0x06tudio

Progetto didattico di Ingegneria del Software
Università degli Studi di Padova



0x06tudio@gmail.com

Tracciabilità Semantica: Casi d'Uso vs Specifiche

Studio Tecnologico – Capitolato C9: SpecCheck (Zucchetti S.p.A.)

Informazioni sul documento

Versione:	0.1.0
Stato:	Approvato
Data ultimo aggiornamento:	2026/11/15
Uso:	Interno
Destinatari:	0x06tudio

Registro delle modifiche

Versione	Data	Descrizione	Autore	Verificatore
0.1.0	2026/11/15	Stesura del documento	Andrea Maggio	Catalin Chirilov

Contents

1 Introduzione **3**
1.1 Scopo del documento . . . 3
1.2 Riferimenti . . . 3
1.2.1 Riferimenti normativi . . . 3
1.2.2 Riferimenti informativi . . . 3

2 Contesto e Sfide della Tracciabilità nel Capitolato C9 **4**

3 Analisi delle Tecnologie Chiave **4**
3.1 NLP per Requirements Engineering (RE) . . . 4
3.2 Semantic Matching e Architetture RAG . . . 4
3.3 Rilevamento Logico delle Discrepanze . . . 5

4 Architettura Proposta e Proof of Concept **5**

5 Valutazione Comparativa delle Soluzioni **6**

6 Conclusioni e Raccomandazioni per il Progetto **7**

1 Introduzione

1.1 Scopo del documento

Il presente studio tecnologico analizza le metodologie, le architetture e gli strumenti idonei a implementare il modulo di **Tracciabilità Semantica tra Casi d'Uso e Specifiche dei Requisiti** nell'ambito del progetto *SpecCheck* (Capitolato C9 proposto da Zucchetti S.p.A.).

Nel contesto dello sviluppo guidato dall'Intelligenza Artificiale, in cui la generazione di artefatti avviene a velocità elevata, il collo di bottiglia si sposta dalla produzione alla verifica. L'obiettivo di questa analisi è definire una pipeline automatizzata, integrabile come estensione (*skill* e *tool*) nell'agente di *harness* **OpenCode**, capace di:

- estrarre e normalizzare le informazioni semantiche dai *Casi d'Uso*$_G$ (diagrammi testuali e documentazione descrittiva);
- allinearle e confrontarle con le *Specifiche dei Requisiti*$_G$ (espresse in Markdown e, opzionalmente, in PDF);
- rilevare formalmente discrepanze, omissioni e contraddizioni logiche, garantendo robustezza anche con modelli linguistici di dimensioni contenute (*Small Language Models*).

1.2 Riferimenti

1.2.1 Riferimenti normativi

- Capitolato d'appalto C9: *SpecCheck – Sviluppo di Skills per OpenCode per la verifica della tracciabilità end-to-end da casi d'uso a requisiti e codice sorgente* (Zucchetti S.p.A., A.A. 2026/2027);
- *Norme di Progetto* del gruppo *0x06tudio* (in fase di redazione).

1.2.2 Riferimenti informativi

- **OpenCode (Agente di Harness):** https://opencode.ai/;
- **Framework di gestione specifiche e harness:**
  - *OpenSpec*: https://openspec.dev/;
  - *GitHub Spec-Kit*: https://github.com/github/spec-kit;
  - *BMAD Method*: https://github.com/bmad-code-org/bmad-method;
- **Benchmark di sviluppo e coding:**
  - *SWE-bench*: https://github.com/swe-bench/SWE-bench;
  - *Terminal-bench*: https://www.tbench.ai/;
- **Piattaforme e modelli linguistici di riferimento:**
  - *Hugging Face* (raccolta modelli LLM e SLM come Gemma e Qwen): https://huggingface.co/;
- **Tecnologie di Information Retrieval e Database Vettoriali:**
  - *BM25 (Best Matching 25)*: Robertson, S., & Zaragoza, H. (2009). *The Probabilistic Relevance Framework: BM25 and Beyond.* Foundations and Trends in Information Retrieval.
  - *LanceDB* (Database vettoriale embedded serverless): https://lancedb.com/
  - *Chroma* (AI-native open-source vector database): https://www.trychroma.com/
  - *SQLite-vec* (Vector search extension per SQLite): https://github.com/asg017/sqlite-vec

2 Contesto e Sfide della Tracciabilità nel Capitolato C9

Il capitolato C9 richiede di verificare che ogni *Caso d'Uso*$_G$ raccolto nella fase di analisi con gli stakeholder trovi effettiva e coerente declinazione nelle *Specifiche dei Requisiti*$_G$ (requisiti obbligatori 1-5). Questa attività presenta tre criticità architetturali fondamentali:

1. **Asimmetria lessicale e strutturale (*Abstraction Gap*):** I Casi d'Uso adottano una prospettiva esterna orientata all'utente (*attore, precondizioni, flusso principale, estensioni*), mentre le Specifiche dei Requisiti assumono una forma propositiva e sistemica (*requisiti funzionali RF, non funzionali RNF, vincoli*). Un semplice confronto sintattico o basato su parole chiave fallisce sistematicamente.

2. **Cardinalità molti-a-molti ($N : M$):** Come evidenziato nel capitolato, un singolo Caso d'Uso può essere soddisfatto da più requisiti atomici e, viceversa, un requisito trasversale (es. autenticazione o logging) può impattare su molteplici Casi d'Uso.

3. **Vincolo sui modelli e determinismo:** Zucchetti metterà a disposizione sia modelli frontier (con tetto di spesa) sia modelli locali/ridotti come **Gemma 4 26B** e **Qwen 3.8 27B** senza limiti di utilizzo. Inoltre, il capitolato prescrive che le attività di analisi strutturata e verifica non siano affidate al solo ragionamento non-deterministico dell'LLM, ma supportate da *plugin* o applicativi esterni deterministici richiamati tramite *skill* e *tool* di OpenCode.

3 Analisi delle Tecnologie Chiave

3.1 NLP per Requirements Engineering (RE)

L'applicazione dei Large Language Models al *Requirements Engineering*$_G$ consente di superare i limiti delle tecniche *NLP*$_G$ classiche (come TF-IDF o Latent Semantic Analysis) nella comprensione dell'intento funzionale.

- **Decomposizione Atomica e Normalizzazione:** Prima del confronto, l'LLM viene istruito tramite *prompting strutturato* a scomporre sia i Casi d'Uso (inclusi diagrammi Mermaid, PlantUML o XMI) sia le specifiche in *unità semantiche atomiche* (es. *Soggetto - Azione - Oggetto - Condizione*).

- **Output Strutturati (JSON Schema):** Per garantire che modelli da 26B–27B parametri (Gemma 4, Qwen 3.8) producano output affidabili e consumabili da un plugin deterministico, è indispensabile vincolare la generazione tramite *Constrained Decoding* (JSON Schema / Pydantic), estraendo entità standardizzate (ID Caso d'Uso, Attori, Postcondizioni, ID Requisito).

- **Gestione del Contesto (Harness Engineering):** In linea con l'architettura di OpenCode e di framework come *OpenSpec*, il contesto non deve essere saturato caricando interi documenti in un unico prompt. Le *skill* devono orchestrare letture incrementali e produrre artefatti intermedi sintetici (es. file di mappatura JSON/Markdown).

3.2 Semantic Matching e Architetture RAG

Quando la base documentale cresce, l'approccio *brute-force* (confronto a coppie $O(N \times M)$ di tutti i Casi d'Uso contro tutti i Requisiti via LLM) diventa lento e costoso. L'adozione di una pipeline *RAG*$_G$ (*Retrieval-Augmented Generation*) risolve il problema della scalabilità:

1. **Ingestion e Chunking Semantico:**
   - *Documenti Markdown (Requisito Obbligatorio):* Parsing basato sull'albero sintattico (AST Markdown) per preservare la gerarchia delle sezioni, i blocchi di diagrammi (Mermaid) e gli identificativi univoci (es. `UC-01`, `RF-04`).
   - *Documenti PDF (Requisito Opzionale):* Utilizzo di parser layout-aware (es. *Docling* o *PyMuPDF4LLM*) capaci di convertire tabelle, elenchi e didascalie di diagrammi UML in Markdown pulito, evitando la perdita di struttura tipica dell'estrazione testuale grezza.
2. **Hybrid Search (Dense + Sparse Retrieval):** I testi dei requisiti vengono indicizzati in un database vettoriale locale leggero (es. *LanceDB*, *Chroma* o *SQLite-vec*). La ricerca combina *Dense Embeddings* (per cogliere la sinonimia semantica tra linguaggio utente e linguaggio tecnico) e ricerca lessicale *BM25* (fondamentale per preservare riferimenti esatti a codici, acronimi e nomi di entità di dominio).
3. **Cross-Encoder Reranking:** Per ogni Caso d'Uso, il retriever estrae i primi k requisiti candidati (es. $k = 10$); un modello *Cross-Encoder* leggero riordina le coppie (UC_i, Req_j) assegnando un punteggio di affinità preciso prima di passarle all'LLM per la verifica logica.

Prima di giungere a una pipeline ibrida, un primo approccio intuitivo potrebbe consistere nel ricorrere alla sola similarità degli embedding (confronto vettoriale puro tramite similarità del coseno tra i testi dei Casi d'Uso e dei Requisiti). Tuttavia, sebbene questo metodo sia rapido ed efficiente su grandi volumi di dati, fallisce inevitabilmente nel cogliere sfumature logiche fondamentali (quali negazioni, inversioni di ruolo, contraddizioni sottili) poiché due frasi semanticamente opposte possono comunque presentare vettori numerici molto vicini nello spazio geometrico.

3.3 Rilevamento Logico delle Discrepanze

Il cuore del sistema di verifica (Requisito 5) consiste nel classificare formalmente la relazione tra un Caso d'Uso e i requisiti candidati recuperati tramite RAG. L'analisi si ispira ai task di *Natural Language Inference* (NLI_G) adattati all'ingegneria del software, classificando ogni collegamento in quattro stati:

- **Copertura Completa (*Entailment*):** L'insieme dei requisiti collegati soddisfa integralmente lo scenario principale, le precondizioni e i flussi alternativi del Caso d'Uso.
- **Copertura Parziale / Sottospecificazione:** Il requisito copre l'azione base del Caso d'Uso ma omette vincoli, eccezioni o postcondizioni esplicitate nel diagramma o nella descrizione.
- **Contraddizione Logica:** I requisiti definiscono comportamenti incompatibili con il Caso d'Uso (es. ruoli autorizzati differenti, stati di uscita discordanti, vincoli temporali opposti).
- **Mancata Copertura (*Orphan Use Case*) / Requisito non tracciato (*Gold Plating*):** Un Caso d'Uso privo di requisiti associati segnala un'omissione progettuale; viceversa, un requisito funzionale privo di alcun Caso d'Uso genitore evidenzia potenziale *scope creep* (funzionalità non richieste).

4 Architettura Proposta e Proof of Concept

Per rispettare il vincolo architetturale di OpenCode (separazione tra ragionamento probabilistico dell'LLM ed esecuzione deterministica), si propone un'architettura ibrida composta da:

- **OpenCode Skill (`speccheck-trace-uc`):** Definisce il workflow operativo, i prompt di estrazione e le regole di giudizio per l'LLM.
- **Tool / Eseguibile Esterno (Python CLI):** Gestisce il parsing deterministico di Markdown/Mermaid/P l'indicizzazione vettoriale locale, la generazione della matrice di tracciabilità in formato `JSON` e l'output formattato per la dashboard TUI.

```python
from pydantic import BaseModel
from typing import Literal, List

class TraceabilityVerdict(BaseModel):
    use_case_id: str
    matched_req_ids: List[str]
    status: Literal["COVERED", "PARTIAL", "CONTRADICTION", "MISSING"]
    missing_aspects: List[str]
    reasoning: str

def verify_use_case(uc_chunk: dict, vector_store, llm_client) ->
    TraceabilityVerdict:
    # 1. Fase deterministica/RAG: recupero Top-K requisiti candidati
    candidates = vector_store.hybrid_search(
        query=uc_chunk["normalized_intent"],
        top_k=8
    )

    # 2. Fase NLP/NLI: valutazione logica vincolata a JSON Schema
    prompt = build_nli_verification_prompt(uc_chunk, candidates)
    verdict = llm_client.generate_structured(
        model="qwen-3.8-27b",
        prompt=prompt,
        response_format=TraceabilityVerdict
    )
    return verdict
```

Listing 1: Schema concettuale della pipeline ibrida di verifica UC-Requisiti.

5 Valutazione Comparativa delle Soluzioni

Di seguito vengono confrontate le principali strategie architetturali per l'implementazione del motore di tracciabilità semantica, valutate rispetto ai vincoli del capitolato C9.

Criterio	**Full-Context LLM (Zero-RAG)**	**Solo Embedding Similarity**	**Pipeline Ibrida (RAG + NLI Strutturato)**
Scalabilità su progetti grandi	Bassa (degrado per *lost-in-the-middle* e limiti di token)	Alta (ricerca vettoriale istantanea)	**Alta** (filtra i candidati prima dell'inferenza LLM)
Rilevamento contraddizioni	Alta con modelli frontier, instabile con SLM	Nulla (due frasi opposte hanno alta similarità coseno)	**Alta** (il passo NLI confronta esplicitamente pre/post-condizioni)
Resa con SLM (Gemma 4 / Qwen 3.8)	Media (difficoltà a ragionare su decine di documenti insieme)	Non applicabile	**Ottima** (contesto focalizzato sul singolo UC + Top-K requisiti)
Determinismo e riproducibilità	Bassa	Alta (ma priva di spiegazione logica)	**Alta** (parsing/retrieval deterministici + output JSON vincolato)
Complessità implementativa	Bassa	Media	Media-Alta (richiede parser AST e DB vettoriale embedded)

6 Conclusioni e Raccomandazioni per il Progetto

L'analisi condotta dimostra che affidare la verifica di tracciabilità direttamente a un LLM caricando l'intera documentazione nel contesto non soddisfa i requisiti di precisione, spiegabilità e compatibilità con modelli ridotti richiesti da Zucchetti S.p.A.

Per lo sviluppo del *Proof of Concept* (PoC) e del successivo *Minimum Viable Product* (MVP) di **SpecCheck**, si raccomanda l'adozione di una **pipeline ibrida deterministico-semantica**:

1. **Core deterministico in Python (CLI / Plugin):** responsabile del parsing strutturato dei file Markdown e dei diagrammi Mermaid/PlantUML, della gestione di un indice vettoriale locale leggero (es. *SQLite-vec* o *LanceDB*) con ricerca ibrida (vettoriale + BM25) e della persistenza dei risultati.
2. **Modulo di inferenza NLI vincolato:** invocato tramite le *skill* di OpenCode su porzioni ristrette di contesto (singolo Caso d'Uso e relativi requisiti candidati) con output forzato su schema JSON, garantendo affidabilità sia su modelli frontier sia su **Gemma 4 26B** e **Qwen 3.8 27B**.
3. **Estendibilità per i requisiti opzionali:** l'architettura modulare permette di integrare successivamente parser PDF (*Docling*/*PyMuPDF4LLM*) nello stadio di *ingestion* senza alterare la logica di confronto e rilevamento delle discrepanze.