_da_validazione = (ora - pattern.evidenza.ultima_validazione).giorni
        recenza = EXP(-giorni_da_validazione / 90)
        punteggio += 0.1 * recenza

        # Qualità corrispondenza caratteristiche
        qualità_match = CALCOLA_MATCH_CARATTERISTICHE(task, pattern.applicabilità)
        punteggio += 0.05 * qualità_match

```

```

con_punteggio.aggiungi((pattern, punteggio))

RESTITUISCI ORDINA_PER_PUNTEGGIO(con_punteggio, decrescente=Vero)

```

3.6 Validazione ed Evoluzione dei Pattern

Lifecycle dei Pattern:

LIFECYCLE DEI PATTERN	
FASE 1: CANDIDATO	<ul style="list-style-type: none"> * Appena scoperto dalla riflessione * Evidenza insufficiente (<5 episodi supporto) * Non ancora usato nella pianificazione * Tracciato per accumulo evidenza <p> (Quando evidenza >= 5 episodi, confidenza > 0.7)</p>
FASE 2: VALIDATO	<ul style="list-style-type: none"> * Evidenza sufficiente raccolta * Significatività statistica raggiunta * Usato in pianificazione ed esecuzione * Monitorato continuamente per performance <p> (Se emerge evidenza contraddittoria)</p>
FASE 3: IN REVISIONE	<ul style="list-style-type: none"> * Nuova evidenza contraddice il pattern * Tasso successo calato significativamente * Temporaneamente disabilitato dall'uso * Processo di ri-validazione attivato <p> (Rami basati su esito revisione)</p>
ESITO A: RAFFINATO (Torna a VALIDATO)	<ul style="list-style-type: none"> * Pattern aggiornato con nuove intuizioni * Condizioni applicabilità raffinate * Evidenza resettata, inizia ri-accumulazione
ESITO B: DEPRECATO	<ul style="list-style-type: none"> * Pattern non più valido * Sostituito da pattern migliore * Mantenuto per registro storico * Non usato in nuove esecuzioni

Processo di Validazione:

Funzione VALIDA_PATTERN(pattern):

```
# Raccogli applicazioni recenti
episodi_recenti = OTTIENI_EPISODI_CHE_USANO_PATTERN(
    pattern.pattern_id,
    da=ora - 90_giorni
)

SE lunghezza(episodi_recenti) < SOGLIA_MIN_EVIDENZA:
    RESTITUISCI "EVIDENZA_INSUFFICIENTE"

# Calcola tasso successo
successi = CONTA(episodio per episodio in episodi_recenti
                  se episodio.risultato == SUCCESSO)
tasso_successo = successi / lunghezza(episodi_recenti)

# Test significatività statistica
p_value = TEST_BINOMIALE(
    successi,
    lunghezza(episodi_recenti),
    tasso_atteso=0.5 # Ipotesi nulla: non meglio del casuale
)

# Aggiorna evidenza pattern
pattern.evidenza.tasso_successo = tasso_successo
pattern.evidenza.significatività_statistica = p_value
pattern.evidenza.num_applicazioni = lunghezza(episodi_recenti)
pattern.evidenza.ultima_validazione = ora

# Decisione
SE tasso_successo >= SOGLIA_SUCCESSO E p_value < 0.05:
    # Calcola confidenza basata su forza evidenza
    confidenza = CALCOLA_CONFIDENZA(tasso_successo, lunghezza(episodi_recenti), p_value)
    pattern.evidenza.punteggio_confidenza = confidenza

    SE confidenza >= SOGLIA_CONFIDENZA:
        pattern.lifecycle.stato = "VALIDATO"
        RESTITUISCI "VALIDATO"
    ALTRIMENTI:
        pattern.lifecycle.stato = "CANDIDATO"
        RESTITUISCI "CANDIDATO"
    ALTRIMENTI:
        pattern.lifecycle.stato = "IN_REVISIONE"
        RESTITUISCI "NECESSITA_REVISIONE"

Funzione CALCOLA_CONFIDENZA(tasso_successo, dimensione_campione, p_value):
    # La confidenza aumenta con:
    # 1. Alto tasso successo
```

```

# 2. Grande dimensione campione
# 3. Forte significatività statistica

componente_successo = tasso_successo

# Confidenza dimensione campione (asintotica a 1)
componente_campione = 1 - EXP(-dimensione_campione / 20)

# Componente significatività statistica
componente_significatività = 1 - p_value

# Combinazione pesata
confidenza =
  0.5 * componente_successo +
  0.3 * componente_campione +
  0.2 * componente_significatività
)

RESTITUISCI CLIP(confidenza, 0, 1)

```

3.7 Operazioni della Cache dei Pattern

1. AGGIUNGI_PATTERN(pattern)
 - * Valida struttura
 - * Imposta stato iniziale a CANDIDATO
 - * Inserisci nel database
 - * Crea indici
 - * Restituisci pattern_id

2. OTTIENI_PATTERN(pattern_id)
 - * Controlla cache
 - * Se miss, interroga database
 - * Restituisci pattern

3. TROVA_APPLICABILI(task, contesto)
 - * Estrai caratteristiche
 - * Interroga pattern indicizzati
 - * Filtra per applicabilità
 - * Ordina per rilevanza
 - * Restituisci top-K

4. AGGIORNA_PATTERN(pattern_id, aggiornamenti)
 - * Recupera pattern corrente
 - * Applica aggiornamenti
 - * Incrementa versione
 - * Aggiorna timestamp
 - * Persisti modifiche

5. REGISTRA_APPLICAZIONE(pattern_id, episodio_id, risultato)
 - * Collega pattern a episodio
 - * Aggiorna statistiche uso
 - * Aggiungi a evidenza
 - * Aggiorna cache

6. VALIDA_PATTERN()
 - * Per ogni pattern VALIDATO:
 - Raccogli evidenza recente
 - Ricalcola statistiche
 - Aggiorna confidenza
 - Aggiusta stato se necessario

7. DEPRECARE_PATTERN(pattern_id, motivo, sostituito_da)
 - * Aggiorna stato a DEPRECATO
 - * Registra motivo
 - * Collega a pattern sostitutivo se esiste
 - * Rimuovi da indici attivi

8. OTTIENI_PATTERN_SIMILI(pattern)
 - * Confronta struttura
 - * Confronta applicabilità
 - * Restituisci pattern simili
 - * Usa per deduplicazione

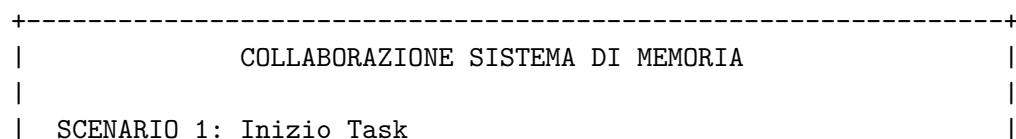
9. UNISCI_PATTERN(pattern_ids)
 - * Combina evidenza
 - * Generalizza applicabilità
 - * Crea pattern unito
 - * Depreca originali

10. POTA_PATTERN_BASSO_VALORE()
 - * Identifica pattern con:
 - Bassa confidenza
 - Basso uso
 - Evidenza contraddittoria
 - * Marca per revisione o eliminazione

4. Integrazione del Sistema di Memoria

4.1 Operazioni Cross-Memoria

I sistemi di memoria cooperano per fornire supporto completo:



```

| +-----+ |
| | 1. Analisi Obiettivo estrae caratteristiche task | | |
| | | |
| | | 2. Interroga CACHE PATTERN per strategie applicabili | |
| | | -> Carica top-3 pattern in MEMORIA DI LAVORO | |
| | | | |
| | | 3. Interroga MEMORIA EPISODICA per task passate simili | |
| | | -> Carica top-5 episodi (riassunti) in MEM LAVORO | |
| | | | |
| | | 4. Motore di Pianificazione usa pattern ed episodi | |
| +-----+ |

| |
| SCENARIO 2: Durante Esecuzione (Fallimento)
| +-----+
| | 1. Task fallisce con firma errore | | |
| | | |
| | | 2. Interroga CACHE PATTERN per pattern recupero | |
| | | -> Carica in MEMORIA DI LAVORO | |
| | | | |
| | | 3. Interroga MEMORIA EPISODICA per fallimenti simili | |
| | | -> Impara come fallimenti passati furono risolti | |
| | | | |
| | | 4. Motore di Adattamento applica recupero appreso | |
| +-----+ |

| |
| SCENARIO 3: Post-Esecuzione (Riflessione)
| +-----+
| | 1. Task completa, traccia esecuzione in MEM DI LAVORO | | |
| | | |
| | | 2. Memorizza episodio completo in MEMORIA EPISODICA | |
| | | | |
| | | 3. Riflessione analizza episodio | |
| | | | |
| | | 4. Estrai pattern -> Aggiorna CACHE PATTERN | |
| | | * Nuovi pattern aggiunti come CANDIDATI | |
| | | * Pattern esistenti aggiornati con nuova evidenza | |
| | | | |
| | | 5. MEMORIA DI LAVORO pulita | |
| +-----+ |
+-----+

```

4.2 Pipeline di Consolidamento Memoria

Processo periodico in background per mantenere qualità memoria:

```

+-----+
| PIPELINE DI CONSOLIDAMENTO MEMORIA | |
| (Eseguita notturna o settimanale) | |
+-----+

```

```

| STAGE 1: CONSOLIDAMENTO MEMORIA EPISODICA |
+-----+
|   * Identifica episodi altamente simili (clustering)   |
|   * Unisci episodi simili in record consolidati      |
|   * Archivia vecchie tracce dettagliate in storage freddo |
|   * Aggiorna stati lifecycle episodi                   |
+-----+
|
| STAGE 2: ESTRAZIONE PATTERN |
+-----+
|   * Analizza episodi recenti per nuovi pattern        |
|   * Esegui algoritmi di pattern mining                 |
|   * Crea pattern CANDIDATI                            |
|   * Aggiungi a Cache Pattern per validazione       |
+-----+
|
| STAGE 3: VALIDAZIONE PATTERN |
+-----+
|   * Per ogni pattern VALIDATO:
|     - Raccogli evidenza applicazione recente          |
|     - Ricalcola statistiche                          |
|     - Aggiorna punteggi confidenza                  |
|   * Per ogni pattern CANDIDATO:
|     - Controlla se evidenza sufficiente accumulata |
|     - Promuovi a VALIDATO se criteri soddisfatti  |
+-----+
|
| STAGE 4: RAFFINAMENTO PATTERN |
+-----+
|   * Identifica pattern con performance in calo      |
|   * Analizza casi fallimento                         |
|   * Raffina condizioni applicabilità                |
|   * Unisci pattern simili                           |
|   * Depreca pattern obsoleti                        |
+-----+
|
| STAGE 5: OTTIMIZZAZIONE INDICI |
+-----+
|   * Ricostruisci indici vettoriali per memoria episodica |
|   * Aggiorna indici invertiti per cache pattern           |
|   * Ottimizza piani query database                      |
|   * Aggiorna cache con elementi caldi                  |
+-----+
|
| STAGE 6: REPORTING |
+-----+
|   * Genera report salute sistema memoria            |

```

```

|   |   * Statistiche su uso archiviazione      |
|   |   * Riepilogo validazione pattern          |
|   |   * Risultati consolidamento              |
| +-----+                                         |
+-----+

```

4.3 Caratteristiche di Performance

Metriche del Sistema di Memoria:

MEMORIA DI LAVORO

- * Capacità: ~20K token (~80KB testo)
- * Tempo accesso: <1ms (in-memory)
- * Operazioni: 1000 per task
- * Durata: Singola esecuzione task

MEMORIA EPISODICA

- * Capacità: Milioni di episodi
- * Dimensione archiviazione: ~10KB per episodio (media)
- * Ricerca semantica: <100ms per top-K=10
- * Query strutturata: <50ms per ID, <200ms per attributi
- * Tasso hit cache: target >70%
- * Episodi memorizzati per giorno: 100-1000 (dipende da uso)

CACHE PATTERN

- * Capacità: Migliaia di pattern
- * Dimensione archiviazione: ~5KB per pattern (media)
- * Matching pattern: <50ms
- * Recupero pattern: <20ms (cached)
- * Ciclo validazione: Settimanale
- * Tasso scoperta pattern: 1-5 nuovi pattern per giorno

Scalabilità Archiviazione:

Anno 1 (10 utenti, sviluppo attivo):

Episodi: ~100K episodi
 Archiviazione: ~1GB episodica + 50MB pattern
 Costo: ~\$10/mese (cloud storage)

Anno 2 (100 utenti, produzione):

Episodi: ~1M episodi
 Archiviazione: ~10GB episodica + 500MB pattern
 Costo: ~\$50/mese

Anno 5 (1000 utenti, maturo):

Episodi: ~10M episodi (con consolidamento)
 Archiviazione: ~50GB episodica + 2GB pattern
 Costo: ~\$200/mese

Nota: Episodi più vecchi archiviati in storage freddo più economico

4.4 API del Sistema di Memoria

API di Alto Livello per Altre Componenti:

```
// API Memoria di Lavoro
MemoriaDiLavoro.inizializza(task)
MemoriaDiLavoro.imposta(chiave, valore, scope)
MemoriaDiLavoro.ottieni(chiave, scope)
MemoriaDiLavoro.inserisci_stato(stato)
MemoriaDiLavoro.aggiungi_intermedio(subtask_id, output)
MemoriaDiLavoro.pota()
MemoriaDiLavoro.pulisci()

// API Memoria Episodica
MemoriaEpisodica.memorizza_episodio(episodio)
MemoriaEpisodica.recupera_per_id(episodio_id)
MemoriaEpisodica.ricerca_semantica(query, k, filtri)
MemoriaEpisodica.ricerca_strutturata(criteri)
MemoriaEpisodica.ottieni_recenti(n, filtri)
MemoriaEpisodica.ottieni_fallimenti_simili(firma_fallimento, k)

// API Cache Pattern
CachePattern.aggiungi_pattern(pattern)
CachePattern.ottieni_pattern(pattern_id)
CachePattern.trova_applicabili(task, contesto)
CachePattern.registra_applicazione(pattern_id, episodio_id, risultato)
CachePattern.valida_pattern()
CachePattern.depreca_pattern(pattern_id, motivo)

// Query Integrate
Memoria.ottieni_contesto_per_task(task)
-> Restituisce: {
    pattern_applicabili: [Pattern],
    episodi_simili: [Episodio],
    conoscenza_dominio: [...]
}

Memoria.impara_da_episodio(episodio, intuizioni_riflessione)
-> Memorizza episodio, aggiorna pattern, restituisce riassunto_apprendimento
```

Prossimo: 04-capability-layer.md -> Specifiche Tool Registry, Model Router, Safety Verifier