RELAZIONE TECNICA: SPECIFICHE E SCELTE PROGETTUALI

Progetto: Multi-Agent Test Generator
Studente: Gianmarco Riviello

1. SPECIFICHE DEL SISTEMA

1.1 Obiettivo del Progetto

Sviluppare un sistema multi-agente per la generazione automatica di test unitari Python con garanzia di copertura di branch >= 80%, utilizzando:
• Validazione formale tramite grammatica Lark personalizzata
• Generazione intelligente tramite Google Gemini AI
• Misurazione affidabile tramite coverage.py

1.2 Specifiche Funzionali

- **SPEC-F1: Validazione Grammatica Rigorosa:** Parsing con grammatica formale (python_subset.lark). Supporta definizioni, if/else annidati e operatori base.
- **SPEC-F2: Analisi Statica AST:** Estrazione di BranchMap, branch points e complessità ciclomatica via modulo ast.
- **SPEC-F3: Generazione Test Iniziale:** Invocazione Google Gemini 2.5 Flash con temperatura 0.2 per determinismo.
- **SPEC-F4: Misurazione Coverage:** Esecuzione via pytest-cov con report JSON e timeout di 60 secondi.
- **SPEC-F5: Ottimizzazione Iterativa:** Identificazione branch non coperti e generazione di test addizionali (max 5 iterazioni).

1.3 SPECIFICHE NON-FUNZIONALI

- NFR-1: Performance
- - Target: < 30 secondi per funzione semplice (< 10 branch)
- - Misurato: ~5-12s (superato)

- NFR-2: Reliability
- - Coverage >= 80% in >= 95% dei casi
- - Retry logic: 3 tentativi per API failures

- NFR-3: Security
- - NO command injection: sanitizzazione input, shell=False

- - NO path traversal: validazione file paths
- - API key in environment variable (no hardcoding)
- - Timeout per prevenire hang/DoS

- NFR-4: Usability
- - Interfaccia programmatica: orchestrator.generate_tests()
- - Launcher interattivo: run_demos.py
- - Logging verboso opzionale per debugging

- NFR-5: Maintainability
- - Type hints su >85% funzioni

- NFR-6: Portability
- - Python 3.10+ (testato fino a 3.13)
- - Windows, Linux, macOS compatibili
- - Dipendenze pinned in requirements.txt

2. SCELTE PROGETTUALI E GIUSTIFICAZIONI

2.1 Architettura: Pattern Multi-Agente

Il sistema è basato su agenti specializzati (CodeAnalyzer, UnitTestGenerator, CoverageOptimizer) per garantire la separazione delle responsabilità (Separation of Concerns) e facilitare la manutenibilità.

2.2 Orchestrazione: LangGraph State Machine

È stato utilizzato LangGraph per gestire il workflow non lineare, supportando nativamente i cicli di ottimizzazione e le decisioni condizionali basate sul livello di coverage raggiunto.

2.3 Metrica Coverage: Branch Coverage

Si è scelto il branch coverage invece del semplice statement coverage perché più rigoroso, garantendo che ogni percorso decisionale del codice venga testato.

2.4 LLM Provider: Google Gemini

SCELTA: Google Gemini 2.5 Flash come provider principale

Performance - Raggiunge spesso 100% coverage al primo tentativo

Costo Zero - API key gratuita (60 req/min)

Python-Optimized - Eccellente per generazione codice Python

LangChain Native - Supporto nativo

Velocità - Risposte < 2s tipicamente

3. SECURITY E BEST PRACTICES

- **Prevenzione Command Injection:** Uso di liste di argomenti in subprocess.run e shell=False esplicito.
- **Protezione Path Traversal:** Validazione dei percorsi file tramite os.path.abspath e verifica della directory di base.
- **Gestione Secrets:** Caricamento della GOOGLE_API_KEY tramite variabili d'ambiente.
- **Qualità del Codice:** Uso di Pydantic per la validazione dei dati e type hints su oltre l'85% delle funzioni.

4. RISULTATO

Sistema robusto, estensibile, sicuro, e dimostrativo delle moderne tecniche di ingegneria del software per sistemi AI-powered.