0x06tudio
Progetto didattico di Ingegneria del Software
Università degli Studi di Padova

0x06tudio@gmail.com

Studio Tecnologico: Benchmark per Agenti Software e Metodologie Spec-Driven

Approfondimento Tecnologico Modulo 2 – Capitolato C9 (SpecCheck)

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 dello studio tecnologico su benchmark per agenti software	Catalin Chirilov	Davide Colabove

Indice

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 Panoramica e Contesto **3**
2.1 Dal Vibe Coding al collo di bottiglia della verifica . . . 3
2.2 Anatomia dell'Harness e ruolo di OpenCode . . . 4

3 Analisi Tecnica **4**
3.1 Evoluzione dei Benchmark: dai test statici agli agenti software . . . 4
3.2 Frontiera di Pareto e vincoli sui modelli per SpecCheck . . . 5
3.3 Workflow AI-Assisted e Framework Spec-Driven . . . 5
3.4 Esempio pratico: orchestrare verifica semantica e plugin deterministico . . . 6
3.5 Progetti simili: strumenti deterministici di Requirements Traceability . . . 6

4 Valutazione e Confronto **7**
4.1 Confronto empirico tra BMAD, Spec-Kit e OpenSpec . . . 7
4.2 Vantaggi, limiti e implicazioni per SpecCheck . . . 7
4.3 Metodologia di Benchmark interno per SpecCheck (PoC e MVP) . . . 8

5 Conclusioni **8**

1 Introduzione

1.1 Scopo del documento

Il presente documento costituisce lo studio tecnologico relativo a ***Benchmark*$_G$ per Agenti Software e Metodologie** per il progetto *SpecCheck* (Capitolato C9, proposto da Zucchetti S.p.A.). L'obiettivo dell'analisi è duplice:

- esaminare lo stato dell'arte dei Benchmark per agenti software e dei framework di sviluppo guidati dalle specifiche Spec-Driven Development, evidenziando i limiti della generazione estemporanea di codice e l'impatto dell'infrastruttura di orchestrazione *Harness*$_G$;
- definire criteri progettuali e metriche di valutazione interni che il gruppo si impegnerà a rispettare e a seguire durante la realizzazione e collaudo di *SpecCheck* su *OpenCode*, garantendo la verificabilità end-to-end tra casi d'uso, specifiche dei requisiti e codice sorgente sia su modelli di grandi dimensioni sia su modelli locali compatti.

1.2 Riferimenti

1.2.1 Riferimenti normativi

- Capitolato d'appalto C9 proposto dal proponente Zucchetti S.p.A., A.A. 2026/2027;
- *Norme di Progetto* (gruppo *0x06tudio*, in fase di redazione).

1.2.2 Riferimenti informativi

- Documentazione ufficiale *OpenCode*: https://opencode.ai/ (ultimo accesso: 2026/11/15);
- *SWE-Bench* – Benchmark per la risoluzione di issue software reali: https://github.com/swe-bench/SWE-bench (ultimo accesso: 2026/11/15);
- *Terminal-bench* – Valutazione di agenti su flussi operativi da terminale: https://www.tbench.ai/ (ultimo accesso: 2026/11/15);
- Framework *OpenSpec*: https://openspec.dev/ (ultimo accesso: 2026/11/15);
- Framework *BMAD Method*: https://github.com/bmad-code-org/bmad-method (ultimo accesso: 2026/11/15);
- Framework *Spec-Kit*: https://github.com/github/spec-kit (ultimo accesso: 2026/11/15);
- I. Eretz Kdosha, *I Tested Three Spec-Driven AI Tools. Here's My Honest Take*, Ran the Builder Blog, Aprile 2026: https://ranthebuilder.cloud/blog/i-tested-three-spec-driven-ai-to (ultimo accesso: 2026/11/15);
- *OpenFastTrace* – Tool open-source per il tracciamento deterministico dei requisiti: https://github.com/itsallcode/openfasttrace (ultimo accesso: 2026/11/15);
- *Doorstop* – Gestione e tracciabilità dei requisiti versionati su Git: https://github.com/doorstop-dev/doorstop (ultimo accesso: 2026/11/15);
- *StrictDoc* – Piattaforma per specifiche tecniche e tracciabilità bidirezionale: https://github.com/strictdoc-project/strictdoc (ultimo accesso: 2026/11/15).

2 Panoramica e Contesto

2.1 Dal Vibe Coding al collo di bottiglia della verifica

L'evoluzione dei Large Language Models (LLM_G) ha trasformato profondamente la produzione del software. Nella prima fase di adozione, sintetizzata dal concetto di *Vibe Coding*$_G$ formulato da Andrej Karpathy (*"il prossimo linguaggio di programmazione sarà l'inglese"*), lo sviluppatore descrive ad alto livello l'intento funzionale lasciando che il modello generi direttamente il codice sorgente.

Tuttavia, l'applicazione su sistemi reali di dimensioni medio-grandi ha evidenziato rapidamente che il *Vibe Coding* puro degenera in una forma moderna di *cowboy coding*: l'interazione conversazionale priva di vincoli architetturali produce codice frammentato, difficile da manutenere e spesso non allineato ai requisiti funzionali iniziali. Come evidenziato nel capitolato C9, l'automazione della scrittura ha spostato il vero collo di bottiglia dell'ingegneria del software dalla **costruzione** alla **verifica**:

- gli strumenti tradizionali di analisi statica (compilatori, linter, test unitari) verificano la correttezza sintattica e il comportamento locale del programma, ma non garantiscono che il codice implementi effettivamente i desiderata espressi nei casi d'uso;
- scrivere codice a velocità agentica senza un presidio formale equivale a produrre ambiguità su larga scala.

2.2 Anatomia dell'Harness e ruolo di OpenCode

Per superare i limiti del semplice chatbot sono nati i sistemi di gestione del contesto e orchestrazione agentica, definiti in letteratura come $Harness_G$ (tra cui *OpenCode*, *Claude Code*, *Codex* e *OpenHands*). Un *Harness* fornisce l'impalcatura operativa attorno all'LLM attraverso:

- **memoria architetturale persistente**, tramite file strutturali di progetto (ad esempio `AGENTS.md`) che vincolano le scelte tecnologiche lungo tutto il ciclo di vita;
- **estensioni modulari**, articolate in $Skill_G$ (file dichiarativi che istruiscono l'agente su come eseguire una procedura specifica), comandi invocabili dall'utente e $Tool_G$ (strumenti per leggere file, eseguire comandi di sistema o interrogare programmi esterni);
- **componenti deterministici di supporto** (plugin o eseguibili sidecar), indispensabili per eseguire controlli oggettivi e tracciamenti di stato che un modello probabilistico non può garantire da solo.

Durante l'incontro esplorativo con il proponente (Zucchetti S.p.A.), è emerso chiaramente come l'efficacia di *OpenCode* dipenda in modo diretto dalla qualità delle descrizioni associate a comandi, *skill* e *tool*: tali descrizioni guidano il ragionamento dell'agente nella selezione dei passi operativi. Nel medesimo incontro, il proponente ha inoltre ribadito la centralità dei **casi d'uso** come punto di origine della catena di tracciabilità (*Casi d'Uso* $\rightarrow$ *Specifiche dei Requisiti* $\rightarrow$ *Codice Sorgente*), sottolineando come la verifica automatica debba partire dagli scenari raccolti con gli utenti per accertarne la copertura lungo l'intero ciclo di sviluppo.

3 Analisi Tecnica

3.1 Evoluzione dei Benchmark: dai test statici agli agenti software

La misurazione delle capacità degli LLM in ambito ingegneristico ha attraversato un netto cambio di paradigma:

- **Benchmark tradizionali (MMLU, GSM8K):** valutano l'erudizione enciclopedica, la comprensione del testo e il ragionamento logico-matematico su singoli prompt isolati. Pur utili per stimare le capacità linguistiche di base, risultano inadeguati a misurare la capacità di operare su repository complessi.
- **Benchmark agentici (SWE-Bench, Terminal-bench):** valutano l'intero ciclo operativo su progetti reali. *SWE-Bench* misura la percentuale di issue GitHub reali risolte autonomamente dall'agente (comprensione della codebase, modifica multi-file e superamento delle suite di test), mentre *Terminal-bench* valuta l'esecuzione di sequenze articolate di comandi da riga di comando, impacchettamento e diagnostica.

Un dato cruciale per il progetto *SpecCheck* riguarda il peso dell'*Harness* rispetto al singolo modello linguistico. Secondo le rilevazioni riportate nel capitolato C9, l'architettura di gestione del contesto e pianificazione incide per circa il **30%** sulle prestazioni complessive di sistemi di vertice su *SWE-Bench* (come nel caso di *ChatGPT 6 Astra*). Analogamente, le classifiche *SWE-Bench* di metà 2026 mostrano come la combinazione tra agente e modello determini scarti significativi a parità di famiglia di modelli (ad esempio, *AMI Agent + Claude-4.6-Opus* raggiunge il 63,0% di issue risolte contro il 36,0% di *SWE-agent + Claude-4.5-Sonnet* e il 25,0% di *Slingshot-v3.1.0 + Claude-4.6-Opus*). Questo conferma che un'ingegnerizzazione accurata delle *skill* e dei *tool* di *OpenCode* può compensare in larga misura i limiti intrinseci del modello sottostante.

3.2 Frontiera di Pareto e vincoli sui modelli per SpecCheck

L'analisi della frontiera di Pareto tra qualità (Arena Score) e costo per milione di token (rapporto input/output 3:1) evidenzia due fasce operative distinte:

- **Modelli proprietari di frontiera:** offrono i punteggi massimi (es. `gpt-6-astra-max` con 1796 punti a $40/M token, `claude-opus-5-max` con 1688 punti a $20/M token, `qwen3.8-max-0902` con 1685 punti a $5/M token), ma comportano costi operativi elevati su analisi iterative di interi repository;
- **Modelli compatti ed economici:** mantengono prestazioni competitive a una frazione del costo (es. `qwen3.8-flash-next` con 1631 punti a $0,39/M token, `glm-5.3-flash` con 1605 punti a $0,21/M token, fino a modelli open-weight leggeri come `granite-4.1-8b` a $0,09/M token).

Il capitolato C9 stabilisce che Zucchetti metterà a disposizione sia modelli di grandi dimensioni (soggetti a un tetto massimo di budget) sia modelli compatti eseguibili senza limitazioni, quali **Gemma 4 26B** e **Qwen 3.8 27B**. Di conseguenza, la progettazione delle *skill* di *SpecCheck* non può fare affidamento esclusivamente sulla capacità di ragionamento di modelli frontier, ma deve scomporre la verifica in sotto-compiti atomici e deterministi affinché anche modelli nella fascia 26B–27B ottengano risultati affidabili.

3.3 Workflow AI-Assisted e Framework Spec-Driven

Nell'ecosistema degli agenti di sviluppo si distinguono due famiglie principali di estensioni:

1. **Raccolte di skill general-purpose:** forniscono workflow trasversali per interviste preliminari, pianificazione, scrittura e debug (tra cui *SuperPowers*, le *Agent-skills* di Addy Osmani e *Grill me* di Matt Pocock). Pur migliorando la disciplina del prompt, non impongono una struttura formale di tracciabilità tra requisiti e sorgente.
2. **Framework Spec-Driven:** strutturano l'intero ciclo di sviluppo attorno a specifiche scritte e revisionate prima della generazione del codice. I principali riferimenti individuati sono *BMAD Method*, *Spec-Kit*, *OpenSpec* e *AI Unified Process*.

Tra questi, **OpenSpec** rappresenta il riferimento architetturale direttamente indicato dal capitolato C9. Il suo funzionamento si basa su:

- **Delta Specs:** invece di rigenerare l'intera documentazione di sistema ad ogni modifica, ogni proposta descrive esclusivamente i requisiti aggiunti, modificati o rimossi (*delta*), che vengono poi consolidati nella documentazione principale (*source-of-truth*) al completamento;
- **Architettura ibrida:** combina comandi slash e *skill* interpretati dall'LLM con un eseguibile CLI dedicato che gestisce in modo deterministico lo stato degli artefatti, il grafo delle dipendenze e la dashboard testuale da terminale (`openspec view`, `openspec status`).

3.4 Esempio pratico: orchestrare verifica semantica e plugin deterministico

Per evitare che l'LLM allucini o salti controlli sui file di progetto, il pattern consigliato per *SpecCheck* prevede che la *skill* di *OpenCode* orchestri sia l'analisi semantica sia l'invocazione di un plugin/eseguibile esterno dedicato alla mappatura deterministica dei file e al calcolo della copertura. Di seguito si riporta uno schema esemplificativo di definizione dichiarativa di una *skill* di verifica:

```
---
name: speccheck-verify-traceability
description: Verifica la corrispondenza end-to-end tra casi d'uso,
  specifiche dei requisiti e codice sorgente, invocando il plugin
  deterministico per l'estrazione strutturale e il report di copertura.
---

# Procedura di Verifica SpecCheck

1. Invoca il tool `speccheck_parse_artifacts` per estrarre in formato
   JSON
   l'elenco deterministico di Casi d'Uso (UC-ID) e Requisiti (REQ-ID).
2. Confronta semanticamente ciascun UC-ID con i REQ-ID associati,
   segnalando eventuali casi d'uso orfani o ambigui.
3. Analizza i moduli sorgente mappati dal plugin e verifica che ogni
   requisito funzionale sia effettivamente implementato nel codice.
4. Genera il file di stato JSON e avvia la dashboard TUI di riepilogo.
```

Listing 1: Schema esemplificativo di una Skill di verifica per OpenCode

3.5 Progetti simili: strumenti deterministici di Requirements Traceability

Oltre ai framework generativi basati su LLM, l'analisi dello stato dell'arte ha considerato strumenti open-source consolidati nell'ambito del *Requirements Traceability*, utili come riferimento architetturale per la componente deterministica (plugin) di *SpecCheck*:

- **OpenFastTrace (OFT):** strumento leggero che analizza file Markdown, codice sorgente e suite di test per verificare la copertura lungo l'intera catena di tracciabilità, segnalando requisiti scoperti o artefatti orfani. Rappresenta il riferimento più affine alla logica di parsing e calcolo della copertura prevista per il plugin di *SpecCheck*.
- **Doorstop:** tool in Python che gestisce i requisiti come file testuali strutturati (YAML/Markdown) versionati su Git, rilevando automaticamente i cosiddetti *suspect links* quando un requisito a monte viene modificato senza un corrispondente aggiornamento nel codice o nei test collegati.
- **StrictDoc:** piattaforma per *specifications-as-code* che supporta la tracciabilità bidirezionale (*Requisiti* → *Architettura* → *Codice* → *Test*) e la generazione automatica di report di copertura.

Il limite principale di questi strumenti classici risiede nell'approccio puramente sintattico: essi richiedono che lo sviluppatore inserisca manualmente identificativi rigidi nei commenti del codice sorgente (ad esempio `REQ-001`) e non effettuano alcuna verifica semantica. Di conseguenza, non sono in grado di rilevare se un modulo sorgente implementi realmente il comportamento descritto in un caso d'uso o se presenti divergenze logiche rispetto alla specifica. *SpecCheck* supera questo limite combinando l'estrazione strutturale deterministica (ispirata a *OpenFastTrace* e *OpenSpec*) con l'analisi semantica guidata da *skill* in *OpenCode*.

4 Valutazione e Confronto

4.1 Confronto empirico tra BMAD, Spec-Kit e OpenSpec

Per valutare l'efficacia concreta dei framework *spec-driven*, sono stati analizzati i risultati di un benchmark comparativo indipendente condotto nel 2026 sullo sviluppo della medesima funzionalità backend (servizio serverless in Python con autenticazione, sicurezza e Infrastructure-as-Code, eseguito con *Claude Sonnet 4.5*). I tre strumenti sono stati valutati su 13 dimensioni qualitative e quantitative (scala 1–5), oltre che su tempi e costi complessivi.

Criterio	**BMAD (v6.0.3)**	**Spec-Kit (v0.1.6)**	**OpenSpec (v1.2.0)**
Punteggio globale	3,65 / 5 (Full) 3,74 / 5 (Quick)	2,77 / 5	**4,00 / 5** (Migliore)
Approccio alle specifiche	PRD completo, architettura e user story (Full) o documento unico (Quick)	`constitution` globale di progetto + specifiche con flag `NEEDS CLARIFICATION`	*Delta specs* incrementali unite a una *source-of-truth* persistente
Verifica e Revisione	Eccellente (5/5): *adversarial code review* dedicata e revisione multi-agente (*Party Mode*)	Carente (2/5): verifica coerenza tra documenti (`/speckit.analyze`), ma nessuna code review integrata	Fluida ma non vincolante: permette iterazioni rapide, ma senza gate automatici tra fasi
Tempi e Costi stimati	Full: 6 giorni, \$200 Quick: 2 giorni, \$85	1,5 giorni, \$75	**1,3 giorni, \$95**
Sviluppo parallelo	Limitato (2/5): output in cartelle condivise di default	Elevato (5/5): directory isolate per singola feature	Elevato (5/5): directory isolate per singola modifica
Criticità principali	Curva di apprendimento ripida (12 agenti) e forte pesantezza nella modalità Full	Il codice generato spesso non rispecchia fedelmente la specifica; aggiornamenti distruttivi	*Bus factor* pari a 1 (singolo maintainer) e rigidità sulla struttura delle cartelle

4.2 Vantaggi, limiti e implicazioni per SpecCheck

Dall'analisi comparativa emergono tre insegnamenti diretti per la progettazione di *SpecCheck*:

- **Pro – Validazione dell'esigenza di SpecCheck:** nel test empirico, né *OpenSpec* né *Spec-Kit* hanno rilevato autonomamente una lacuna nei controlli di autorizzazione durante la pianificazione, e *Spec-Kit* ha generato codice non aderente alle specifiche. Ciò dimostra che i framework attuali sono ottimizzati per la **generazione**, lasciando scoperto il problema della **verifica formale e della tracciabilità inversa** richiesto da Zucchetti.
- **Pro – Complementarità tra OpenSpec e BMAD:** *OpenSpec* offre l'architettura più leggera ed efficace per tracciare lo stato degli artefatti e visualizzarlo da terminale (`openspec view`), mentre *BMAD* dimostra l'efficacia di una fase di **revisione avversariale** (*adversarial review*) separata dalla fase di generazione. *SpecCheck* dovrebbe integrare entrambi i principi: la struttura leggera e deterministica di *OpenSpec* unita a una logica di verifica critica ispirata a *BMAD*.
- **Contro e mitigazioni (Anti-Lock-In):** data la rapida evoluzione e il rischio di *bus factor* dei tool emergenti, il gruppo *0x06tudio* deve incapsulare l'installazione (tramite script Bash/PowerShell unico come richiesto dal capitolato), definire formati interni standardizzati

(Markdown/Mermaid/XMI e JSON) e astrarre i comandi utente tramite *skill* proprietarie di *OpenCode*.

4.3 Metodologia di Benchmark interno per SpecCheck (PoC e MVP)

Poiché i benchmark pubblici come *SWE-Bench* misurano la capacità di scrivere patch di codice ma non la capacità di verificare la tracciabilità documentale, il gruppo deve adottare un **protocollo di benchmark interno** per validare *SpecCheck* durante le fasi di *Proof of Concept*$_G$ (PoC) e *Minimum Viable Product*$_G$ (MVP):

1. **Costruzione del Golden Dataset:** selezione di un progetto software campione dotato della catena completa *Casi d'Uso (Mermaid/XMI + descrizioni)* → *Specifiche dei Requisiti (Markdown)* → *Codice Sorgente* (eventualmente adattando benchmark accademici consolidati per il *Traceability Link Recovery*, quali *eTour* o *iTrust*, già dotati di *ground truth* verificata), in cui vengono iniettate in modo controllato quattro classi di anomalie:
   - *Caso d'uso non coperto:* scenario presente nei diagrammi ma assente nei requisiti;
   - *Requisito non implementato:* specifica formalizzata ma priva di riscontro nel codice;
   - *Implementazione non conforme:* comportamento del codice divergente rispetto alla specifica;
   - *Codice non tracciato (requisito opzionale):* funzionalità presenti nel sorgente ma non derivate da alcuna specifica.
2. **Metriche quantitative di valutazione:** per ciascuna esecuzione vengono misurate *Precision* (percentuale di discrepanze segnalate che corrispondono a difetti reali, riducendo i falsi allarmi per lo sviluppatore), *Recall* (percentuale di discrepanze reali effettivamente individuate), *F1-Score*, tempo di esecuzione e consumo di token.
3. **Confronto multi-modello e regressione automatica:** ogni versione delle *skill* verrà eseguita sul medesimo *Golden Dataset* confrontando almeno un modello frontier (entro il budget aziendale) con i modelli locali **Gemma 4 26B** e **Qwen 3.8 27B**. La suite costituirà la base per i test automatici di regressione richiesti a partire dall'MVP.

5 Conclusioni

L'analisi dei benchmark per agenti software e delle metodologie *spec-driven* conferma la solidità e l'attualità delle scelte architetturali delineate nel capitolato C9. Il passaggio da test statici (*MMLU*) a benchmark agentici (*SWE-Bench*) dimostra che l'affidabilità di un sistema basato su LLM dipende in misura determinante dall'*Harness* e dalla scomposizione procedurale dei compiti, più che dalla sola dimensione del modello.

Alla luce del confronto tra *BMAD*, *Spec-Kit* e *OpenSpec*, e delle indicazioni emerse nell'incontro con Zucchetti S.p.A., il gruppo *0x06tudio* adotterà le seguenti linee guida operazionali:

- replicare il modello architetturale ibrido di *OpenSpec*, affiancando alle *skill* e ai comandi di *OpenCode* un plugin/eseguibile deterministico dedicato al parsing strutturale, al calcolo della copertura e al rendering della dashboard;
- progettare prompt e descrizioni delle *skill* secondo un approccio di *verifica avversariale* granulare, così da garantire elevata accuratezza anche sui modelli compatti **Gemma 4 26B** e **Qwen 3.8 27B**;
- predisporre fin dalla fase di PoC un *Golden Dataset* articolato sulla catena *Casi d'Uso* → *Requisiti* → *Codice* per misurare oggettivamente *Precision*, *Recall* e costi prima del rilascio dell'MVP.