

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

UNIVERSITÀ DEGLI STUDI DI MODENA

E REGGIO EMILIA

Dipartimento di Ingegneria “Enzo Ferrari”

Corso di Laurea Magistrale in Ingegneria Informatica (LM-32)

Testing e Validazione di Workflow Agentico a livello Enterprise

Progetto Agentic Platform

Relatore:

Prof. Simonini Giovanni

Candidato:

Davide Zanni

Matricola 196476

ANNO ACCADEMICO 2025/2026

Sommario

L'adozione aziendale dei modelli linguistici di grandi dimensioni (*Large Language Model*, LLM) incontra un problema centrale: integrarli in un sistema *enterprise* senza comprometterne l'affidabilità. Un LLM è flessibile ma probabilistico: allucina, sbaglia l'instradamento e produce risposte difficili da tracciare. Il contesto aziendale richiede invece decisioni deterministiche e verificabili. Per conciliare le due esigenze presentiamo Agentic Platform, un'architettura multi-agente realizzata durante il tirocinio presso E4 Analytics.

Lo sviluppo procede per quattro fasi incrementali, sulla stessa infrastruttura *cloud-native* Kubernetes. La prima è un prototipo sulla piattaforma *low-code* n8n, dove fissiamo il principio dell'architettura, la separazione tra *reasoning* ed *execution*: l'LLM pianifica e sintetizza, una logica deterministica esegue. Il prototipo presenta però sei criticità operative. Nella seconda fase riscriviamo l'intera architettura in Python con LangGraph. Il piano di *workflow* è tipizzato: la validazione lo respinge prima dell'esecuzione se richiama un agente inesistente. Nella terza fase migriamo sul *framework* agentico dichiarativo OpenCode; nella quarta sostituiamo il recupero vettoriale con la *Direct Corpus Interaction*, che espone il *corpus* all'agente come un *file system* da percorrere.

Le prove sperimentali procedono su due fronti. Il primo è un *benchmark* di *routing* su 100 *query* stratificate, di cui 11 avversarie. L'accuratezza di instradamento sale da 0,76 con LangGraph a 0,94 con OpenCode. Le due misure provengono da esecuzioni separate: il confronto è longitudinale e non appaiato.

Il secondo fronte confronta tre sistemi di recupero: un RAG classico, uno avanzato e un agente su *file system*. Li misuriamo con Ragas su SciFact, HotpotQA e MultiHop-RAG. Come giudice impieghiamo un modello di famiglia diversa da quella del generatore, per attenuare il *self-evaluation bias*. L'agente ottiene la qualità di recupero e sintesi migliore, a un costo per *query* quasi cinque volte superiore. Il RAG avanzato mantiene il vantaggio sulla copertura esaustiva.

Ne deriva un principio progettuale. Un percorso incrementale, guidato dalle misure, rende ispezionabile ogni passo del sistema: il piano è un oggetto tipizzato, validato prima dell'esecuzione e registrato insieme alla traccia. Sul risultato finale pesa più la disciplina con cui separiamo pianificazione ed esecuzione che la potenza del singolo modello.

Indice

Elenco delle Figure **iv**

Elenco delle Tabelle **v**

1 Introduzione **1**
1.1 Il contesto dell'innovazione agentica nei sistemi *enterprise* 1
1.2 Il percorso ingegneristico: quattro fasi verso un'architettura *enterprise-grade* . 2
1.3 La separazione tra *reasoning* ed *execution* 4
1.4 Obiettivi della ricerca 5
1.5 Struttura della tesi 6

2 Revisione della letteratura **7**
2.1 Agenti autonomi basati su LLM: da ReAct a Plan-and-Execute 7
2.2 Sistemi multi-agente: architetture, coordinazione e limiti 11
2.3 Architetture cognitive: il meta-modello CoALA 12
2.4 *Tool use*, *function calling* e protocolli di integrazione 13
2.5 Modular RAG: ottimizzazione del contesto e ancoraggio fattuale 14
2.6 Ottimizzazione del recupero: *chunking*, ricerca ibrida e *reranking* 18
2.7 Valutazione dei sistemi agentici: LLM-as-Judge e metriche complementari . . . 20
2.8 Orchestrazione a grafi e piattaforme *low-code* 22
2.9 Definizione del gap accademico e domanda di ricerca 24

3 Tecnologie e strumenti **27**
3.1 Motore inferenziale: i modelli Qwen e il *serving* vLLM 27
3.2 Pydantic come strato di validazione del piano 29
3.3 Orchestrazione deterministica con LangGraph 30
3.4 Model Context Protocol 31
3.5 Persistenza ibrida ed *embedding*: PostgreSQL, pgvector e bge-m3 31
3.6 Piattaforma di prototipazione *low-code*: n8n 32
3.7 *Framework* agentico dichiarativo: OpenCode 33
3.8 *Backend* RAG (pgvector) e ricerca *web* 34
3.9 Ragas ed *embedding* di valutazione 34
3.10 *Stack* di *retrieval* sperimentale: Qdrant, Chonkie e giudice Ragas 35

4 Metodologia della ricerca **37**
4.1 La cornice: Design Science Research 37
4.2 Operazionalizzazione delle domande di ricerca 38
4.3 Valutazione comparativa delle piattaforme *low-code* 38
4.4 Protocollo del *benchmark* di *routing* 40
4.5 Protocollo dello studio comparativo esteso: SciFact, HotpotQA e MultiHop-RAG 43
4.6 Riproducibilità 46

5 Architettura del sistema: orchestrazione deterministica (LangGraph) 47
5.1 Vista d'insieme dell'architettura 47
5.2 Separazione fra ragionamento ed esecuzione 49
5.3 La *Semantic Cache* 50
5.4 Il piano di *workflow*: il modello `WorkflowPlan` 51
5.5 Il grafo di stato: *Route Executor* e *Feedback Loop* 53
5.6 Gli agenti specializzati e la *Deep Research* 55
5.7 Osservabilità e contesto compatto 56
5.8 *Checkpointing* e *human-in-the-loop* 57
5.9 Sintesi: *trade-off* e limiti dell'architettura 57

6 Architettura v3: OpenCode e *Direct Corpus Interaction* 59
6.1 Motivazioni della migrazione a OpenCode (v3) 59
6.2 Architettura del sistema OpenCode 60
6.3 Agenti specializzati e modello di permessi (OpenCode) 63
6.4 *Custom tool* e integrazioni MCP (OpenCode) 66
6.5 Flussi operativi (OpenCode) 67
6.6 *Direct Corpus Interaction* e *pipeline* di *benchmark* di recupero (Agentic RAG) . 68
6.7 Le configurazioni di recupero del RAG avanzato 70
6.8 Sintesi 71

7 Risultati sperimentali: orchestrazione (LangGraph, OpenCode) e recupero (Agentic RAG) 72
7.1 Valutazione dell'architettura OpenCode (v3) 72
7.2 Studio comparativo esteso: RAG classico, avanzato e agente su *file system* . . . 79
7.3 Sintesi rispetto alla domanda di ricerca 92

8 Discussione dei risultati 95
8.1 Il *trade-off* tra determinismo e flessibilità adattiva 95
8.2 Il principio di restrizione dell'autonomia esecutiva 97
8.3 Analisi dei limiti e dei *trade-off* architetturali 98

9 Conclusioni e sviluppi futuri 102
9.1 Risposta alla domanda di ricerca 102
9.2 Contributi del lavoro 104
9.3 Direzioni future di ricerca 105
9.4 Implicazioni per i professionisti 107
9.5 Considerazioni finali 107

Bibliografia 109

Elenco delle Figure

1.1 Fasi evolutive del progetto Agentic Platform. 3
1.2 Architettura a tre stadi con separazione deterministica. 4

2.1 I quattro moduli dell'architettura CoALA. 12

5.1 Vista logica del flusso agentico di *Agentic Platform*. 49
5.2 Macchina a stati del *Feedback Loop*. 54

7.1 Routing benchmark: deterministico (LangGraph) *vs* dichiarativo (OpenCode). . 76
7.2 MRR dei tre sistemi sui tre dataset. 80
7.3 Medie di recupero dei tre sistemi sui tre dataset. 80
7.4 Sweep delle strategie di chunking. 81
7.5 Ricerca ibrida nativa con fusione RRF. 82
7.6 Reranking early e late su RAG-Fusion. 83
7.7 Sweep del pool pre-rerank e contributo dei componenti. 84
7.8 Configurazione di riferimento del RAG avanzato. 84
7.9 Scomposizione sequenziale della query sui dataset multi-hop. 85
7.10 Agente su file system dopo la correzione Anti-Hedge v2. 86
7.11 Trade-off costo/qualità sui tre dataset. 88
7.12 Medie di qualità del contesto e faithfulness. 88
7.13 Medie di sintesi generativa e accuratezza dei verdetti. 89
7.14 Agente su file system, DCI senza e con politica GRASP. 91
7.15 Quadro sinottico dei quattro sistemi di recupero. 92

Elenco delle Tabelle

1.1 Criticità strutturali della piattaforma *low-code*. 3

2.1 Paradigmi di ragionamento per agenti LLM. 8
2.2 Confronto qualitativo fra *framework* multi-agente. 11
2.3 Metriche di valutazione di una *pipeline* RAG per stadio. 16

3.1 Configurazione dei modelli LLM per fase. 28

4.1 Operazionalizzazione delle domande di ricerca. 39
4.2 Valutazione comparativa delle piattaforme *low-code*. 40
4.3 *Pattern* di delega del *benchmark* di *routing*. 41

5.1 Risposte architetturali della v2 alle criticità della Fase 1. 48
5.2 Latenza della ricerca coseno su pgvector. 51
5.3 Campi dei modelli Pydantic del piano. 52
5.4 Piano generato dal *Planner* per lo scenario B001. 53
5.5 Agenti specializzati dell'architettura v2. 55
5.6 *Trade-off* quantitativi tra v1 (n8n) e v2 (LangGraph). 57

6.1 Sforzo di manutenzione, v2 *vs* v3. 60
6.2 *Endpoint* del *Bridge* FastAPI. 62
6.3 Matrice degli agenti della versione OpenCode. 63
6.4 *Custom tool* TypeScript del *runtime* OpenCode. 66

7.1 Condizioni sperimentali del confronto di *routing*. 73
7.2 Benchmark di *routing* deterministico, LangGraph *vs* OpenCode. 74
7.3 Analisi qualitativa di tre errori di *routing* della v2. 77
7.4 Metriche di recupero su HotpotQA-100. 79
7.5 MRR di sei varianti del RAG avanzato sui tre dataset. 82
7.6 Scomposizione della *query* contro RAG-Fusion. 85
7.7 Griglia *answer* su SciFact-100. 86
7.8 Griglia *answer* su HotpotQA-100 e MultiHop-RAG-100. 87
7.9 Agente su *file system*, DCI senza e con politica GRASP. 90
7.10 Evidenze per dimensione della domanda di ricerca. 93

8.1 Mappa fra l'architettura cognitiva di Sumers et al. e *Agentic Platform*. 98

9.1 Verifica delle tre dimensioni della domanda di ricerca. 104

1. Introduzione

1.1 Il contesto dell'innovazione agentica nei sistemi *enterprise*

Integrare un *Large Language Model* (LLM) in un processo aziendale reale richiede più del modello. Serve un'infrastruttura di esecuzione con separazione degli ambienti e persistenza dello stato. Servono *pattern* di progettazione già consolidati e una strategia di migrazione incrementale.

La letteratura attribuisce agli LLM capacità di ragionamento multi-passo, di pianificazione e di invocazione di strumenti esterni [58, 57, 48]. Su queste capacità poggiano forme di orchestrazione orientate all'intento [55, 61]. Adottiamo la definizione di Wang et al. [55]: un sistema agentico basato su LLM impiega un modello linguistico pre-addestrato per interpretare il contesto, pianificare azioni, invocare strumenti e mantenere uno stato informativo utile al raggiungimento di un obiettivo.

In produzione questi sistemi devono soddisfare quattro requisiti: affidabilità, trasparenza decisionale, auditabilità e integrazione con i sistemi *legacy*. La natura probabilistica degli LLM entra in conflitto con tutti e quattro [61, 19]. La letteratura discute i *framework* multi-agente e il *pattern* Plan-and-Execute, ma non documenta per intero un caso reale. Manca cioè un percorso di progettazione e validazione condotto dal prototipo al *deployment* entro un singolo contesto organizzativo (*gap* formalizzato nel Capitolo 2).

Rispondiamo a questo *gap* con il nostro progetto, **Agentic Platform**, sviluppato durante il tirocinio presso E4 Analytics. Si tratta di un'architettura agentica realizzata in quattro fasi: prototipazione *low-code* su n8n, orchestrazione deterministica su LangGraph, transizione al *framework* dichiarativo OpenCode e sostituzione del recupero vettoriale con la *Direct Corpus Interaction* (DCI), che legge il *corpus* direttamente dal *file system*. L'ultima fase è validata con il *benchmark* Ragas. Le quattro fasi poggiano su una stessa infrastruttura *cloud-native* basata su Kubernetes. Formuliamo la domanda di ricerca. La riprendiamo nella Sezione 2.9, insieme ai *gap* che la motivano:

> **RQ.** *In che misura la separazione formale fra* reasoning *ed* execution*, adottata come invariante architetturale, riduce gli errori di instradamento verso gli agenti rispetto a un'orchestrazione che lascia decisione ed esecuzione nello stesso passo? Misuriamo*

la riduzione su un banco di 100 query *annotate, con accuratezza di* routing *e F1 per agente.*

Scomponiamo la RQ in tre sotto-domande, riprese al termine della revisione della letteratura (Sezione 2.9):

RQ1 (formalismo). *Quale formulazione operativa della separazione* reasoning/execution*, fra piano di workflow tipizzato, grafo di stato e matrice di permessi, riduce gli errori di instradamento rispetto a un orchestratore che decide ed esegue nello stesso passo? Confrontiamo le formulazioni su accuratezza di* routing *e tasso di allucinazione di strumento.*

RQ2 (processo). *A quali condizioni un percorso incrementale a quattro fasi porta un prototipo agentico fino a un* deployment enterprise *con vincoli di auditabilità e* compliance*? Consideriamo la condizione soddisfatta quando ogni transizione di fase poggia su un'evidenza misurata nella fase precedente e l'artefatto finale risulta in esercizio sull'infrastruttura aziendale.*

RQ3 (sostrato). *Cosa si guadagna e cosa si perde passando da piattaforma* low-code *(n8n) a orchestrazione imperativa (LangGraph) e infine a* framework *dichiarativo (OpenCode)? Valutiamo il* trade-off *su affidabilità di instradamento e onere di manutenzione dei piani tipizzati.*

1.2 Il percorso ingegneristico: quattro fasi verso un'architettura *enterprise-grade*

Le fasi sono quattro e ciascuna deriva dalle evidenze raccolte nella precedente (Figura 1.1). L'infrastruttura *cloud-native* basata su Kubernetes resta comune a tutte, e non conta come fase a sé. Fornisce a ogni fase un ambiente di esecuzione ripetibile: *namespace* separati per sviluppo e produzione, *data layer* persistente su volumi dedicati, *networking* con bilanciamento del carico.

Nella prima fase costruiamo un prototipo *low-code* sull'infrastruttura Kubernetes già predisposta. Confrontiamo n8n, Dify e Langflow (Tabella 4.2 del Capitolo 4) e selezioniamo n8n. Su questa piattaforma implementiamo il prototipo *Planner-Executor*, che separa la pianificazione dall'esecuzione tramite un piano JSON processato in modo deterministico. Su 50 *query* di test rileviamo però sei criticità strutturali (Tabella 1.1; analisi completa nel Capitolo 5).

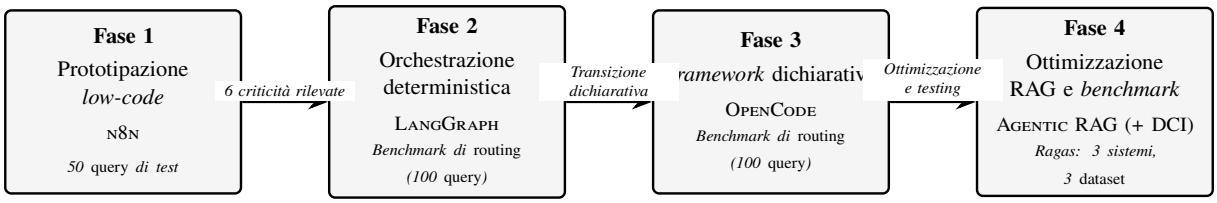


Figura 1.1: Le quattro fasi del progetto Agentic Platform. Ogni transizione poggia sulle criticità misurate nella fase precedente: dal prototipo *low-code* all'orchestrazione su LangGraph, poi al *framework* dichiarativo OpenCode, infine al recupero con *Direct Corpus Interaction* (DCI) validato su Ragas.

Le individuiamo incrociando due sorgenti: l'analisi sistematica dei log di esecuzione e l'ispezione manuale dei *workflow*. Le due sorgenti hanno peso diverso. Solo la prima criticità è quantificata sul campione di test, con 6 *query* su 50 terminate senza risultato utile. Le altre cinque emergono dall'ispezione dei *workflow* e le riportiamo in forma qualitativa, come motivazione documentata della riscrittura. Riguardano tutte requisiti di produzione che la piattaforma *low-code* non garantisce nativamente. Il *pattern Planner-Executor* supera invece la verifica sperimentale.

Tabella 1.1: Le sei criticità strutturali identificate su n8n nella Fase 1.

#	Criticità	Descrizione sintetica
1	Loop di esecuzione non controllati	Cicli privi di condizione di uscita esplicita: 6 *query* su 50 terminavano con *workflow* incompleto e nessun percorso di recupero
2	Gestione non ottimale del contesto RAG	Invio al modello di contenuti non sufficientemente filtrati o gerarchizzati
3	Scarsa trasparenza del processo decisionale	Difficoltà nel ricostruire e mostrare all'utente i criteri di *routing* e selezione delle azioni
4	Assenza di *caching* semantico	Mancanza di meccanismi per il riuso di risultati su *query* semanticamente equivalenti
5	Mancanza di aggiornamento incrementale verso il client	Assenza di meccanismi per comunicare progressivamente stato e avanzamento dell'elaborazione
6	Assenza di persistenza del contesto utente	Mancato mantenimento di preferenze e stato informativo tra sessioni

Queste criticità sono limiti generali dell'approccio *low-code*, e motivano la seconda fase. Il determinismo è già acquisito su n8n: qui lo portiamo a livello *enterprise*, in codice Python. Riscriviamo l'architettura su LangGraph [24] per ottenere controllo esplicito sul *routing*, esecuzione parallela dei nodi indipendenti e tracciamento del processo decisionale. Il prototipo n8n esaurisce così la funzione di *proof-of-concept*: ha validato i *pattern* architetturali prima della riscrittura.

Nella terza fase riscriviamo nuovamente l'architettura. Passiamo da una definizione imperativa, il grafo LangGraph in Python, a una definizione dichiarativa basata sul *framework* OpenCode (v3). Il *prompt* è scritto in Markdown, il *tool* in TypeScript, e il recupero RAG resta su pgvector.

Conserviamo la separazione *reasoning/execution*, ma ne spostiamo l'*enforcement* dal *Route Executor* Python a una matrice di permessi dichiarativi, che vincola l'*orchestrator* a delegare ai subagenti specializzati.

Nella quarta fase, il progetto *Agentic RAG*, sostituiamo l'interrogazione di un *vector store* esterno con la lettura diretta del *file system*, concessa all'agente di *Agentic File Search* tramite permessi nativi. La *Direct Corpus Interaction* rimpiazza la *similarity search* parametrica con un recupero testuale deterministico. Validiamo il recupero con uno studio comparativo: mettiamo l'agente a confronto con un RAG classico e uno avanzato su tre dataset, tramite il *framework* Ragas. Il giudice appartiene a una famiglia di modelli diversa da quella del generatore, scelta che attenua il *self-evaluation bias* (Capitolo 3).

1.3 La separazione tra *reasoning* ed *execution*

La separazione tra pianificazione ed esecuzione attraversa tutte e quattro le fasi del progetto. Il principio risale a STRIPS [11] e alla pianificazione automatica classica [14]. Applicato agli LLM, confina la componente probabilistica alla sola pianificazione e affida l'esecuzione a procedure deterministiche e verificabili. Nella Fase 1 lo adottiamo per necessità, quando l'agente monolitico su n8n ne mostra i limiti.

Quando un singolo componente opaco produce direttamente l'azione, ragionamento ed esecuzione si confondono. Ne deriva opacità decisionale, e con essa l'allucinazione operativa: il modello seleziona uno strumento inesistente o ne sbaglia i parametri. Patil et al. [38] documenta il fenomeno sulle chiamate API, e lo ritroviamo nel nostro prototipo *low-code* (Capitolo 7). La riproducibilità ne risente in modo diretto [55]. Rispondiamo separando il ciclo di vita della richiesta in tre stadi funzionalmente indipendenti, collegati da uno schema strutturato e serializzato in JSON (Figura 1.2).

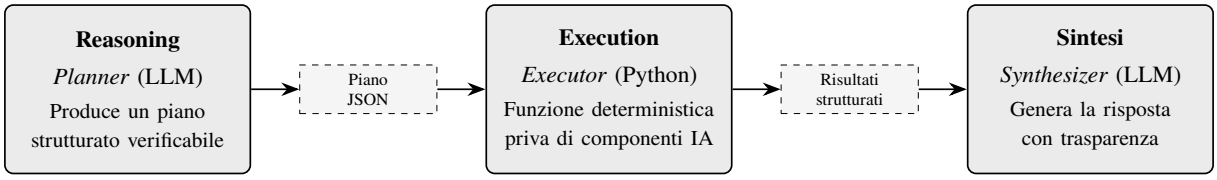


Figura 1.2: I tre stadi dell'architettura con separazione deterministica. Il piano JSON funge da meccanismo di disaccoppiamento: il *Planner* produce solo il piano; l'*Executor* esegue solo funzioni deterministiche. Si noti che il determinismo dell'*Executor* è garantito a parità di stato delle dipendenze esterne (database, API); la gestione dei casi di non-idempotenza è descritta nel Capitolo 5.

Nello stadio di *Reasoning* il *Planner* produce un piano strutturato. Nell'*Execution* l'*Executor*, Python deterministico, lo esegue e isola gli errori di pianificazione da quelli esecutivi. Nella

Sintesi il *Synthesizer* genera la risposta finale ed esplicita le azioni compiute. Lo schema realizza il *pattern* **Plan-and-Execute** [56], adatto agli scenari che richiedono prevedibilità. Per la stessa ragione Sumers et al. [50] isola la *decision procedure* come componente distinto. Il piano tipizzato, il *routing* e gli agenti specializzati sono trattati formalmente nel Capitolo 5.

1.4 Obiettivi della ricerca

Cinque obiettivi traducono la domanda di ricerca della Sezione 1.1 in altrettanti risultati attesi:

1. **Infrastruttura *enterprise-grade***: *cluster* Kubernetes con separazione ambienti, persistenza e *scaling* orizzontale come infrastruttura trasversale alle quattro fasi [22];

2. **Prototipazione e analisi criticità**: prototipo *low-code* (n8n) per validare i *pattern* e identificare le criticità strutturali;

3. **Architettura *production-grade***: sistema multi-agente su LangGraph con separazione *reasoning*/*execution* come *pattern* centrale;

4. **Transizione al paradigma dichiarativo**: reimplementazione dell'architettura sul *framework* dichiarativo OpenCode (v3), spostando l'*enforcement* della separazione da codice imperativo a una matrice di permessi dichiarativi;

5. **Validazione empirica**: due campagne distinte. La prima confronta l'instradamento di LangGraph e OpenCode su 100 *query* annotate. La seconda valuta il recupero, mettendo a confronto un RAG classico, uno avanzato e un agente su *file system* con *Direct Corpus Interaction* su tre dataset, con *framework* Ragas e un giudice appartenente a una famiglia di modelli diversa dal generatore (Agentic RAG).

Il lavoro porta cinque contributi. Sul piano ingegneristico, una piattaforma agentica in esercizio su Kubernetes, documentata dal prototipo al *deployment*. Sul piano architetturale, la separazione *reasoning*/*execution*, portata da vincolo di codice imperativo a matrice di permessi dichiarativi. Il contributo metodologico è un banco di 100 *query* annotate, che rende confrontabili due orchestratori sulla stessa base. Sul piano comparativo, la misura di cosa si guadagna e cosa si perde in tre transizioni di sostrato. Sul piano operativo, una politica di lettura mirata che riduce il costo del recupero agentico su documenti lunghi. Le evidenze a sostegno arrivano nel Capitolo 9 (Sezione 9.2).

1.5 Struttura della tesi

Il lavoro adotta il **Design Science Research** (DSR) [17, 39]: l'artefatto tecnologico è insieme prodotto della ricerca e strumento per generare conoscenza. Lo stato dell'arte e la formalizzazione del *gap* sono nel Capitolo 2. Segue nel Capitolo 3 lo *stack* tecnologico adottato. Il Capitolo 4 dettaglia il disegno sperimentale delle quattro fasi, dai criteri di valutazione delle piattaforme *low-code* alle metriche impiegate nella validazione finale.

I Capitoli 5 e 6 documentano l'architettura in ordine cronologico: dal *Planner*/*Executor*/*Synthesizer* su LangGraph (v2) alla migrazione dichiarativa OpenCode (v3) e alla *Direct Corpus Interaction* (Agentic RAG). Il Capitolo 7 riporta la validazione empirica di entrambe le architetture e dei *benchmark* Ragas. Il Capitolo 8 discute i *trade-off* emersi dal confronto fra le fasi. Chiude il Capitolo 9, con contributi, limitazioni e direzioni future.

2. Revisione della letteratura

Integrare un sistema agentico basato su LLM in un contesto *enterprise* solleva un problema ricorrente. Il modello è probabilistico per natura, mentre l'azienda richiede prevedibilità, tracciabilità, controllo del flusso e *compliance*. La letteratura propone numerose architetture agentiche, ma ne documenta solo in parte la trasferibilità a contesti reali. In questo capitolo ricostruiamo i principali orientamenti della ricerca e formalizziamo il *gap* che motiva il lavoro (Sezione 2.9).

Il percorso del capitolo segue il ciclo di vita di un agente. Partiamo dai **paradigmi di ragionamento** di un agente singolo (Sezione 2.1) e dai **sistemi multi-agente** che ne coordinano più di uno (Sezione 2.2). Segue una cornice teorica unificante, il meta-modello **CoALA** (Sezione 2.3), e poi le capacità operative: l'uso di **strumenti esterni** (Sezione 2.4) e il recupero di informazioni con il **RAG** (Sezione 2.5).

Una sezione dedicata raccoglie le tecniche con cui si ottimizza il recupero, dalla segmentazione dei documenti alla riscrittura della domanda fino al riordino dei candidati (Sezione 2.6). Quel lessico è un prerequisito per interpretare i risultati del Capitolo 7.

Chiudono la **valutazione** degli agenti (Sezione 2.7) e i **sostrati di orchestrazione** con cui costruirli (Sezione 2.8).

2.1 Agenti autonomi basati su LLM: da ReAct a Plan-and-Execute

Wooldridge e Jennings [59] definiscono un agente come un'entità software con quattro proprietà. L'agente opera senza supervisione costante (*autonomia*) e risponde ai cambiamenti dell'ambiente (*reattività*). Prende inoltre l'iniziativa verso un obiettivo, invece di limitarsi a reagire (*proattività*), e interagisce con altri agenti o con l'utente (*abilità sociale*). Per rappresentare lo stato interno, la tradizione propone il modello BDI (*Belief-Desire-Intention*), che descrive l'agente come la terna $\langle B, D, I \rangle$. Le credenze B sono ciò che l'agente ritiene vero sul mondo, i desideri D gli obiettivi da raggiungere, le intenzioni I i piani che ha deciso di perseguire.

Un **agente autonomo basato su LLM** eredita questa concezione, ma ne ridefinisce due componenti costitutive. Il motore di ragionamento diventa il completamento testuale di un

modello addestrato su grandi quantità di dati [55, 61], al posto di una procedura simbolica scritta a mano. Lo stato interno, a sua volta, si riduce al testo libero del *prompt* corrente, al posto della terna BDI tipizzata.

Nei sistemi classici la transizione tra stati è deterministica: dati lo stato s_t e l'azione a_t, il nuovo stato $s_{t+1} = f(s_t, a_t)$ è univocamente determinato. Nell'agente LLM la stessa transizione diventa probabilistica, perché dipende da come il modello campiona le parole successive. A parità di input, due esecuzioni possono divergere e la prevedibilità *a priori* si perde. Questo problema orienta l'intera revisione.

I paradigmi agentici sono preceduti dal *Chain-of-Thought* [58]. Invece della sola risposta finale, il modello esplicita i passaggi intermedi del ragionamento. Sui compiti complessi la qualità migliora e il ragionamento diventa ispezionabile. Da qui la domanda successiva: come si trasforma questo ragionamento in azione sul mondo? La letteratura risponde con i paradigmi di ragionamento agentico, che differiscono per il modo in cui combinano pianificazione, esecuzione e revisione dell'errore (Tabella 2.1).

Tabella 2.1: Confronto tra i principali paradigmi di ragionamento per agenti LLM. La colonna "Rif. proposta" rinvia all'articolo che introduce il paradigma; la colonna "Rif. critica" a lavori che ne documentano i limiti.

Paradigma	**Principio**	**Punti di forza**	**Limiti**	**Rif. proposta**	**Rif. critica**
ReAct	Co-generazione di ragionamento e azione	Adattività in scenari aperti e non strutturati	Saturazione del contesto, ridotta prevedibilità *ex ante*	[64]	[29, 55]
Plan-and-Execute	Separazione tra piano ed esecuzione	Tracciabilità, controllo, auditabilità del piano	Rigidità del piano iniziale in contesti dinamici	[56, 57]	[55]
Reflexion	Auto-valutazione linguistica e revisione iterativa	Correzione senza aggiornamento dei pesi	Costo computazionale e maggiore latenza	[49]	[55]

Il paradigma **ReAct** (*Reasoning and Acting*) [64] intreccia pensiero e azione in un unico flusso: l'agente ragiona su cosa fare, esegue un passo, osserva il risultato e ragiona di nuovo. In termini formali, fonde il ragionamento r_t e l'azione a_t nella stessa generazione. Yao et al. [64] misurano +34 punti percentuali di *success rate* assoluto su ALFWorld rispetto ai metodi di *imitation* e *reinforcement learning*.

La fusione tra ragionamento e azione presenta però due limiti in contesto *enterprise*. Il primo è la lunghezza: a ogni passo la traiettoria si allunga, riempie il contesto e fa emergere il fenomeno *lost-in-the-middle*, per cui il modello trascura le informazioni collocate al centro della finestra [29]. Il secondo è l'assenza di un piano separato e ispezionabile, perché ragionamento ed

esecuzione restano mescolati. Un errore di ragionamento si traduce così immediatamente in una scelta di *routing* errata, difficile da prevedere e da controllare prima che si verifichi [38, 55].

Il paradigma **Plan-and-Execute** ribalta questa logica e separa il momento della decisione da quello dell'azione. L'agente costruisce dapprima un piano completo, poi componenti distinte e più semplici lo eseguono passo per passo. L'idea deriva dall'AI *planning* classico, dal sistema STRIPS [11] alla tradizione dell'*automated planning* [14]. In quella tradizione un pianificatore cerca una sequenza di azioni che conduca da uno stato iniziale a un obiettivo. La novità consiste nell'applicarla agli agenti LLM, come fa Wang et al. [56]: il piano è generato dal modello in linguaggio strutturato, e il risolutore simbolico viene meno.

Il vantaggio principale è che il piano diventa un artefatto verificabile prima dell'esecuzione. Su questa proprietà si fonda l'architettura della tesi. Il prezzo è una minore flessibilità: con informazioni incomplete o mutevoli durante l'esecuzione, un piano deciso in anticipo si adatta con più difficoltà.

Una terza via, **Reflexion** [49], aggiunge una capacità ulteriore: apprendere dai propri errori senza aggiornare i pesi. Dopo un tentativo fallito, l'agente formula a parole l'origine dell'errore e conserva questa riflessione come memoria, riutilizzabile al tentativo successivo. Gli autori la chiamano *verbal reinforcement*. Il costo cresce con il numero di tentativi. Anche qui il ciclo resta interno alla generazione del modello, privo di una separazione netta tra chi decide e chi esegue.

I tre paradigmi rispondono alla stessa domanda in modi diversi: ReAct privilegia l'adattabilità, Reflexion l'apprendimento dagli errori, Plan-and-Execute la prevedibilità. La scelta dipende dal contesto. Dove servono libertà e reattività, un ciclo stretto tra ragionamento e azione risulta preferibile. Dove sono richieste tracciabilità e controllo, come in azienda, la preferenza si sposta verso la separazione tra piano ed esecuzione: il piano è l'unico punto in cui si può intervenire prima che l'azione avvenga. È la direzione che adottiamo nell'architettura descritta dal Capitolo 5.

2.1.1 Dall'autocompletamento agli agenti autonomi di software engineering

L'ingegneria del software offre un caso d'uso empirico per la traiettoria appena descritta. L'evoluzione degli agenti applicati alla scrittura di codice attraversa quattro stadi, e ciascuno deriva dai limiti del precedente.

Nel primo stadio il modello completa il codice locale, *token* per *token*, dentro l'editor: è l'autocompletamento *inline* delle prime versioni di GitHub Copilot [15]. La generazione resta

circoscritta al frammento visibile. Il sistema non conserva memoria del progetto e non pianifica alcuna sequenza di modifiche.

Nel secondo stadio l'interazione diventa conversazionale. Il programmatore inserisce porzioni di codice nel *prompt* di una chat integrata nell'ambiente di sviluppo e chiede spiegazioni o *refactoring*. Il limite è di nuovo strutturale: la persona resta l'unico tramite fra la generazione del modello e la sua esecuzione, perché copia manualmente l'output nell'editor e ne verifica l'esito eseguendo i test.

Nel terzo stadio il modello ottiene accesso diretto agli strumenti: legge e scrive file, esegue comandi di *shell*, compila il progetto ed esegue i test. Si definiscono così i ***coding agent*** autonomi a ciclo singolo, che riprendono lo schema ReAct e lo applicano a un ambiente di sviluppo. SWE-agent [62] ne è un esempio documentato: gli autori progettano un'interfaccia agente-computer con comandi dedicati alla navigazione e alla modifica del codice. Jimenez et al. [20] valutano questi sistemi su SWE-bench, un banco di 2.294 problemi reali estratti da dodici *repository* Python su GitHub.

Emerge qui il limite principale del paradigma: l'accumulo iterativo di *log* di errore, output di terminale e *diff* di codice nella stessa finestra satura il contesto e disperde l'attenzione sui vincoli iniziali [29]. Il ciclo di correzione tende allora a non convergere, con allucinazioni ricorrenti sulle firme delle funzioni.

Il quarto stadio risponde a questo limite con la separazione dei ruoli. Il compito si distribuisce tra componenti specializzate, ciascuna con una finestra di contesto isolata: un *Architect* analizza i requisiti e produce il piano di modifica, un *Coder* implementa interventi circoscritti su un singolo file, un *Tester* esegue i test unitari e valida l'esito prima del *commit*. MetaGPT [18] formalizza questa divisione tramite procedure operative standard assegnate a ruoli distinti. La saturazione della memoria di lavoro si riduce, perché nessuna componente eredita la cronologia completa dei tentativi precedenti.

La traiettoria conferma, in un dominio applicativo concreto, l'argomento sostenuto in questo capitolo. Un ciclo probabilistico non strutturato resta efficace finché il compito è breve, poi degrada al crescere del contesto accumulato. Un flusso strutturato con separazione esplicita fra piano ed esecuzione contiene quel degrado, perché isola ogni fase in un contesto proprio e sottopone il piano a validazione deterministica prima dell'azione. È lo stesso principio che il Capitolo 5 applica alla nostra architettura.

2.2 Sistemi multi-agente: architetture, coordinazione e limiti

L'impiego di più agenti in luogo di uno solo definisce un **sistema multi-agente** (*Multi-Agent Systems*, MAS), estensione naturale dell'agente singolo. Più entità specializzate cooperano sullo stesso compito, ciascuna con un ruolo preciso, nell'ipotesi che la divisione del lavoro migliori la qualità delle decisioni [61, 55]. I *framework* multi-agente differiscono fra loro per tre aspetti: come gli agenti si coordinano, come gestiscono lo stato condiviso, quanto controllo concedono sul flusso di esecuzione. La Tabella 2.2 li confronta in modo qualitativo lungo queste dimensioni, senza pretesa di misura assoluta.

Tabella 2.2: Confronto qualitativo tra i principali *framework* multi-agente rispetto a requisiti rilevanti in contesti *enterprise*.

Framework	**Orchestrazione**	***Type safety***	**Determinismo**	**Auditabilità**	**Grafi ciclici**	**Rif.**
AutoGen	Conversazionale	Bassa–Media	Basso	Bassa–Media	Limitati	[60]
MetaGPT	SOP prescrittive	Media–Alta	Medio	Media	No	[18]
CAMEL	*Role-playing*	Bassa	Basso	Bassa	No	[27]
CrewAI	Gerarchica	Media	Medio	Media	Limitati	[7]

Nota: classificazione qualitativa da proprietà architetturali dichiarate in letteratura e documentazione tecnica. Attribuiamo *type safety* alta quando lo stato condiviso è definito da uno schema validato a *runtime*, media quando la validazione copre le sole firme dei messaggi, bassa quando lo stato transita come testo non tipizzato. **LangGraph** è volutamente escluso, perché appartiene a un'altra categoria: è un motore di orchestrazione a grafi di stato, trattato nella Sezione 2.8.

I quattro *framework* adottano strategie di coordinazione distinte. **AutoGen** [60] modella la collaborazione come una conversazione: ogni agente ha un ruolo e il compito si risolve attraverso lo scambio di messaggi, con la possibilità di inserire l'utente nello scambio. È molto flessibile, ma il flusso emerge dal dialogo e resta difficile da vincolare in anticipo. **MetaGPT** [18] segue la direzione opposta. Codifica procedure operative standard (*Standard Operating Procedure*) ispirate a un'azienda software e assegna a ciascun agente un ruolo professionale, dall'analista allo sviluppatore al revisore. La struttura è più rigida, ma anche più prevedibile e ripetibile.

CAMEL [27] studia la cooperazione tramite *role-playing*: due agenti, uno che istruisce e uno che esegue, si scambiano messaggi in autonomia per portare a termine il compito. Gli autori mostrano che ruoli ben definiti favoriscono la convergenza. **CrewAI** [7] organizza infine gli agenti in una *crew* con coordinamento gerarchico e delega esplicita dei compiti, e privilegia la semplicità d'uso rispetto al controllo fine del flusso.

2.2.1 Limiti trasversali

I *framework* della Tabella 2.2 si coordinano in modi diversi, ma condividono un limite di fondo. In tutti, il confine tra la **parte probabilistica** (il modello che ragiona) e la **parte deterministica** (il codice che esegue) resta sfumato: lo stesso agente, di norma, decide e agisce insieme. Le conseguenze sono tre: la prevedibilità *a priori* si riduce, la validazione delle decisioni arriva tardi, il nesso fra input, stato risultante e azione compiuta diventa difficile da ricostruire a posteriori. Le rassegne disponibili documentano risultati sperimentali su compiti circoscritti [55, 61], ma non riportano garanzie architetturali su questo confine.

2.3 Architetture cognitive: il meta-modello CoALA

Il meta-modello *Cognitive Architectures for Language Agents* (**CoALA**) di Sumers et al. [50] riconduce i diversi agenti linguistici a una tassonomia unitaria. Rilegge la tradizione delle architetture cognitive alla luce delle proprietà degli LLM e organizza l'agente in quattro moduli principali.

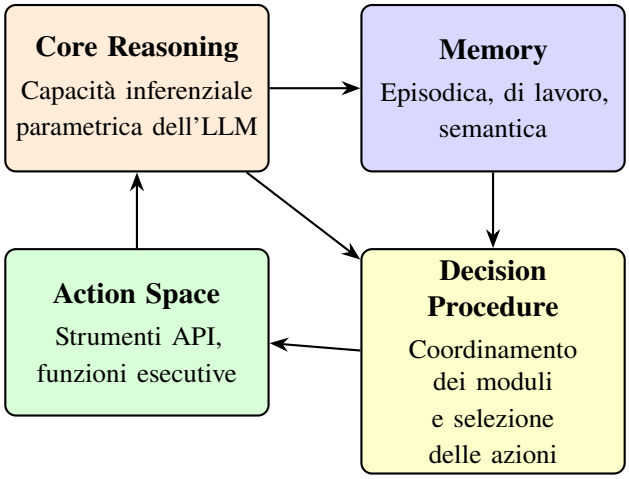


Figura 2.1: I quattro moduli dell'architettura CoALA [50]. La *decision procedure* coordina l'interazione tra gli altri moduli e può ricevere input sia dalla memoria sia direttamente dal *core reasoning*.

I quattro moduli (Figura 2.1) coprono le funzioni essenziali di un agente. Il *core reasoning* è la capacità di ragionamento del modello. La *memory* conserva le informazioni su orizzonti diversi. L'*action space* raccoglie gli strumenti e le funzioni che l'agente può invocare. La *decision procedure* coordina gli altri tre e seleziona l'azione da compiere.

Sumers et al. [50] rilevano che in molti agenti odierni la *decision procedure* resta nascosta dentro il completamento testuale del modello. Il modello decide così autonomamente quale strumento usare, quando e come proseguire, senza costituire un componente a sé. La scelta

semplifica il disegno iniziale, ma lega il controllo del flusso a un processo probabilistico, sensibile al campionamento e alla formulazione del *prompt*. Quando invece portiamo la *decision procedure* fuori dal modello, o la vincoliamo con strutture formali, il comportamento dell'agente diventa più trasparente e più facile da verificare.

Tra i quattro moduli, la *memory* separa un *chatbot* da un agente. Gli assistenti più semplici non conservano alcuno stato al termine della sessione. Architetture come i *Generative Agents* di Park et al. [37] mostrano invece il valore di una memoria persistente: le esperienze vengono registrate in un flusso, richiamate in base a pertinenza, *recency* e importanza, e periodicamente sintetizzate in riflessioni di livello superiore. CoALA generalizza questa idea e distingue tre orizzonti: la memoria di lavoro, volatile e legata al compito corrente; la memoria episodica, che conserva le interazioni passate; la memoria semantica, che raccoglie conoscenza stabile e condivisa tra sessioni.

La cornice di CoALA rende quindi esplicito dove collocare il processo decisionale e con quali vincoli, prima ancora di scegliere quale LLM adottare. Non prescrive un'implementazione.

2.4 *Tool use*, *function calling* e protocolli di integrazione

Un agente LLM è utile in produzione solo se può agire su strumenti esterni. Senza accesso a funzioni, API, basi di dati o servizi, produce solo testo: può descrivere come si prenota un volo, ma non prenotarlo. La capacità di invocare strumenti, il *tool use*, trasforma il ragionamento in azione.

Toolformer [48] mostra che un modello apprende quando e come invocare le API, integrando le chiamate nel processo generativo, ma non affronta la validazione formale delle chiamate prodotte. **Gorilla** [38] documenta un problema concreto: i modelli generici producono spesso chiamate sintatticamente plausibili ma semanticamente non valide, con API inesistenti o parametri non conformi. In produzione il rischio è critico: l'errore si traduce in un'azione operativa indesiderata, non solo in testo scorretto.

Per arginare questi errori, i principali fornitori di modelli hanno introdotto il *function calling* e lo *structured output*: il modello emette dati conformi a uno schema predefinito, in luogo del testo libero. Lo schema dichiara i nomi dei parametri, i loro tipi e quali sono obbligatori. Per ottenere un meteo, ad esempio, il modello emette un oggetto con i campi `city` (stringa) e `unit` (un valore tra "celsius" e "fahrenheit"), al posto di una frase. Il *constrained decoding* irrigidisce

ulteriormente il vincolo, perché forza la generazione a rispettare lo schema *token* per *token*.

La garanzia che ne otteniamo è sintattica: l'ambiguità del *parsing* a *runtime* sparisce, ma la correttezza semantica dei valori resta da verificare a valle. Il guadagno riguarda comunque il piano operativo: l'output diventa un artefatto software validabile prima dell'esecuzione.

Uno schema obbligatorio introduce un **contratto** formale tra lo strato inferenziale e quello esecutivo, nello spirito del *Design by Contract* [32]. Il contratto fissa due elementi: quali argomenti devono arrivare in ingresso e in che forma deve presentarsi l'output. Il confine tra "ciò che il modello propone" e "ciò che il sistema esegue" diventa così un punto di controllo verificabile. Ne derivano tre effetti: gli argomenti vengono validati prima dell'esecuzione, l'interpretazione del testo libero esce dal percorso critico, il processo decisionale diventa ispezionabile a posteriori. Nel Capitolo 7 misuriamo il primo effetto sul tasso di allucinazione di strumento.

Resta un problema pratico: ogni strumento, finora, richiede un'integrazione *ad hoc*. Un ulteriore sviluppo affronta proprio questo aspetto, il **Model Context Protocol** (MCP) [1]. MCP standardizza l'interfaccia tra agenti e fonti esterne tramite JSON-RPC 2.0 e tre primitive: i *Tools* sono le azioni che l'agente può invocare, le *Resources* i dati che può leggere, i *Prompts* i modelli di interazione riusabili. L'effetto è analogo a quello che USB ha avuto per le periferiche. Uno strumento esposto via MCP è utilizzabile da qualunque agente compatibile, senza integrazioni su misura.

Nel *tool use* la separazione fra generazione linguistica ed esecuzione si concretizza in un artefatto verificabile, lo schema della chiamata.

2.5 Modular RAG: ottimizzazione del contesto e ancoraggio fattuale

Un modello linguistico conosce solo ciò che ha osservato durante l'addestramento. In un dominio aperto, o su dati che cambiano nel tempo, questa conoscenza non basta e diventa presto obsoleta. La *Retrieval-Augmented Generation* (RAG) risponde al limite con un'idea semplice: prima di rispondere, il sistema recupera i documenti pertinenti da una fonte esterna e li passa al modello come contesto [26, 13]. Restano però aperti due problemi: la qualità di ciò che viene recuperato e l'uso che il modello ne fa davvero [13].

Il paradigma **Modular RAG** [13] affronta il problema scomponendo la *pipeline* in moduli ottimizzabili separatamente: *pre-retrieval* (riformulazione della domanda), *retrieval* vettoriale

(la ricerca vera e propria), *post-retrieval* (il reranking dei risultati) e generazione. Il reranking risponde in particolare al fenomeno *lost-in-the-middle* [29]: se l'informazione rilevante finisce a metà del contesto invece che in cima, l'accuratezza cala da circa il 76% al 56%. La Tabella 2.3 riepiloga le metriche di valutazione per stadio.

Come si passa da un documento a un vettore confrontabile? Il testo della fonte viene segmentato in frammenti (*chunk*) e ogni frammento è convertito in un vettore numerico da un modello di *embedding*, come BGE-M3 [3], che colloca vicini nello spazio i testi semanticamente affini. Alla domanda dell'utente associamo un vettore dello stesso tipo e cerchiamo i frammenti più vicini.

Su collezioni grandi una ricerca esatta sarebbe troppo lenta. Si impiegano allora indici approssimati come HNSW [31], che individuano i vicini più prossimi in tempo sublineare, con una perdita minima di accuratezza. Questa catena (*chunking*, *embedding*, indice vettoriale) spiega perché Gao et al. [13] distinguono tre generazioni di RAG: *naive* (recupera e concatena), *advanced* (aggiunge pre- e post-elaborazione del recupero) e *modular* (compone moduli riconfigurabili e iterabili).

Le metriche in uso hanno statuti diversi. Quelle classiche di recupero si calcolano contro relazioni di rilevanza note e prescindono dal modello generativo. Quelle sulla qualità del contesto e sulla generazione richiedono invece un giudice, umano o LLM, oppure una risposta di riferimento.

In produzione le metriche operative trasversali (latenza per stadio, costo in *token*, dimensione dell'indice) vincolano il *trade-off* tra ricchezza del contesto e costo inferenziale. Nel nostro lavoro il RAG svolge anche una seconda funzione, l'**ancoraggio fattuale della pianificazione**: restringe il dominio delle ipotesi plausibili su cui costruiamo il piano.

2.5.1 RAG agentico

Nel Modular RAG il recupero resta un passo statico. La *pipeline* riformula la domanda, interroga l'indice, ri-ordina i risultati e genera, sempre nello stesso ordine, con un solo passaggio. Il **RAG agentico** rovescia questa impostazione: il recupero non è più un passo a monte della generazione, ma un'azione che l'agente decide, parametrizza e itera autonomamente. In termini CoALA (Sezione 2.3), il *retrieval* entra nell'*action space* dell'agente. I moduli *pre-* e *post-retrieval* di Gao et al. [13] cessano di essere stadi fissi e diventano capacità che l'agente invoca solo quando servono.

Tabella 2.3: Metriche di valutazione per le componenti di una *pipeline* RAG, organizzate per stadio: recupero, qualità del contesto e generazione.

Stadio	**Metrica**	**Definizione operativa**
Recupero	Recall@k	Frazione di documenti rilevanti presenti nei primi k risultati
Recupero	Precision@k	Frazione di documenti restituiti effettivamente rilevanti
Recupero	MRR	$\mathbb{E}[1/\text{rank}]$ del primo documento rilevante
Recupero	nDCG@k	Guadagno cumulato scontato per posizione, normalizzato
Contesto	Context Precision	Frazione di *chunk* recuperati che risultano rilevanti rispetto alle evidenze di riferimento
Contesto	Context Recall	Frazione delle evidenze di riferimento effettivamente coperte dai *chunk* recuperati
Generazione	Faithfulness	Frazione di affermazioni in output supportate dal contesto recuperato
Generazione	Answer Relevance	Aderenza semantica della risposta alla *query*
Generazione	Answer Correctness	Corrispondenza fra la risposta prodotta e la risposta di riferimento, sul piano fattuale e semantico

Il cambiamento riguarda prima di tutto il controllo. In un RAG statico la sufficienza dell'evidenza è decisa *a priori* dal progettista, che fissa quanti frammenti recuperare prima di generare. In un RAG agentico è invece l'agente a stabilire, durante l'esecuzione, se ciò che ha raccolto basta o se serve un altro giro.

Il meccanismo eredita il paradigma **ReAct** [64], che intreccia passi di ragionamento e passi di azione. L'agente formula una *query*, osserva i risultati, ne valuta la pertinenza e, se non bastano, riformula e recupera di nuovo. Si apre così un ciclo di recupero iterativo, in cui ogni passo dipende dall'esito del precedente, fino a una condizione di arresto decisa dall'agente stesso. Approcci di auto-correzione come **Reflexion** [49] estendono l'idea: l'agente critica l'evidenza raccolta e usa quella critica per indirizzare il recupero successivo.

Il guadagno principale è l'adattività. Una domanda multi-*hop*, che impone di comporre informazioni sparse su più documenti, mal si presta al recupero a passaggio unico. L'agente può invece scomporla, recuperare per sottoproblemi e ricombinare le evidenze. Il prezzo è speculare: ogni iterazione aggiunge una chiamata al modello e una al sistema di recupero, e il costo in *token* e in latenza cresce con il numero di giri. Il controllo del flusso, inoltre, torna a dipendere da un

processo probabilistico. È la stessa tensione tra parte probabilistica e controllo deterministico che attraversa tutto il capitolo. Il RAG agentico scambia la prevedibilità del costo fisso con la flessibilità, e risulta preferibile dove la natura delle domande lo giustifica.

2.5.2 Direct Corpus Interaction

Su questa linea si colloca un paradigma più radicale. Nelle sue forme note il RAG agentico mantiene comunque un *vector store*, e l'agentività riguarda soltanto il momento e i parametri dell'interrogazione: l'indice resta. La **Direct Corpus Interaction** (DCI), formalizzata da Li et al. [28], rimuove anche quello.

L'indice vettoriale sparisce del tutto, e l'accesso al *corpus* passa a un agente dedicato, l'*Agentic File Search* (AFS), con i soli permessi nativi del *file system*: `glob` per individuare i file, `grep` per cercare nel testo, `read` per leggere i passi rilevanti. Vengono meno il *chunking*, l'*embedding* e le chiamate di rete. Resta l'esplorazione testuale diretta, con gli stessi strumenti che userebbe una persona davanti a una cartella di documenti.

La differenza con il RAG agentico classico riguarda il meccanismo stesso del recupero. Là opera una ricerca per similarità semantica su vettori. Qui opera un recupero lessicale e deterministico, dove ogni `grep` e ogni `read` restano tracciati nella sessione come qualunque altra operazione dell'agente. Il recupero diventa così un'azione agentica pienamente osservabile, al posto di una chiamata opaca a un servizio esterno. L'intero *retrieval* rientra nell'*action space* dell'agente, governato dalla stessa matrice di permessi che disciplina le altre sue capacità.

Il *trade-off* si manifesta su due fronti. La rinuncia all'*embedding* riduce la copertura semantica: la DCI trova ciò che corrisponde ai termini cercati e lascia fuori le formulazioni equivalenti espresse con altre parole. Su *corpus* molto estesi, inoltre, la ricerca testuale grezza allunga la latenza, e il Capitolo 7 ne misura l'entità sul percentile 95. In cambio otteniamo trasparenza, nessuna infrastruttura di indicizzazione da mantenere e un perimetro di sola lettura verificabile.

Il nostro apporto si articola in due punti. Il primo è l'adozione della DCI entro un'architettura agentica *enterprise*, dove il recupero ricade sotto la stessa matrice di permessi che già governa le altre capacità dell'agente. Il secondo è una validazione empirica indipendente che ne delimita il perimetro di validità, con tre sistemi di recupero a confronto su tre dataset e un giudice separato dal generatore. Gli autori documentano dove il paradigma funziona, noi misuriamo anche i casi in cui non risulta efficace (Capitolo 7).

2.6 Ottimizzazione del recupero: *chunking*, ricerca ibrida e *reranking*

La DCI rappresenta un polo, dove il *vector store* sparisce. Al polo opposto troviamo il RAG avanzato, che lo conserva e lo ottimizza in ogni modulo. Fra i due si collocano le tecniche che rendono più efficace un sistema denso: come segmentare i documenti, come combinare ricerca lessicale e semantica, come riscrivere la domanda, come riordinare i candidati. Nel Modular RAG queste scelte restano implicite in un singolo passo di similarità. Qui le rendiamo esplicite: nella parte finale del lavoro le mettiamo a confronto (Capitolo 7, Sezione 7.2) contro la DCI e contro un RAG classico di riferimento.

2.6.1 Strategie di segmentazione

Il taglio di un documento, prima di trasformarlo in vettori, condiziona ogni stadio successivo: un frammento troppo corto perde il contesto attorno alla risposta, uno troppo lungo diluisce il segnale in rumore.

Tre strategie rispondono in modi diversi. La segmentazione a *token* taglia a lunghezza fissa: è semplice, ma ignora la struttura del testo. La segmentazione semantica taglia dove la similarità fra frasi consecutive cala, seguendo lo stacco di argomento. La segmentazione ricorsiva rispetta invece la gerarchia sintattica: prima il paragrafo, poi la frase, poi la parola.

Librerie come Chonkie [4] implementano le tre varianti dietro un'unica interfaccia. La dimensione del blocco governa un compromesso fra contesto perso e rumore introdotto, e non ha un valore ottimale universale. Nella campagna finale confrontiamo le tre strategie sulle dimensioni 256, 512 e 1024 *token* (Figura 7.4).

2.6.2 Ricerca ibrida: denso più lessicale

Il recupero denso trova ciò che è semanticamente affine, ma risulta impreciso sui termini rari, sui nomi propri e sulle sigle, dove serve la corrispondenza esatta. Il recupero lessicale fa il contrario. BM25 [47] deriva dal quadro della rilevanza probabilistica, alternativo al modello a spazio vettoriale: pesa le occorrenze dei termini della *query* nel documento, satura il contributo delle ripetizioni e normalizza sulla lunghezza del documento. Resta la *baseline* lessicale di riferimento.

La ricerca ibrida unisce le due liste di risultati. La fusione ingenua, alternare i due ranghi senza pesi, fallisce perché i punteggi densi e lessicali sono definiti su scale incomparabili. La *Reciprocal Rank Fusion* (RRF) [6] risolve il problema ignorando i punteggi e sommando i reciproci dei ranghi: un documento in alto in entrambe le liste sale, uno presente in una sola scende. La fusione conviene sul lato server, dove il *vector store* indicizza vettori densi e sparsi e ne combina i ranghi in un'unica interrogazione, senza un secondo passaggio applicativo che riscarichi entrambe le liste.

2.6.3 Espansione e scomposizione della *query*

La domanda dell'utente è spesso una formulazione fra le tante possibili. Se il *corpus* usa altri termini per lo stesso concetto, una sola interrogazione recupera poco. A monte della ricerca si colloca perciò un modulo che riscrive la domanda prima di eseguirla, e le strategie disponibili sono due.

La prima riformula in parallelo. L'espansione della *query* genera con l'LLM più varianti della stessa domanda, recupera per ciascuna e fonde i risultati con RRF: è la *RAG-Fusion* [45]. Amplia la copertura quando la domanda è ambigua, ma moltiplica il costo: con k riformulazioni la *pipeline* esegue k generazioni LLM e k ricerche prima della fusione.

La seconda procede per passi. La scomposizione della *query* spezza una domanda complessa in sotto-domande e le esegue in sequenza, formulando ciascuna sulla base del contesto già recuperato. Il bersaglio sono le domande multi-*hop*, dove la risposta si compone attraversando un'entità intermedia che la formulazione iniziale non nomina. Recuperare quell'entità al primo passo permette di scrivere il secondo con il termine esatto.

Le due strategie differiscono per la dipendenza fra le interrogazioni. Nella *RAG-Fusion* le riformulazioni sono indipendenti e si eseguono in parallelo, quindi la latenza resta quella di un solo giro di recupero. Nella scomposizione ogni passo attende l'esito del precedente: la latenza cresce con il numero di passi e un errore iniziale indirizza male tutti i passi successivi. La Sezione 7.2.3 confronta le due strategie sui dataset multi-*hop* della campagna finale.

2.6.4 *Reranking* dei candidati

In fondo alla *pipeline* agisce il reranking, che riordina i candidati già recuperati con un *cross-encoder*. Il modello legge *query* e documento insieme e ne stima la pertinenza in un solo passaggio. È più preciso del *bi-encoder* usato per l'*embedding*, che codifica i due testi separatamente e non

può cogliere le loro interazioni fini [34]. Il prezzo è il costo per coppia, motivo per cui il reranker opera solo sui pochi candidati già selezionati.

Il recupero si articola quindi in due tempi. Un primo passo, denso o ibrido, seleziona un *pool* ampio di candidati e privilegia la copertura. Il *cross-encoder* riordina poi quel *pool* e ne trattiene i primi k. La dimensione del *pool* governa il compromesso: allargarlo aumenta la probabilità di includere il documento rilevante, ma allunga il tempo di riordino.

Espansione e reranking interagiscono in modo poco intuitivo quando compaiono nella stessa *pipeline*. Applicare il reranking prima o dopo la fusione (*early* contro *late*) produce risultati diversi a seconda del *corpus*, come verifichiamo nella Sezione 7.2.

2.6.5 Valutazione del recupero e verifica dei fatti

Le metriche Ragas della Sezione 2.7 misurano la generazione e si appoggiano a un giudice LLM. Il recupero, però, si valuta anche a monte, prima che il generatore intervenga. Recall@k, MRR e nDCG@k (Tabella 2.3) confrontano i documenti recuperati con le relazioni di rilevanza note, senza bisogno di alcun giudice: sono deterministiche e riproducibili. Le trattiamo come complementari alle metriche di generazione, perché isolano la qualità del recupero dal rumore introdotto a valle.

Un secondo tipo di compito allarga il quadro. Nella verifica dei fatti il sistema classifica un'affermazione come sostenuta, smentita o priva di evidenza sufficiente, senza produrre una risposta testuale libera. Cambia di conseguenza la misura: al posto della similarità con una risposta di riferimento subentra l'accuratezza di classificazione rispetto a un'etichetta nota. SciFact [54] fornisce il caso di studio: affermazioni scientifiche annotate con il verdetto e con gli *abstract* che lo sostengono. Da questa risorsa estraiamo il campione di cento affermazioni impiegato nella campagna finale. È un regime in cui un buon recupero non basta, perché resta da decidere il verdetto. Il Capitolo 7 misura l'ampiezza di questo scarto.

2.7 Valutazione dei sistemi agentici: LLM-as-Judge e metriche complementari

Valutare un sistema agentico è difficile per una ragione specifica. Rispetto alla generazione breve o al *question answering* chiuso, un agente può sbagliare anche nel percorso, oltre che nel contenuto della risposta: sceglie lo strumento sbagliato, pianifica male, usa il contesto in modo

improprio, è troppo lento. Le metriche solo testuali, come BLEU o ROUGE, di norma non bastano.

2.7.1 Il paradigma LLM-as-Judge

Una risposta a questa difficoltà è il paradigma **LLM-as-Judge** [65]: un modello linguistico valuta la qualità degli output, di solito su criteri espliciti definiti da *rubric* strutturate. Sul *benchmark* MT-Bench, Zheng et al. [65] riportano un accordo del ~80% tra GPT-4 giudice e annotatori umani esperti, vicino all'accordo tra umani sugli stessi *item*. La valutazione copre così anche la qualità semantica e la coerenza, oltre alla correttezza superficiale. Zheng et al. [65] segnalano però limiti importanti:

- ***Verbosity bias***: tendenza a favorire risposte più lunghe;
- ***Position bias***: tendenza a preferire la prima risposta nelle comparazioni A/B;
- ***Self-enhancement bias***: preferenza per stili generativi simili a quelli del giudice.

Un caso insidioso è il ***self-evaluation bias***, in cui il modello giudice coincide con il generatore. Le metriche risultano allora sovrastimate in modo sistematico, perché premiano l'auto-coerenza del modello prima ancora della qualità della risposta. Il limite riguarda anche il nostro lavoro: nella valutazione preliminare della Fase 2 il giudice e il generatore sono lo stesso modello, servito sull'infrastruttura locale. Interpretiamo quei valori con la cautela che ne consegue. Nella campagna finale separiamo invece i due ruoli e affidiamo il giudizio Ragas a un modello di famiglia diversa dal generatore (Capitolo 7).

Per rendere più affidabile il giudizio si rinuncia al giudice unico. Verga et al. [52] propongono di sostituirlo con una giuria di modelli diversi (*Panel of LLM evaluators*): si raccolgono i voti di più valutatori eterogenei e se ne aggrega il giudizio. La media attenua i *bias* del singolo modello, in particolare il *self-enhancement*, e si avvicina di più all'accordo umano, a fronte di un costo computazionale maggiore.

Nei sistemi RAG si è affermato **Ragas** [9], che scompone la qualità in metriche calcolate tramite LLM: *faithfulness*, *answer relevancy*, *context precision/recall* e *answer correctness*. Il suo contributo metodologico è disaccoppiare gli errori di *retrieval* da quelli di *generation*: un *context recall* basso con *faithfulness* alta indica un problema di recupero, non di generazione. Ragas eredita comunque le stesse fragilità di *LLM-as-Judge*, incluso il *self-evaluation bias*.

Per attenuare le distorsioni residue, la letteratura suggerisce rubriche esplicite, *position swapping* nelle comparazioni A/B (si scambia l'ordine delle risposte per neutralizzare il *position bias*) e, soprattutto, il confronto con giudici umani [65, 30].

2.7.2 Metriche complementari

LLM-as-Judge stima bene la qualità semantica dell'output, ma da solo non basta. Un agente può fallire lungo il percorso anche quando la risposta finale sembra buona. Servono perciò metriche rivolte al processo, accanto a quelle sul prodotto. Il ***task success rate*** misura la frazione di compiti portati a termine. La ***tool call accuracy*** valuta se l'agente ha scelto lo strumento giusto e con i parametri corretti. La ***latency*** e il ***cost***, in tempo e in *token*, ne misurano la sostenibilità in produzione. La **robustezza** verifica che a input equivalenti o leggermente perturbati corrispondano comportamenti coerenti.

La qualità dell'output finale compete a LLM-as-Judge, mentre la comprensione del sistema come processo richiede queste metriche operative e l'osservabilità del flusso. Per questo affianchiamo al giudice Ragas due misure indipendenti da esso: l'accuratezza di instradamento, calcolata in modo programmatico contro un *gold standard* annotato, e l'ispezione delle traiettorie di errore (Capitolo 7).

2.8 Orchestrazione a grafi e piattaforme *low-code*

Per implementare un sistema agentico serve un'infrastruttura di orchestrazione che coordini stato, transizioni, chiamate a strumenti e cicli di revisione. Oggi tre categorie di strumenti sono rilevanti: le piattaforme *low-code* (prototipazione visuale), i *framework* a grafi di stato (controllo programmatico, anche con cicli) e i *framework* agentici **dichiarativi** o *configuration-driven* (comportamento definito da artefatti di configurazione).

2.8.1 Piattaforme *low-code*

Piattaforme come **n8n** [33], **Dify** [8] e **Langflow** [25] permettono di costruire *workflow* LLM con interfacce visuali a nodi. I blocchi si dispongono su un *canvas* e si collegano con frecce, e ogni blocco incapsula una chiamata al modello, a un'API o a un database. Chi costruisce il flusso non scrive codice, e un prototipo funzionante richiede la sola composizione dei nodi sul *canvas*. La Sezione 4.3 riporta i criteri con cui abbiamo valutato queste piattaforme prima di adottarne una.

I limiti emergono quando il problema richiede controllo fine dello stato, cicli di revisione e ispezionabilità programmatica del *routing*. Lo stato transita come *payload* da nodo a nodo, privo di uno schema tipizzato, perciò un errore di forma si scopre tardi, a *runtime*. Le logiche cicliche complesse sono difficili da rappresentare su un *canvas* pensato per flussi lineari.

Il *versioning* opera su un *workflow* serializzato in JSON, il cui *diff* non è ispezionabile in revisione, e i test automatizzati sulla logica dei singoli nodi non hanno un supporto nativo. Il *routing* resta a sua volta implicito nella configurazione visuale. La Tabella 4.2 riporta la valutazione puntuale di questi criteri. Queste piattaforme restano efficaci per la prototipazione rapida, ma risultano meno adeguate quando i requisiti di controllo e auditabilità diventano stringenti.

2.8.2 Grafi di stato

I *framework* basati su **grafi di stato** adottano un'idea collaudata dell'ingegneria del software, la macchina a stati, e la applicano agli agenti. Il *workflow* è descritto come un insieme di nodi e di transizioni che operano su uno stato esplicito e condiviso, e i cicli di re-pianificazione, correzione e supervisione sono supportati in modo nativo. Un esempio rilevante è **LangGraph** [24]. Ogni nodo è una funzione che legge e aggiorna una porzione dello stato tipizzato, e le transizioni condizionali selezionano il percorso successivo, anche all'indietro, per ritentare un passo fallito. Il flusso è definito in codice, lontano tanto dalla conversazione emergente quanto dal *canvas* visuale.

Da qui derivano validazione a monte, *logging* strutturato, persistenza dello stato e punti di supervisione umana. I grafi di stato sono quindi più adatti ai sistemi agentici destinati all'esercizio, quando il flusso richiede cicli di re-pianificazione, stato tipizzato e interruzione durevole con ripresa esatta dopo un guasto.

2.8.3 *Framework* agentici dichiarativi e *configuration-driven*

All'estremo opposto, lontano sia dal *low-code* visuale sia dall'orchestrazione imperativa in codice, si colloca una terza categoria emergente: i *framework* agentici **dichiarativi** e *configuration-driven*. Qui la struttura del sistema si dichiara, e il *framework* ne garantisce il rispetto: la programmazione esplicita del flusso viene meno. Strumenti come **OpenCode** [36] definiscono il comportamento con tre artefatti dichiarativi: un file di configurazione centrale (*provider* del modello, agenti e permessi), la logica cognitiva di ogni agente scritta come *prompt* Markdown con *frontmatter*,

e gli strumenti operativi come moduli di codice isolati (*tool* TypeScript o Python). Il sistema diventa così un insieme di file di testo, versionabili come qualunque altro sorgente.

Il tratto distintivo del paradigma è il meccanismo con cui controlla il *routing*. Nei grafi di stato il flusso è vincolato dalla topologia programmata. Nei *framework* dichiarativi il vincolo deriva invece da una **matrice di permessi** (*permission model*) a livello di configurazione. A ogni agente concediamo o neghiamo l'accesso a ogni *tool*, così un *orchestrator* privo di permessi operativi è strutturalmente obbligato a delegare ai subagenti specializzati. Il *routing* emerge quindi dalla composizione tra capacità dichiarate e divieti imposti, e sposta il controllo dal *flow control* esplicito alla qualità dei *prompt* e alla granularità dei permessi.

Il paradigma offre due vantaggi: la portabilità del *deployment*, perché gli artefatti sono testuali e versionabili, senza immagini applicative *custom*, e la velocità di estensione, perché una nuova *skill* è un *prompt* più un modulo *tool*, senza toccare il grafo. I costi sono simmetrici: si perde controllo fine sul flusso e *type safety*, dato che lo stato condiviso tipizzato dei grafi di stato non ha un equivalente diretto.

La letteratura empirica su questo paradigma è ancora scarsa, e mancano confronti diretti tra orchestrazione imperativa e dichiarativa sullo stesso dominio. È qui che contribuiamo, con la migrazione documentata da LangGraph a OpenCode nel Capitolo 6.

2.9 Definizione del gap accademico e domanda di ricerca

La rassegna ha percorso il ciclo di vita di un agente, dai paradigmi di ragionamento ai sostrati che li ospitano. Una sola tensione l'attraversa per intero: da un lato la parte probabilistica del sistema, il modello che ragiona, dall'altro la necessità di un controllo deterministico e verificabile. I paradigmi di ragionamento la affrontano con la separazione tra piano ed esecuzione. CoALA la formula come problema di collocazione della *decision procedure*. Il *tool use* la traduce in schemi tipizzati. L'orchestrazione la distribuisce su sostrati diversi. Da questa lettura unitaria emergono tre *gap* convergenti, rispetto ai quali il presente lavoro si posiziona.

G1 Separazione formale fra *reasoning* ed *execution* come invariante architetturale. Plan-and-Execute [56, 57] e CoALA [50] convergono su un punto: la *decision procedure* non dovrebbe coincidere con il completamento testuale del modello. La letteratura documenta però poche implementazioni in cui quella separazione passi per un piano formale fra strato inferenziale e strato esecutivo, validato al confine con uno schema Pydantic o JSON

Schema. Nella maggior parte dei casi resta una convenzione di *prompting*. Il *gap* raccoglie la critica ai MAS della Sezione 2.2.1 e quella alla *decision procedure* implicita della Sezione 2.3.

G2 Validazione *enterprise* sul processo end-to-end. Gran parte degli studi su agenti LLM e MAS misura le prestazioni su *benchmark* sintetici o su compiti di generazione di codice [18, 27, 60]. Restano poco documentati i processi aziendali reali, dove auditabilità, *compliance*, latenza e integrazione con i sistemi esterni valgono come vincoli congiunti. Manca soprattutto il tracciamento dell'intero percorso ingegneristico, dalla prototipazione all'esercizio, con la composizione fra scelte infrastrutturali e scelte architetturali.

G3 Scelta del sostrato di orchestrazione: *low-code*, grafi di stato e *framework* dichiarativi. Piattaforme *low-code* (n8n, Dify, Langflow), *framework* a grafo di stato (LangGraph) e *framework* agentici dichiarativi (OpenCode) sono oggi alternative concorrenti sullo stesso problema. La letteratura fornisce però poche evidenze comparative dirette su ciò che si guadagna e su ciò che si perde nel passaggio dall'una all'altra, misurando l'affidabilità di instradamento e l'onere di manutenzione dei piani tipizzati. Manca soprattutto un confronto condotto sullo stesso dominio applicativo fra le due forme di orchestrazione: quella **imperativa**, dove il grafo di stato è programmato, e quella **dichiarativa**, fatta di configurazione, *prompt* e *permission model*. Il *gap* raccoglie la discussione delle Sezioni 2.8.1, 2.8.2 e 2.8.3.

Su questa base, la **domanda di ricerca principale** del presente lavoro è:

> **RQ**. *In che misura la separazione formale fra* reasoning *ed* execution*, adottata come invariante architetturale, riduce gli errori di instradamento verso gli agenti rispetto a un'orchestrazione che lascia decisione ed esecuzione nello stesso passo? Misuriamo la riduzione su un banco di 100* query *annotate, con accuratezza di* routing *e F1 per agente.*

La RQ si articola nelle tre sotto-domande già enunciate nel Capitolo 1 (Sezione 1.1), in corrispondenza biunivoca con i tre *gap*. RQ1 raccoglie G1 e riguarda il formalismo: quale formulazione operativa della separazione riduce gli errori di instradamento. RQ2 raccoglie G2 e riguarda il processo: a quali condizioni un percorso incrementale porta un prototipo fino all'esercizio *enterprise*. RQ3 raccoglie G3 e riguarda il sostrato: cosa si guadagna e cosa si perde attraversando *low-code*, orchestrazione imperativa e *framework* dichiarativo.

Le RQ1–RQ3 vengono operazionalizzate in metriche specifiche nel Capitolo 4 (Sezione 4.2) e affrontate empiricamente nel Capitolo 7. Il capitolo successivo presenta le tecnologie selezionate per l'implementazione dell'architettura proposta.

3. Tecnologie e strumenti

Due vincoli aziendali guidano la scelta dello *stack*. I modelli sono eseguiti *on-premises*, e il sistema comunica con applicativi eterogenei già in esercizio. L'architettura aggiunge vincoli propri: separare *reasoning* ed *execution*, conservare lo stato e una memoria semantica, reggere il carico concorrente su una sola GPU H100.

Valutiamo ogni strumento in base a quattro proprietà del sistema: la tipizzazione dello stato, l'auditabilità dell'esecuzione, la modularità e la portabilità del *deployment*. Solo per la piattaforma *low-code* della Fase 1 il confronto è formalizzato con un punteggio (Tabella 4.2). Per gli altri componenti motiviamo la scelta nelle sezioni che seguono.

Lo *stack* rispecchia le quattro fasi del progetto. n8n (Sezione 3.6) è il prototipo *low-code* della Fase 1. LangGraph (Sezione 3.3) e pgvector (Sezione 3.5) sono i componenti centrali dell'orchestratore deterministico (v2, Fase 2). OpenCode (Sezione 3.7), il *backend* RAG (Sezione 3.8) e Ragas (Sezione 3.9) appartengono alle Fasi 3–4 (v3 e Agentic RAG).

Ordiniamo le sezioni per tema. Trattiamo n8n dopo i componenti dell'orchestratore deterministico, perché la sua analisi richiama i limiti che ne hanno motivato il superamento.

3.1 Motore inferenziale: i modelli Qwen e il *serving* vLLM

Il modello al centro del sistema produce piani strutturati, sceglie gli strumenti e genera *output* conformi a uno schema definito a monte. Lo valutiamo sulla conformità allo schema e sul comportamento nel *tool use*, più che sulle capacità conversazionali.

Nelle quattro fasi abbiamo usato più configurazioni di *serving*, con un vincolo comune: l'inferenza avviene *on-premises*, via vLLM su GPU NVIDIA H100 80 GB, senza API *cloud* di terze parti. La Tabella 3.1 riporta il modello di ciascuna fase e distingue i ruoli operativi (*Planner*, *Synthesizer*, agenti) dal ruolo di valutazione (*judge*).

L'inferenza interna risponde a due vincoli aziendali. Il primo è la sovranità del dato: i *payload* aziendali non devono uscire dal perimetro verso un fornitore esterno. Il secondo riguarda la spesa. Con una GPU dedicata il costo è fissato dall'ammortamento dell'acceleratore. Con le API *cloud* cresce invece con il numero di *token* elaborati.

Tabella 3.1: Configurazione dei modelli LLM per fase del progetto. Tutte le configurazioni operano *on-premises* su GPU H100, servite localmente via vLLM.

Fase	Ruolo	Modello
Fase 1 (n8n)	*Planner* prototipo	GPT-Oss-120b
Fase 2 (LangGraph)	*Planning*/*Synthesis*/agenti + LLM-*as*-*Judge* preliminare	Qwen3-Coder-30B-A3B -Instruct nello sviluppo; Qwen3.6-27B-AWQ-INT4 nella campagna comparativa
Fasi 3–4 (OpenCode/Agentic RAG)	*Orchestrator*/subagenti/generatore dei tre sistemi RAG, agente su *file system* compreso	Qwen3.6-27B-AWQ-INT4
Fase 4 (Agentic RAG)	*Judge* Ragas	MiniMax-M3

Nella Fase 2 un solo modello copre i ruoli operativi e la campagna preliminare LLM-*as*-*Judge*. Nella Fase 4 tutti e tre i sistemi RAG separano i due ruoli: `qwen36-27b-awq-int4` genera, `minimax-m3` giudica (Sezione 3.10). Il confronto definitivo tra v2 e v3 si basa sul *benchmark* di *routing* programmatico contro *gold standard*, senza giudice LLM (Capitolo 7).

Il modello cambia da una fase all'altra, seguendo gli aggiornamenti dello *stack* di *serving* aziendale riportati nella Tabella 3.1. Questo cambio rende meno affidabili i confronti fra fasi diverse, e ne teniamo conto fra le minacce alla validità (Capitoli 7 e 9). Un secondo rischio compare quando il modello giudice coincide con il generatore: il *self-evaluation bias*. Nella campagna RAG lo attenuiamo con un giudice di famiglia diversa dal generatore, per tutti e tre i sistemi (Sezione 3.10).

Per la Fase 2 selezioniamo Qwen 3 Coder [43], nella variante `Qwen3-Coder-30B-A3B-Instruct`, come modello di riferimento per il *Planner* e il *Synthesizer*. È un'evoluzione di Qwen 2.5 Coder [44]. Lo serviamo localmente via vLLM [23, 53] su una GPU H100 80 GB. vLLM divide la *cache* KV in blocchi non contigui, con la tecnica *PagedAttention*, e li assegna alle richieste concorrenti man mano che servono. Una sola H100 regge così le misure prestazionali della Fase 2 senza replicare il modello.

Qwen3-Coder vale per lo sviluppo della v2 e per le misure raccolte in quella fase. Quando rimisuriamo la v2 per il confronto con la v3, lo *stack* aziendale è già passato a Qwen3.6-27B-AWQ-INT4. La campagna comparativa usa quindi questo modello per entrambe le architetture (Capitolo 7, Sezione 7.1.1).

Perché proprio questo modello? Lo scegliamo su quattro requisiti. Il *Planner* deve produrre JSON conforme a `WorkflowPlan`, e il modello è addestrato alla generazione strutturata. I pesi sono pubblici, quindi possiamo eseguirlo *on-premises*. Il modello supporta nativamente il *tool calling*. L'ultimo requisito è il nodo singolo: l'architettura a esperti attiva 3,3 miliardi di parametri su 30,5 per *token*, e anche senza quantizzazione i pesi occupano circa 61 GB. Su una

H100 da 80 GB resta così margine per la *cache* KV. Nel nostro caso d'uso questi requisiti pesano più della qualità sui *benchmark* generalisti.

Le alternative non rispettano il vincolo del nodo singolo alla stessa precisione. Un modello denso da 70B in BF16 richiede circa 140 GB e non entra in una H100. Mixtral 8×22B ha 141 miliardi di parametri totali: in BF16 occupa circa 280 GB e richiede più acceleratori. La quantizzazione a 4 bit riduce queste occupazioni, con una possibile perdita di qualità, ed è la soluzione adottata nelle Fasi 3–4 con Qwen3.6-27B-AWQ-INT4. La configurazione di *serving* delle campagne è nella Tabella 7.1 del Capitolo 7.

Impieghiamo un solo modello per tutti i ruoli operativi. Così il confronto tra i ruoli non risente di differenze fra modelli, e sul nodo GPU gestiamo un solo *endpoint* di inferenza.

3.2 Pydantic come strato di validazione del piano

L'architettura affida a Python l'orchestrazione e a Pydantic v2 [5] la validazione e la serializzazione dei dati. Nel sistema circolano dati di natura diversa: l'*output* dell'LLM, i *payload* delle API, i risultati del recupero, lo stato del grafo. Pydantic v2, con il *core* scritto in Rust, li converte in modelli validati. Non abbiamo misurato il costo della validazione. Lo consideriamo trascurabile rispetto alla latenza della chiamata al modello, dell'ordine del secondo (Capitolo 5).

Lo strato rende esplicita l'interfaccia tra la componente probabilistica e quella deterministica. Un *output* malformato del *Planner* produce un errore di validazione prima che il *Route Executor* esegua uno *step*. Tre modelli dati definiscono il piano e il confine *reasoning*/*execution*: `WorkflowStep`, `WorkflowPlan` ed `ErrorState`. Il dettaglio dei campi è nella Tabella 5.3, presentata con la formalizzazione del piano nel Capitolo 5 (Sezione 5.4).

Il vincolo principale di questo strato riguarda i nomi degli strumenti. Il campo `tool` di `WorkflowStep` è un `Literal` che ammette i soli agenti registrati. Se il *Planner* nomina uno strumento non enumerato, Pydantic solleva un `ValidationError` prima dell'invocazione. Un'intera classe di errori passa così dal momento dell'esecuzione a quello della validazione del piano. La coerenza del grafo delle dipendenze non rientra nello schema. Il *Route Executor* la verifica durante l'esecuzione e avvia uno *step* solo quando i suoi `depends_on` hanno già prodotto un risultato.

Un secondo vincolo agisce a monte. Il *Planner* riceve lo schema `WorkflowPlan` come firma di funzione, tramite il *function calling* di vLLM [53], e ne produce gli argomenti. I due vincoli

insieme sono la spiegazione più probabile del calo delle allucinazioni di strumento: da 3 casi su 27 nel *Planner* n8n allo 0,7% dell'orchestratore v2 (Sezione 7.1.3). Il dato però è descrittivo. I due campioni sono distinti, non appaiati, e usano modelli diversi, quindi non isolano l'effetto dei vincoli.

Ogni confine tra un nodo inferenziale e un nodo deterministico passa da un modello Pydantic tipizzato. Nel progetto trattiamo l'assenza di uno schema come un difetto di *design*.

3.3 Orchestrazione deterministica con LangGraph

Per separare *reasoning* ed *execution* serve un'infrastruttura che rappresenti lo stato in modo esplicito e permetta cicli di revisione. Una catena lineare di chiamate non prevede cicli. Per questo selezioniamo LangGraph [24].

Perché non uno *scheduler* a DAG statico, più semplice da realizzare? Un sistema agentico si comporta come una macchina a stati la cui forma dipende dai dati in ingresso. La topologia del flusso è generata a *runtime* dal *Planner*, e non è nota quando il grafo viene definito. Quando un passo fallisce, l'esecuzione deve poter ripartire da un passo precedente. Una pausa per l'intervento umano, infine, deve sopravvivere a un riavvio del sistema. Uno *scheduler* statico non copre nessuno dei tre casi.

LangGraph modella il flusso come un insieme di nodi che leggono e scrivono uno stato condiviso. Ogni nodo è una funzione Python, e il codice determina la transizione al nodo successivo in base allo stato corrente. Possiamo così tenere distinti i nodi che invocano l'LLM dai nodi puramente deterministici.

LangGraph permette anche l'esecuzione parallela. Il *Planner* può produrre un piano con più passi indipendenti. In quel caso il *Route Executor*, una funzione Python pura, li instrada insieme, e LangGraph esegue un'istanza di ciascun agente con il proprio compito (*fan-out*). Se invece un passo fallisce, un *Feedback Loop* riporta il controllo al *Planner*, per un massimo di tre tentativi (Capitolo 5).

Un *checkpointer* su PostgreSQL rende lo stato persistente. Dopo un guasto il sistema riprende dall'ultimo *checkpoint*, e può sospendersi in attesa di un'approvazione umana (*Human-in-the-Loop*) senza perdere il contesto. La topologia completa e i dettagli implementativi sono nel Capitolo 5.

3.4 Model Context Protocol

Un agente diventa operativo quando invoca strumenti esterni, come una ricerca sul *web* o una *query* al *database*. Se ogni strumento espone un'interfaccia diversa, l'orchestratore accumula un adattatore dedicato per ciascuno. Il Model Context Protocol (MCP), proposto da Anthropic nel 2024 [1], definisce un'interfaccia unica per questi strumenti.

In MCP le parti sono due: il *client*, cioè il nostro agente, e il *server*, che espone lo strumento. La struttura interna dello strumento resta nascosta al *client*. Con `tools/list` chiede al *server* l'elenco degli strumenti disponibili, con `tools/call` ne invoca uno e riceve il risultato. I messaggi seguono il formato JSON-RPC 2.0, un dettaglio che la logica dell'agente non tratta direttamente.

Il vantaggio principale di MCP è il disaccoppiamento. Il *Route Executor* invoca gli strumenti solo tramite le due primitive, senza dipendere dalla loro implementazione. Possiamo aggiungere o sostituire uno strumento senza modificare l'orchestratore.

I due strumenti esterni della v2 hanno però profili di esecuzione diversi, e usano due trasporti distinti (Tabella 5.5). Il *Web Search Agent* segue uno schema domanda-risposta: invia la *query*, attende il risultato dell'API esterna e lo restituisce. La latenza dipende dal servizio remoto, e basta il trasporto *Streamable* HTTP, con richiesta e risposta in un unico scambio.

Il *Deep Research Agent* lavora invece per passi. Durante l'esecuzione del *tool* il *server* chiede al *client* nuove generazioni del modello, tramite la primitiva *Sampling* di MCP. In questo modo il *server* non conserva credenziali e usa l'infrastruttura vLLM già autenticata. Nella v2 lo scambio avviene su un canale SSE (*Server-Sent Events*), che riporta l'avanzamento della ricerca in tempo reale. Le fermate per l'approvazione umana usano invece la primitiva `interrupt()` di LangGraph (Sezione 5.6.1).

3.5 Persistenza ibrida ed *embedding*: PostgreSQL, pgvector e bge-m3

Adottiamo PostgreSQL 16.3 [41] con l'estensione pgvector 0.7.0 [40], e usiamo un solo motore per la persistenza relazionale e la ricerca vettoriale. Il *database* svolge quattro funzioni: persiste lo stato del grafo tramite *checkpointer*, registra i *log* strutturati, conserva il catalogo dei prodotti e i risultati degli agenti, contiene la *Semantic Cache*.

Con pgvector la v2 evita un *database* vettoriale dedicato come Milvus, Weaviate o Qdrant. Gestiamo un solo motore e un solo *backup*, e le transazioni che toccano insieme tabelle relazionali e vettori sono atomiche. Il compromesso è adatto alla scala del progetto, fra 10^3 e 10^4 *embedding*. Oltre il milione di vettori i motori dedicati offrono indici distribuiti su più nodi, che pgvector non prevede.

Anche la v3 usa pgvector come *vector store*. La *knowledge base* del `rag-agent` e il catalogo prodotti risiedono sullo stesso motore (Sezione 3.8).

Nella v2 i vettori provengono da BAAI/bge-m3 [3], un modello di *embedding* multilingue che supporta recupero denso, sparso e multi-vettore. Usiamo il solo recupero denso, con vettori da 1.024 dimensioni che corrispondono alla colonna `vector(1024)`. Il modello è eseguito in locale, come quello generativo, e sul recupero multilingue riporta risultati allo stato dell'arte su MIRACL e MKQA [3].

3.5.1 La *Semantic Cache* come tabella vettoriale

La *Semantic Cache* è una tabella pgvector. Ogni riga conserva l'*embedding* di una *query* già elaborata e la risposta memorizzata. Il confronto usa la distanza coseno di pgvector, sugli stessi vettori `vector(1024)` prodotti da BAAI/bge-m3 per il *RAG Agent* della v2. Una ricerca esatta su questa colonna scandirebbe l'intera tabella: per evitarlo la indicizziamo con HNSW [31], l'indice approssimato introdotto nella Sezione 2.5.

Per l'indice usiamo i parametri predefiniti di pgvector: `m` = 16 connessioni per nodo ed `ef_construction` = 64. Alla scala di 10^3–10^4 vettori non li abbiamo tarati, e il richiamo dell'indice rispetto alla ricerca esatta non compare fra le misure del progetto. Il funzionamento del nodo e la soglia di similarità sono descritti nel flusso dell'architettura LangGraph (Sezione 5.3).

3.6 Piattaforma di prototipazione *low-code*: n8n

Prima di sviluppare l'orchestratore deterministico ci serve un modo rapido per validare i requisiti funzionali multi-agente. Per la Fase 1 selezioniamo quindi una piattaforma *low-code*.

In una piattaforma di questo tipo un flusso di lavoro si costruisce in un editor visuale, collegando blocchi tra loro. Ogni blocco corrisponde a un'azione, per esempio una chiamata a un servizio, una richiesta a un modello o una scrittura su *database*. Le frecce tra i blocchi definiscono l'ordine delle operazioni. Il vantaggio è il tempo di iterazione: modificare un flusso

non richiede codice di infrastruttura né ricompilazione. Nell'orchestratore in codice della v2, per confronto, l'aggiunta di un nuovo agente ha richiesto fra le otto e le dodici ore (Capitolo 6).

Tra le piattaforme di questo genere selezioniamo n8n [33], dopo un confronto con Dify [8] e Langflow [25] su cinque criteri (Tabella 4.2, Capitolo 4). n8n ottiene 21 punti su 25, contro 17 di Dify e 13 di Langflow. Il distacco maggiore riguarda l'integrazione con API esterne e l'ecosistema di nodi già pronti. Il nodo `Code` accetta inoltre JavaScript scritto su misura: nel prototipo questi nodi contengono 501 righe di codice (Tabella 5.6). n8n perde un punto rispetto a Dify solo sull'orchestrazione agentica.

Su n8n implementiamo il prototipo *Planner-Executor* descritto nel Capitolo 1. Il *Planner*, basato su GPT-Oss-120b, produce un piano JSON, e i nodi della piattaforma lo elaborano in modo deterministico. Questo prototipo è la *baseline* funzionale del confronto con LangGraph.

Con questa fase verifichiamo quali requisiti la piattaforma soddisfa prima di investire nello sviluppo *custom*. Le criticità emerse, riassunte nella Tabella 1.1 del Capitolo 1, motivano la migrazione verso l'orchestratore deterministico (Capitolo 5).

3.7 *Framework* agentico dichiarativo: OpenCode

Nelle Fasi 3–4 riscriviamo l'architettura con OpenCode [36], una piattaforma dichiarativa (il paradigma è inquadrato nella Sezione 2.8.3). In LangGraph un grafo Python descrive come l'esecuzione passa da un nodo al successivo. In OpenCode dichiariamo il compito di ogni agente e le regole che deve rispettare, e il flusso deriva dalla configurazione.

Un sistema OpenCode si definisce su tre livelli. Il file di configurazione `opencode.json` dichiara i modelli e gli agenti. Ogni agente è un file Markdown in `.opencode/agents/`, con un *front-matter* YAML e un *prompt* che ne descrive il compito in linguaggio naturale. Gli strumenti operativi sono moduli TypeScript in `.opencode/tools/`, uno per ogni azione, come una ricerca o una *query* al *database*. Con questa organizzazione i 1.366 LOC di orchestrazione Python della v2 (Capitolo 5) diventano file di configurazione e *prompt* testuali.

Il coordinamento spetta all'*orchestrator*. Questo agente riceve la richiesta dell'utente, ne interpreta il tipo e la assegna a uno dei quattro subagenti: *web*, *database*, RAG o ricerca approfondita (Tabella 6.3). L'orchestratore pianifica e delega, mentre l'esecuzione tocca ai subagenti. È la stessa separazione tra ragionamento ed esecuzione delle fasi precedenti.

Cambia il meccanismo che garantisce la separazione. In LangGraph una funzione Python instrada le richieste. In OpenCode la garanzia viene dai permessi dichiarati nel *front-matter* di ogni agente: l'orchestratore ha accesso al solo *tool* `task`, con cui delega il compito a un subagente, e gli strumenti operativi gli sono negati.

Il *trade-off* è discusso nei Capitoli 6 e 7. Guadagniamo portabilità e riduciamo il codice di orchestrazione. In cambio, la separazione passa dal codice Python a una regola di configurazione, applicata dal *runtime* OpenCode (Sezione 6.3).

3.8 *Backend* RAG (pgvector) e ricerca *web*

Nella versione OpenCode il *vector store* del `rag-agent` è ancora pgvector [40], la stessa estensione PostgreSQL della v2, eseguita con Docker Compose. I *tool* `rag_query.ts` e `product_search_vector.ts` calcolano l'*embedding* della *query* e lo confrontano con l'indice pgvector per distanza coseno. `rag_query.ts` affianca a questo ramo denso un ramo lessicale, e fonde i due *ranking* per un recupero ibrido sulla *knowledge base* (Sezione 2.6.2). Il *tool* `db_query.ts` usa lo stesso motore per le *query* SQL esatte. Valgono qui le ragioni del motore unico esposte nella Sezione 3.5, con lo stesso limite di scala.

Rispetto alla prima fase della v2 cambia l'*embedder*: da BAAI/bge-m3 a `qwen3-embedding-0-6b`. La sostituzione allinea il *backend* RAG all'*endpoint* aziendale aggiornato e precede la campagna di valutazione. Al momento del confronto le due architetture recuperano quindi con lo stesso *embedder*, come discutiamo nel Capitolo 6.

Per la ricerca *web* il *tool* `web_search.ts` usa MCP (Sezione 3.4), lo stesso protocollo della v2. La richiesta arriva in JSON-RPC a un *server* MCP che espone la ricerca come strumento. Nella versione attuale il *server* si appoggia all'API di Exa AI [10]. Il motore di ricerca a valle può cambiare senza modifiche al *tool* dell'agente.

3.9 Ragas ed *embedding* di valutazione

Per misurare la qualità della *pipeline* RAG nella Fase 4 impieghiamo il *framework* Ragas [9]. Ragas scompone la qualità in cinque metriche, calcolate con l'aiuto di un LLM giudice.

Come localizziamo un errore della *pipeline*? Le metriche coprono due stadi. Per il contesto, *context precision* misura se i documenti pertinenti occupano le prime posizioni del *ranking*, e *context recall* quanta parte delle evidenze attese è stata recuperata. Per la generazione, *faithfulness*

verifica che la risposta non aggiunga contenuti assenti dal contesto, *answer relevancy* ne misura la pertinenza alla domanda e *answer correctness* l'accordo con la risposta di riferimento. Un *context recall* basso indica così un problema di recupero, una *faithfulness* bassa un problema di generazione.

Ragas eredita le fragilità del paradigma LLM-*as-Judge*. Il modello giudice applica i propri *bias*, e fra questi il *self-evaluation bias* compare quando valuta testi generati dallo stesso modello. Come contromisura separiamo generatore e giudice. La Sezione 3.10 ne descrive l'applicazione e i limiti. Per il confronto semantico tra risposta e riferimento Ragas usa anche un *embedder*, `qwen3-embedding-0-6b`. Leggiamo i valori assoluti con cautela, finché un *ensemble* di giudici eterogenei non li conferma (Capitolo 9).

3.10 *Stack* di *retrieval* sperimentale: Qdrant, Chonkie e giudice Ragas

Lo studio comparativo esteso della parte finale (Capitolo 7, Sezione 7.2) usa uno *stack* dedicato, distinto da quello *enterprise*. Il *backend* su pgvector serve l'*Agentic Platform* in esercizio (Sezione 3.8). Lo *stack* sperimentale serve un ambiente di prova *standalone*, in un *namespace* separato dello stesso *cluster*. Le due infrastrutture condividono la GPU e hanno basi di codice separate.

Il *vector store* dello *stack* è Qdrant [42]. Qdrant indicizza vettori densi e sparsi nella stessa collezione e li fonde lato *server* con la *Reciprocal Rank Fusion* (Sezione 2.6.2). La ricerca ibrida richiede così una sola chiamata al *database*. Con pgvector, che dalla versione 0.7.0 gestisce anche vettori sparsi con il tipo `sparsevec`, la fusione dei *ranking* è un passo applicativo, come in `rag_query.ts`.

Chonkie [4] segmenta i documenti. Espone dietro un'unica interfaccia le strategie di *chunking* a *token*, semantica e ricorsiva (Sezione 2.6.1), e ci permette di trattare la strategia come parametro dell'esperimento.

La coppia di modelli di recupero segue lo schema *retrieve-then-rerank* della Sezione 2.6.4. L'*embedding* è prodotto da `qwen3-embedding-0-6b`, un *bi-encoder* da 1.024 dimensioni che usiamo anche nel *backend* RAG della v3 e nel giudizio Ragas. Il *reranker* è `qwen3-reranker-0-6b`, un *cross-encoder* servito dallo stesso *endpoint*. Il primo modello seleziona i candidati per similarità, il secondo li riordina valutando *query* e documento insieme.

Lo *stack* separa anche il generatore dal giudice. Il *self-evaluation bias* (Sezione 3.9) compare quando il modello che valuta coincide con quello che genera. In tutti e tre i bracci, RAG classico, RAG avanzato e agente su *file system*, la generazione usa `qwen36-27b-awq-int4` (Qwen3.6-27B). Il giudizio Ragas usa `minimax-m3` (MiniMax-M3), un modello di famiglia diversa, impiegato solo come giudice. La separazione riduce la tendenza del giudice a premiare testi simili ai propri, e lascia invariati gli altri suoi *bias*.

4. Metodologia della ricerca

I due capitoli precedenti hanno fissato il contesto teorico e lo *stack* tecnologico. Resta da definire il metodo. Questo capitolo descrive il disegno della ricerca: la cornice epistemologica che lo governa, la traduzione delle domande di ricerca in grandezze misurabili e i protocolli delle due campagne sperimentali, con le condizioni che ne rendono ripetibile la misura.

L'ordine è il seguente. La cornice *Design Science Research* apre il capitolo con le sue implicazioni operative (Sezione 4.1). Seguono l'operazionalizzazione di RQ1–RQ3 in metriche (Sezione 4.2) e la valutazione comparativa che porta alla scelta della piattaforma *low-code* di Fase 1 (Sezione 4.3). Le due sezioni centrali descrivono il protocollo del *benchmark* di *routing* (Sezione 4.4) e quello dello studio comparativo sul recupero (Sezione 4.5). Il capitolo si chiude con le condizioni di riproducibilità (Sezione 4.6).

4.1 La cornice: Design Science Research

Il lavoro produce un artefatto software e, insieme, conoscenza sul modo in cui quell'artefatto si comporta. Le due cose non sono separabili. L'intreccio esclude un disegno sperimentale classico, con l'ipotesi fissata prima di costruire il sistema. Adottiamo perciò il *Design Science Research* (DSR) nella formulazione di Hevner et al. [17] e nel processo di Peffers et al. [39]. In quella cornice l'artefatto tecnologico è al tempo stesso prodotto della ricerca e strumento per generarla.

La scelta ha tre conseguenze operative. L'unità di analisi diventa l'architettura, cioè il modo in cui i componenti probabilistici e quelli deterministici sono composti. Il lavoro procede per iterazioni *build-measure-learn*: le criticità rilevate alla fine di una fase diventano il requisito d'ingresso della fase successiva. L'evidenza si costruisce su confronti fra fasi successive, quindi longitudinali e non appaiati: il Capitolo 7 ne tiene conto nella lettura dei confronti (Sezione 7.1.1).

Le quattro fasi del progetto sono altrettanti cicli DSR. Nella Fase 1 costruiamo un prototipo *low-code* e ne rileviamo sei criticità (Tabella 1.1): una quantificata su 50 *query* di test, le altre cinque emerse dall'ispezione dei *workflow*. Nella Fase 2 l'architettura è riscritta su LangGraph per rispondere a quelle criticità. Nella Fase 3 passa a un *framework* dichiarativo, e ne misuriamo la delega sullo stesso *benchmark* della fase precedente. Con la Fase 4 la valutazione si estende al

recupero delle informazioni. Ogni fase eredita l'infrastruttura e il *data layer* delle precedenti, e questa continuità rende possibili i confronti.

Il tipo di evidenza cambia però da una fase all'altra. Le Fasi 2 e 3 sono confrontate su un *benchmark* comune e quantitativo. La Fase 1 produce evidenza prevalentemente qualitativa: un inventario di criticità strutturali e due soli dati quantitativi, da campagne distinte. Sono le 6 *query* su 50 terminate senza risultato utile e il tasso di allucinazione di strumento del *benchmark end-to-end* del tirocinio. Le assegniamo perciò il ruolo di *requirements elicitation*: motiva le fasi successive senza competere con esse sul piano quantitativo.

4.2 Operazionalizzazione delle domande di ricerca

Come traduciamo RQ1–RQ3 (Sezione 2.9) in qualcosa di misurabile? Ogni domanda va scomposta in costrutti, e ogni costrutto in una metrica con una sorgente di dato dichiarata. La Tabella 4.1 riassume la mappatura.

Tre criteri governano la scelta delle metriche. Il primo è la preferenza per le misure che non richiedono un giudice LLM. Dove una grandezza è calcolabile contro un riferimento noto, come la delega o il recupero puro, la calcoliamo con codice deterministico: i *bias* discussi nella Sezione 2.7 restano così fuori dalla misura. Il secondo criterio è la separazione degli stadi. Teniamo distinte le metriche di recupero da quelle di generazione, perché un unico punteggio *end-to-end* confonde due sorgenti d'errore diverse. Il terzo criterio riguarda la numerosità: riportiamo sempre l'n del sottoinsieme su cui una metrica è calcolata, anche quando è piccolo.

Due costrutti restano volutamente fuori dal perimetro quantitativo. La qualità linguistica delle risposte non è misurata da un giudice a *scorecard* né validata da annotatori umani, e su di essa non avanziamo alcuna affermazione. Delle prestazioni operative misuriamo latenza e costo, lasciando fuori il costo energetico e ambientale.

4.3 Valutazione comparativa delle piattaforme *low-code*

La Fase 1 richiede una piattaforma di prototipazione. La scelta vincola i *pattern* architetturali realizzabili nel prototipo, quindi la sottoponiamo a una valutazione strutturata su criteri espliciti.

Le piattaforme a confronto sono tre, tutte *low-code* per *workflow* basati su LLM: n8n [33], Dify [8] e Langflow [25]. I criteri sono cinque (Tabella 4.2) e ciascuno traduce un vincolo del contesto aziendale.

Tabella 4.1: Operazionalizzazione di RQ1–RQ3 in metriche osservabili, con la campagna che le produce e il capitolo che ne riporta i risultati.

RQ	Costrutto	Metrica operativa	Sorgente
RQ1	Allucinazione di strumento	Frequenza di invocazione di un agente o *tool* non esistente o non pertinente	*Benchmark* E2E di Fase 1 (Sezione 7.1.3)
RQ1	Correttezza della delega	*Routing accuracy*, *agent* F1, *precision*, *recall*	*Benchmark* di *routing* (Sezione 4.4)
RQ2	Auditabilità del processo	Ricostruibilità *ex post* di ogni decisione di *routing* dal piano serializzato e dai *checkpoint*, verificata per ispezione	Ispezione dell'artefatto (Capitolo 5)
RQ2	Tracciabilità delle scelte	Presenza, per ciascuna delle tre transizioni di fase, di un'evidenza documentata che la motiva, verificata per ispezione	Documentazione delle quattro fasi
RQ3	Affidabilità per sostrato	*Routing accuracy* e tassi di sovra-delega, sotto-delega e delega errata, per architettura	*Benchmark* di *routing*
RQ3	Manutenibilità	File toccati e ore-uomo per l'aggiunta di un agente; ampiezza della *diff* per il cambio di modello	*Log* di sviluppo (stima dell'autore)
RQ3	*Type safety*	Classi di errore intercettate alla validazione anziché a *runtime*	Ispezione dei modelli Pydantic del piano
RQ3	Qualità del recupero	Recall@1 e Recall@10, MRR, nDCG@10; metriche Ragas; *verdict accuracy*; costo e *token*	Studio comparativo (Sezione 4.5)

Il *pattern Planner-Executor* richiede logica non lineare, e da qui deriva il peso della flessibilità del *workflow*. Il prototipo deve comunicare con sistemi interni già esistenti, da cui il criterio sull'integrazione API e MCP. La piattaforma va installata sul *cluster* Kubernetes aziendale, quindi manutenibilità e *deploy* diventano dirimenti. L'ampiezza dell'ecosistema riduce il codice da scrivere a mano, grazie ai nodi di integrazione già disponibili. L'orchestrazione agentica, infine, è l'oggetto stesso della prova.

Tabella 4.2: Valutazione comparativa delle tre piattaforme *low-code*, scala 1–5. Valutazione dell'autore condotta durante il tirocinio presso E4 Analytics, su ispezione della documentazione e prove di installazione sul *cluster* interno. I punteggi esprimono l'idoneità alla **prototipazione rapida** e non si estendono a un rilascio *enterprise* definitivo.

Criterio	**n8n**	**Dify**	**Langflow**	**Nota**
Flessibilità del *workflow*	**4**	3	3	n8n: nodi IF, Switch e Merge per diramare e ricongiungere il flusso
Integrazione API / MCP	**5**	3	2	n8n: nodo HTTP Request generico e nodi *custom* scritti in TypeScript
Manutenibilità e *deploy*	**4**	**4**	3	n8n e Dify: immagini container ufficiali, installate sul *cluster* interno
Ecosistema e *community*	**5**	3	2	n8n: catalogo di nodi di integrazione nativi e nodi della *community* installabili
Orchestrazione agentica	3	**4**	3	Dify: applicazioni *Agent* predefinite, in modalità *Function Calling* o ReAct
Totale	**21**	17	13	

Selezioniamo **n8n**. Prevale su flessibilità, integrazione ed ecosistema, e perde un punto proprio sull'orchestrazione agentica, dove Dify offre astrazioni più pronte. Per il nostro scopo la perdita è accettabile. Serve un ambiente in cui costruire a mano la separazione fra pianificazione ed esecuzione e osservarne i limiti: un agente preconfezionato nasconderebbe proprio ciò che vogliamo misurare.

Il valore di questa valutazione va circoscritto. I punteggi sono attribuiti da un solo valutatore, senza un secondo giudice indipendente e senza misura di accordo. Valgono quindi come motivazione documentata della scelta e nulla di più. Si riferiscono inoltre allo stato delle tre piattaforme al momento della valutazione, cioè ai rilasci disponibili durante il tirocinio.

4.4 Protocollo del *benchmark* di *routing*

Il *benchmark* di *routing* misura una sola grandezza: la richiesta è stata affidata all'agente giusto? Lo costruiamo per rispondere in modo programmatico, senza giudice LLM.

4.4.1 Disegno del *benchmark* e *gold standard*

Il riferimento è un insieme di 100 *query* (`Q001`–`Q100`), versione 3.1 del *ground truth*, congelato prima delle esecuzioni e condiviso senza modifiche fra le due architetture. A ogni *query* è associato l'insieme di agenti attesi, scelti fra i tre disponibili (`DB`, `WEB` e `RAG`). Dove la richiesta lo prevede, il riferimento indica anche l'ordine in cui gli agenti devono intervenire.

Le 100 *query* si distribuiscono su tre livelli di difficoltà (36 facili, 40 medie e 24 difficili) e su nove *pattern* di delega (Tabella 4.3). Di queste, 81 sono formulazioni distinte e 19 sono parafrasi di *query* già presenti. Le parafrasi misurano la robustezza al *wording*: verificano se il sistema instrada allo stesso modo due domande equivalenti scritte in modo diverso. Fra le formulazioni difficili includiamo *query* avversarie. Ciascuna contiene una superficie lessicale che orienta verso un agente, mentre l'intento reale della domanda ne richiede un altro.

Tabella 4.3: I nove *pattern* di delega del *gold standard* v3.1 e la loro numerosità.

Pattern	**Delega attesa**	*n*	**Che cosa mette alla prova**
`B_DB`	{DB}	12	Riconoscimento del dato *enterprise* strutturato
`B_RAG`	{RAG}	18	Riconoscimento della documentazione interna
`B_WEB`	{WEB}	12	Riconoscimento dell'informazione esterna
`D_DB_RAG`	{DB, RAG}	11	Delega parallela su due fonti interne
`D_DB_WEB`	{DB, WEB}	9	Delega parallela interno più esterno
`D_RAG_WEB`	{RAG, WEB}	9	Delega parallela documentale più esterno
`C_SEQ`	DB → WEB	9	Dipendenza sequenziale: il secondo agente opera sull'*output* del primo
`E`	∅	9	*Guardrail*: la richiesta non va delegata
`ADV`	varia	11	Superficie lessicale contraria all'intento reale
Totale		**100**	

La nomenclatura dei *pattern* segue il numero di agenti attesi. I `B` ne richiedono uno solo, i `D` due agenti indipendenti in parallelo, `C_SEQ` due agenti in sequenza con dipendenza di dato, `E` nessun agente.

Gli ultimi due *pattern* hanno una funzione diagnostica distinta. Il *pattern* `E` verifica il comportamento opposto a quello che di solito si misura, cioè la capacità di **non** delegare quando la richiesta è conversazione generica o cade fuori dal perimetro. Un sistema che delega sempre otterrebbe un buon punteggio sugli altri *pattern* e fallirebbe qui. Il *pattern* `ADV` verifica se la delega segue l'intento oppure la superficie lessicale della domanda. Sono i due *pattern* che distinguono un instradamento robusto da uno guidato dalle parole chiave.

4.4.2 Strumentazione e definizione delle metriche

Le due architetture registrano la propria delega in modi diversi, e la normalizziamo prima di qualunque confronto. La v2 la dichiara nel campo `tool` dei `WorkflowStep` del piano. La v3 la espone come sequenza di deleghe `task` dell'*Orchestrator*. Un *adapter* per sistema riconduce i nomi nativi ai tre nomi canonici `DB`, `WEB` e `RAG`, e registra per ogni invocazione l'istante di inizio e di fine. Deduciamo il parallelismo da quegli istanti: due agenti risultano paralleli quando i rispettivi intervalli di esecuzione si sovrappongono.

Ogni metrica confronta due elenchi di agenti: quelli che il sistema ha attivato davvero e quelli che il *gold standard* si attende per quella *query*. Da questo confronto ricaviamo cinque grandezze.

- ***Routing accuracy***: la frazione di *query* in cui i due elenchi coincidono esattamente. È una misura severa, perché non ammette punteggi parziali: un agente in più o in meno conta come errore pieno;
- ***Agent precision* e *recall***: considerano i singoli agenti anziché l'elenco nel suo insieme. Per ogni *query* la *precision* è la quota di agenti attivati che erano effettivamente attesi, il *recall* è la quota di agenti attesi che il sistema ha attivato. Il valore riportato è la media sulle *query* valutate. Quando né il *gold standard* né il sistema prevedono agenti, come in un *pattern* `E` risolto correttamente, entrambe valgono 1. L'***agent* F1** è la media armonica delle due, calcolata anch'essa per *query* e poi mediata. Le tre grandezze assegnano un punteggio anche alle deleghe parzialmente corrette, che la *routing accuracy* scarta per intero;
- **Sovra-delega, sotto-delega e delega errata**: scompongono l'errore in tre forme distinte. Il sistema sovra-delega quando attiva tutti gli agenti attesi e ne aggiunge almeno un altro. Sotto-delega quando ne attiva solo una parte, senza però attivarne di sbagliati. Delega in modo errato negli altri casi, cioè quando attiva agenti non previsti e allo stesso tempo ne trascura di previsti. Le tre categorie sono disgiunte e ogni errore ricade in una sola di esse;
- **Accuratezza sequenziale**: si calcola sul solo *pattern* `C_SEQ`. Misura la frazione di *query* in cui la sequenza osservata coincide, fase per fase, con quella attesa: stesso numero di fasi e stessi agenti in ciascuna fase;
- **Robustezza semantica**: verifica se formulazioni equivalenti ricevono la stessa delega. Le 19 parafrasi si aggregano con le rispettive formulazioni originali in 16 gruppi di intento,

per un totale di 35 *query*. Su ogni gruppo misuriamo la concordanza in due modi. La forma stretta è soddisfatta solo quando tutte le *query* del gruppo vanno agli stessi agenti. La forma Jaccard è più permissiva: media la sovrapposizione fra le coppie del gruppo, quindi assegna un valore anche alle concordanze parziali. I risultati sono nel Capitolo 7 (Tabella 7.2).

La valutazione è programmatica: nessun modello linguistico interviene nella misura, e il confronto avviene fra etichette di agente. Il *benchmark* risulta così immune ai *bias* del paradigma *LLM-as-Judge* (Sezione 2.7), che restano rilevanti nella campagna sul recupero.

Gli intervalli di confidenza al 95% sono stimati con *bootstrap* percentile su 5.000 ricampionamenti, con seme fissato a 42 (`BOOTSTRAP_SEED` in `benchmarks/routing_benchmark/metrics.py`). La loro portata va precisata. Catturano la variabilità dovuta al campione di *query* e lasciano fuori quella dovuta alla natura stocastica dei modelli. Entrambe le architetture sono eseguite in una sola *run*, quindi non stimiamo la stabilità fra esecuzioni ripetute. È un limite del disegno e lo dichiariamo qui.

4.5 Protocollo dello studio comparativo esteso: SciFact, HotpotQA e MultiHop-RAG

Il recupero delle informazioni è valutato con una campagna comparativa controllata. Il disegno consente a un terzo di ripetere la misura, perché sottoinsiemi e semi sono congelati e le metriche di recupero sono calcolate da codice deterministico (Sezione 4.6).

Il confronto riguarda tre sistemi RAG su tre dataset, con un giudice indipendente dal generatore. Il primo sistema è un RAG classico di riferimento, a recupero vettoriale puro. Il secondo è un RAG avanzato: ibrido con BM25, fusione RRF e *reranking cross-encoder* (Sezione 2.6). Il terzo è un agente su *file system* e adotta la *Direct Corpus Interaction* introdotta nella Sezione 2.5.2.

4.5.1 I tre dataset e la costruzione del sottoinsieme

I tre dataset sollecitano il recupero in modi diversi. **SciFact** [54] è un compito di verifica di affermazioni scientifiche. Il sottoinsieme contiene 100 *claim* e 310 *abstract*: i 110 rilevanti per quelle *claim* e 200 distrattori difficili. I distrattori sono gli *abstract* più simili al centroide dei documenti rilevanti, selezionati per similarità coseno con `qwen3-embedding-0-6b` su un

campione casuale di 1.000 documenti del *corpus*. Con distrattori così prossimi, il recupero deve distinguere documenti vicini nello spazio semantico.

HotpotQA [63] è multi-*hop* e include distrattori nativi: 10 paragrafi per *query*, di cui solo due rilevanti, su un *pool* di 1.000 documenti. **MultiHop-RAG** [51] è costruito su documenti interi anziché su paragrafi isolati, con 609 articoli di notizie. È per questo la *run* più costosa e più lenta della campagna (Capitolo 7).

Per rendere confrontabili i tre sistemi congeliamo lo stesso sottoinsieme di 100 *query* per ciascun dataset, estratto con seme 42. Un unico costruttore condiviso genera *query* e distrattori, così che i tre sistemi ricevano in ingresso le stesse domande e lo stesso *pool* di candidati. La comparabilità deriva dalla costruzione del sottoinsieme, a monte di qualunque analisi.

4.5.2 Le due *milestone* di misura e il verdetto di classificazione

La misura avviene su due livelli distinti. Il primo, *retrieval*, valuta solo il recupero con Recall@1 e Recall@10, MRR e nDCG@10 (Sezione 2.6.5). Questi valori sono calcolati localmente contro le relazioni di rilevanza note, senza alcun giudice LLM. Il secondo livello, *answer*, valuta la risposta generata con le metriche Ragas [9] e ne rileva il costo in euro, la latenza e i *token* di ingresso.

La separazione è deliberata e applica il secondo criterio della Sezione 4.2. Il primo livello isola la qualità del recupero dal rumore del generatore e del giudice, e possiamo così attribuire un eventuale errore allo stadio che lo produce.

SciFact impone una metrica in più. Non ha una risposta di riferimento testuale, perché il compito è classificare un'affermazione come sostenuta, smentita o priva di evidenza sufficiente. La correttezza diventa allora *verdict accuracy*, cioè la corrispondenza esatta fra il verdetto prodotto e l'etichetta del dataset originale (*split* `dev`).

Per i due sistemi vettoriali il verdetto è un'etichetta strutturata, facile da confrontare. L'agente su *file system* risponde invece in prosa libera. Per questo sistema la *verdict accuracy* richiede un classificatore LLM dedicato, `minimax-m3`, che legge la risposta generata e la riconduce a una delle tre etichette. Il classificatore si limita a riconoscere l'etichetta nel testo, quindi lo teniamo distinto dal giudice che valuta la qualità generativa.

4.5.3 Giudice indipendente e integrità della misura

Il giudice Ragas è `minimax-m3` (MiniMax-M3), un modello di famiglia diversa dal generatore `qwen36-27b-awq-int4` (Qwen3.6-27B). La separazione dei due ruoli attenua il *self-evaluation bias* del paradigma *LLM-as-Judge* (Sezione 2.7), che emerge quando giudice e generatore condividono lo stesso modello. È la sola mitigazione strutturale applicabile alle metriche Ragas, e lascia scoperti i *bias* residui del giudice.

Abbiamo verificato la *pipeline* di misura prima di leggere i risultati. Durante la campagna su MultiHop-RAG sono emersi cinque difetti che alteravano i punteggi senza generare errori visibili, tutti corretti a monte dell'analisi:

- una metrica di correttezza istanziata senza il suo modulo di similarità;
- un limite di *token* in uscita troppo basso, che troncava il giudizio;
- i valori `NaN` inclusi nel calcolo della media;
- una contaminazione linguistica fra italiano e inglese nel *prompt* di generazione;
- un *timeout* HTTP che introduceva risposte d'errore poi valutate come valide.

L'elenco documenta un controllo di integrità della strumentazione, e come tale appartiene ai requisiti di riproducibilità. I dettagli numerici e le figure della campagna sono nel Capitolo 7 (Sezione 7.2).

4.5.4 Il quarto braccio: agente su *file system* con e senza GRASP

Un quarto braccio sperimentale misura l'effetto della politica di lettura GRASP, adattata via *prompt* dal lavoro di Gandhi et al. [12] (Capitolo 6, Sezione 6.6.1), sull'agente su *file system*. Ripetiamo l'intera campagna sui tre dataset con la politica attiva, a parità di sottoinsieme, di giudice e di generatore. L'unica variabile che cambia è il comportamento predefinito della lettura. Al posto del `read` integrale del file, l'agente riceve i parametri `offset` e `limit` calcolati attorno alla riga individuata dal `grep`: la lettura parte circa quindici righe prima della corrispondenza e ne copre circa trenta.

Il confronto isola così l'effetto della finestra mirata rispetto alla lettura integrale, e usa le stesse metriche delle sezioni precedenti. L'ipotesi riguarda il rapporto fra consumo e copertura.

Una lettura più stretta dovrebbe ridurre i *token* di ingresso, e con essi costo e latenza, ma potrebbe abbassare la Recall@10. Riportiamo entrambi gli esiti anche quando il secondo è sfavorevole.

4.6 Riproducibilità

La ripetibilità del lavoro poggia su quattro condizioni. I sottoinsiemi di *query* delle due campagne sono congelati e versionati prima delle esecuzioni, e non li abbiamo modificati dopo aver visto i risultati. Le versioni dei componenti *software* e dell'infrastruttura sono dichiarate nel Capitolo 7 (Tabella 7.1). Il seme dei ricampionamenti *bootstrap* è fissato a 42. Le metriche di recupero e quelle di *routing* sono calcolate da codice deterministico, quindi riproducibili a partire dagli stessi *log* di esecuzione.

Resta un limite di riproducibilità. L'inferenza è eseguita su un *endpoint on-premises* con campionamento attivo, quindi la ripetizione di una campagna non restituisce gli stessi *output* carattere per carattere. Il procedimento di misura resta riproducibile, mentre la singola risposta generata varia da un'esecuzione all'altra.

Nel capitolo successivo presentiamo l'architettura.

5. Architettura del sistema: orchestrazione deterministica (LangGraph)

Questo capitolo descrive l'architettura di *Agentic Platform* nella sua seconda versione, l'orchestratore deterministico costruito su LangGraph. La chiamiamo v2 perché deriva dal prototipo n8n della fase precedente. Il criterio di progetto assegna la pianificazione al modello linguistico e l'esecuzione al codice.

L'impostazione riprende la pianificazione esplicita prima dell'azione, formulata nel *Plan-and-Solve prompting* [56] e nella pianificazione interattiva per agenti multi-compito [57]. Il modello genera il piano in forma strutturata e assume così il ruolo che nella pianificazione classica spetta al risolutore simbolico [11]. L'attuazione passa invece a componenti software deterministiche. La migrazione al *framework* dichiarativo OpenCode e l'ottimizzazione del recupero con *Direct Corpus Interaction* sono il tema del Capitolo 6.

Le sei criticità rilevate su n8n (Capitolo 1, Tabella 1.1) hanno una causa comune. Quando il modello decide e agisce nello stesso passo, il comportamento del sistema diventa difficile da prevedere. Abbiamo perciò assunto la separazione fra ragionamento ed esecuzione come vincolo di progetto. Ne derivano quattro scelte. La pianificazione resta distinta dall'attuazione, e il piano ha una forma esplicita, validata prima dell'esecuzione. L'orchestrazione avviene su un grafo di stato che gestisce gli errori, e lo stato del grafo viene salvato a ogni passo. L'esecuzione diventa così osservabile e, nella *Deep Research*, ammette punti di controllo umano.

La Tabella 5.1 associa a ogni criticità il componente della v2 che la copre e la sezione che lo descrive. Una sola voce ha copertura parziale: l'aggiornamento incrementale verso il *client* esiste solo sul canale della *Deep Research*, mentre l'orchestrazione principale restituisce una risposta unica (Sezione 5.6.1).

5.1 Vista d'insieme dell'architettura

L'architettura si compone di sei blocchi funzionali: la *Semantic Cache*, il *Planner*, il *Route Executor*, i tre agenti specializzati, il *Deep Research Agent* e il *Synthesizer*. La *Semantic Cache* opera in ingresso al grafo e intercetta le richieste già processate. Segue il *Planner*, affidato a un LLM, che interpreta la domanda dell'utente e ne ricava un piano. Il piano di *workflow* è

Tabella 5.1: Le sei criticità rilevate sul prototipo n8n e il componente della v2 che vi risponde. La voce 5 ha copertura parziale, dichiarata nella colonna di destra.

#	Criticità (Fase 1)	Risposta nella v2	Stato e sezione
1	Loop di esecuzione non controllati	Tetto di tre riformulazioni sulla costante `_MAX_REPLAN_ATTEMPTS`, poi *graceful degradation* verso il *Synthesizer*	Risolta: terminazione garantita per costruzione (Sez. 5.5)
2	Gestione non ottimale del contesto RAG	*Digest* di contesto per *step*, al posto del riuso integrale della cronologia	Implementata, riduzione dei *token* non misurata (Sez. 5.7)
3	Scarsa trasparenza del processo decisionale	Piano Pydantic ispezionabile con campo `reason`, transizioni persistite dal *checkpointer*	Risolta (Sez. 5.4 e 5.7)
4	Assenza di *caching* semantico	Nodo *Semantic Cache* in ingresso al grafo, soglia coseno 0,95 su indice pgvector	Risolta: soglia conservativa; ricerca vettoriale entro il requisito di latenza al P95, calcolo dell'*embedding* escluso (Sez. 5.3)
5	Mancanza di aggiornamento incrementale verso il *client*	Canale a eventi SSE che riporta l'avanzamento della ricerca	Parziale: solo sulla *pipeline Deep Research* (Sez. 5.6.1)
6	Assenza di persistenza del contesto utente	*Checkpointer* PostgreSQL sullo stato completo del *workflow*	Risolta, senza limite temporale alla ripresa (Sez. 5.8)

un modello dati validato con Pydantic, ed è l'unico passaggio fra il *Planner* e il codice che lo esegue [5].

Dal piano in poi l'esecuzione è affidata a codice Python. Il *Route Executor* esegue i passi, anche in parallelo, e li delega a tre agenti specializzati: il *DB Agent*, il *Web Search Agent* e il *RAG Agent*. Il *Deep Research Agent* opera su un percorso separato, attivato direttamente dall'utente. Il *Synthesizer*, di nuovo un LLM, raccoglie i risultati e scrive la risposta per l'utente.

La Figura 5.1 illustra l'interazione fra questi componenti, insieme al percorso di errore che riporta il controllo al *Planner*.

Una richiesta attraversa il grafo in quest'ordine. La *Semantic Cache* verifica se una risposta equivalente è già disponibile e, in caso affermativo, evita la pianificazione. Altrimenti il *Planner* genera un piano in formato JSON, validato rispetto allo schema Pydantic. Il *Route Executor* manda in esecuzione gli *step* indipendenti in parallelo, tramite la `Send` API di LangGraph, e raccoglie i risultati nello stato condiviso. Il *Synthesizer* compone la risposta. In caso di errore il *Feedback Loop* torna al *Planner*, per un massimo di tre tentativi.

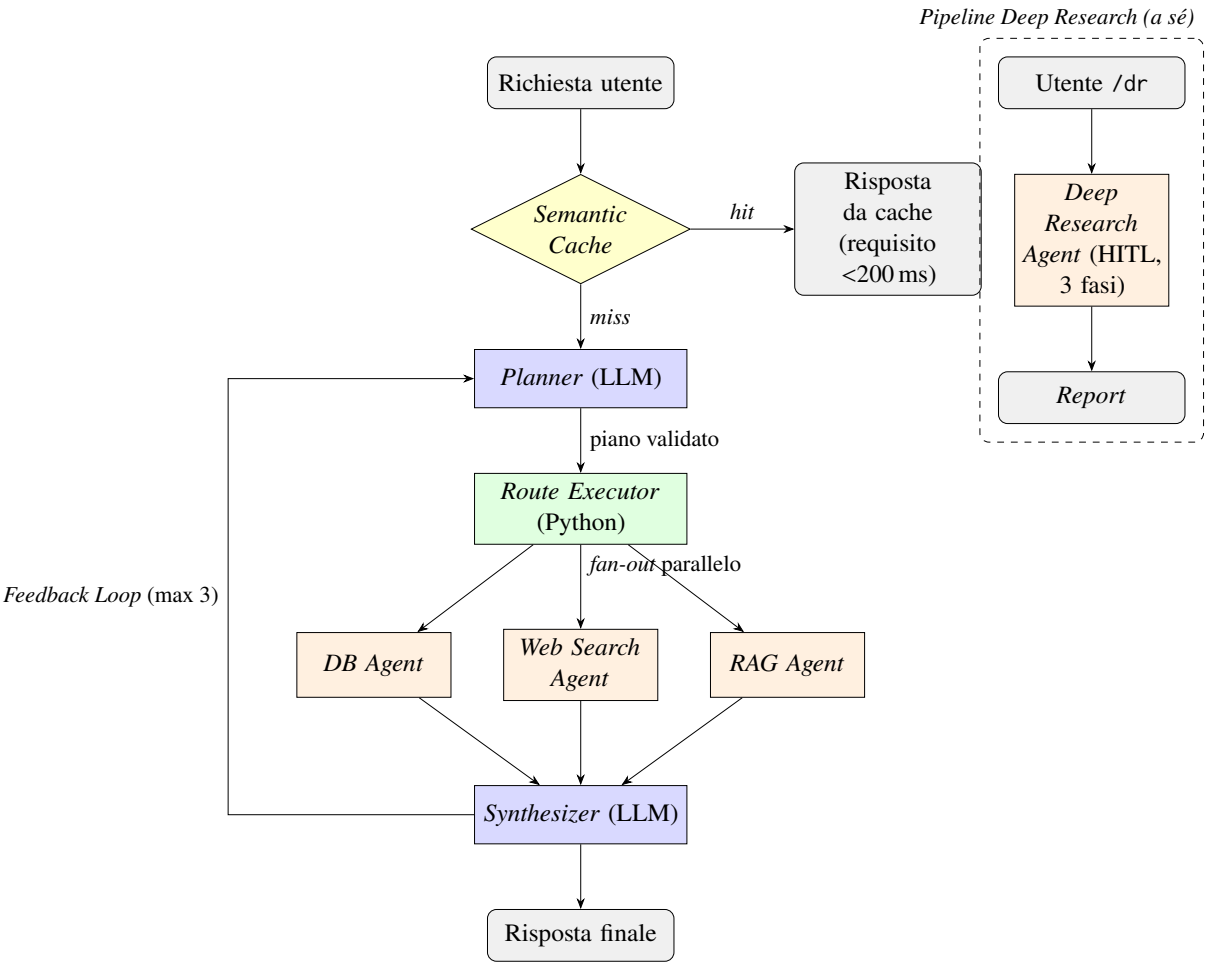


Figura 5.1: Vista logica del flusso agentico nell'architettura v2 di *Agentic Platform*, con il *Feedback Loop* verso il *Planner* limitato a tre cicli di revisione. Il riquadro tratteggiato isola la *pipeline Deep Research*, attivata dall'utente con `/dr` e indipendente dall'orchestrazione principale. Legenda: blu = nodi inferenziali (LLM), verde = nodi deterministici (Python), arancione = agenti.

Nel percorso di orchestrazione il modello linguistico interviene in due soli nodi, il *Planner* e il *Synthesizer*. Dentro gli agenti restano chiamate inferenziali circoscritte: il *DB Agent* sceglie il motore di recupero e genera la *query*, il *Web Search Agent* valida i risultati della ricerca, il *RAG Agent* calcola l'*embedding* della richiesta. Nessuna di queste chiamate sceglie quale agente invocare, perché quella decisione è già fissata nel piano validato. L'instradamento fra i nodi resta quindi deterministico, e ogni transizione del grafo è ispezionabile.

5.2 Separazione fra ragionamento ed esecuzione

L'architettura separa due livelli. Il livello di ragionamento interpreta la richiesta, pianifica e formula la risposta. Il livello di esecuzione svolge le operazioni concrete con codice deterministico. Se un modello interpreta, sceglie il passo successivo, compone i comandi e mantiene lo stato, l'esecuzione non è più verificabile passo per passo. La letteratura propone la decomposizione

esplicita in passi per migliorare l'accuratezza sui compiti che richiedono più passaggi [56, 57].

Nel nostro sistema il *Planner* riceve la descrizione degli strumenti e il contesto utile, e restituisce un piano che validiamo prima di accettarlo. Da quel momento il controllo passa al codice Python. Il modello Pydantic `WorkflowPlan` è l'interfaccia fra il *Planner* e il codice di esecuzione. Uno strumento inesistente nel campo `tool` produce un `ValidationError` prima di qualsiasi invocazione. Il campo `reason` conserva la motivazione del piano, e ogni passo resta ricostruibile dai suoi `inputs` e dalle dipendenze dichiarate. Il Capitolo 7 verifica questa separazione sul *routing benchmark*.

5.3 La *Semantic Cache*

In un contesto aziendale molte richieste si ripetono, e ciascuna costa almeno due chiamate al modello, al *Planner* e al *Synthesizer*. La *Semantic Cache* è il primo nodo del grafo e governa l'arco condizionale verso il *Planner*. Quando riconosce una richiesta già processata, restituisce la risposta memorizzata e salta l'intero percorso di orchestrazione.

Il nodo converte la domanda dell'utente in un *embedding* a 1.024 dimensioni con il modello BAAI/bge-m3 [3]. Cerca poi sull'indice pgvector [40] la richiesta passata con la similarità coseno più alta. Se la similarità supera 0,95 il nodo restituisce la risposta memorizzata; altrimenti il flusso prosegue verso il *Planner*. L'unica chiamata a un modello è il calcolo dell'*embedding*, affidato a un modello distinto da quello generativo.

Come fissiamo la soglia? Una soglia bassa riutilizza più risposte, ma aumenta il rischio di restituirne una non pertinente. Il catalogo contiene famiglie di prodotti che differiscono per una sola sigla, come il Dell R750 e il Dell R650. Due richieste di prezzo su questi server hanno in comune tutte le parole tranne una, e una soglia permissiva le tratterebbe come equivalenti: la cache restituirebbe il prezzo dell'altro server. Per questo scegliamo una soglia conservativa, 0,95, e accettiamo che richieste equivalenti formulate in modo distante non vengano riconosciute. Non abbiamo misurato la similarità coseno di coppie di questo tipo con bge-m3, quindi non sappiamo se 0,95 basta a separarle.

Il secondo parametro di progetto è la latenza. Il *lookup* ha senso solo se richiede una frazione del tempo del percorso completo. Nel caso più semplice della v2, con due chiamate al modello, quel percorso richiede 2,2 secondi (Sezione 5.9). Il requisito per il *lookup* è quindi di 200 millisecondi, meno di un decimo di quel tempo.

La Tabella 5.2 riporta la verifica del requisito sulla sola ricerca vettoriale. Abbiamo misurato la ricerca coseno sui primi cinque vicini, su una tabella di mille *embedding* a 1.024 dimensioni e con venti connessioni concorrenti. La mediana è di 104 millisecondi e il novantacinquesimo percentile di 196, entro il requisito; il novantanovesimo, 281 millisecondi, lo supera. Le cifre includono un *round-trip* di rete verso un host remoto, che da solo vale fra i 50 e i 60 millisecondi. La misura esclude invece il calcolo dell'*embedding* della domanda, che il nodo esegue prima della ricerca e che non abbiamo cronometrato.

Tabella 5.2: Latenza della ricerca coseno sui primi cinque vicini, misurata su PostgreSQL con pgvector, mille *embedding* a 1.024 dimensioni e venti connessioni concorrenti.

Operazione	Operazioni	ops/s	P50	P95	P99
Ricerca coseno, primi 5 vicini	5.074	169	104 ms	196 ms	281 ms

Nessun errore su 5.074 operazioni. La misura isola la *query* vettoriale: il calcolo dell'*embedding* della domanda, la lettura della risposta memorizzata e la restituzione al *client* restano fuori dal conteggio.

5.4 Il piano di *workflow*: il modello `WorkflowPlan`

Il piano passa dal *Planner* al codice attraverso modelli Pydantic, che ne fissano campi e tipi. L'impostazione richiama il *design by contract*, dove i componenti software interagiscono attraverso interfacce formalizzate [32]. Nei sistemi LLM questo significa validare l'output del modello prima di usarlo.

Il piano prodotto dal *Planner* è un oggetto `WorkflowPlan`, cioè una lista di passi atomici, i `WorkflowStep`, con quattro campi aggiuntivi: `version`, `workflow_summary`, `relevant_chat_history` e `reason`. La Tabella 5.3 raccoglie i campi dei tre modelli del piano, `WorkflowStep`, `WorkflowPlan` ed `ErrorState`.

Quattro campi governano il comportamento del piano. Il campo `reason` del `WorkflowPlan` registra la motivazione complessiva, a supporto dell'audit. In `WorkflowStep`, `parallel_group` segnala gli *step* avviabili insieme e `depends_on` dichiara quelli propedeutici. Il campo `tool` ha un tipo chiuso, un `Literal` sui tre agenti registrati sul grafo. Se il *Planner* produce uno strumento inesistente, la validazione solleva un `ValidationError` prima che un agente venga invocato. Il nodo ritenta allora la generazione fino a due volte; se anche l'ultimo tentativo fallisce, restituisce un piano vuoto e il grafo passa direttamente al *Synthesizer*.

Sulle dipendenze la validazione è più debole. Lo schema Pydantic tipizza `depends_on` come lista di stringhe. Non verifica che gli identificatori esistano nel piano, né esclude i cicli. L'ordine lo impone il *Route Executor*, che manda in esecuzione uno *step* solo quando tutti i suoi

Tabella 5.3: Campi dei modelli Pydantic del piano: `WorkflowStep`, `WorkflowPlan` ed `ErrorState`.

Campo	Tipo	Descrizione
`WorkflowStep`		
`step_id`	str	Identificatore univoco dello *step* (alias `id`)
`tool`	`Literal`	`db_agent` | `web_search_agent` | `rag_agent`
`inputs`	dict	`query` per l'agente e *digest* di contesto (`digest`)
`depends_on`	list[str]	*Step* propedeutici (per `step_id`)
`parallel_group`	str opz.	Gruppo di esecuzione parallela
`relevant_chat_history`	str	Identificatori di turno o `NONE` (default)
`pass_full_history`	bool	Invia la cronologia integrale al posto del *digest* (default `False`)
`WorkflowPlan`		
`version`	str	Versione dello schema (default `1.0`)
`workflow_summary`	str	Sintesi dell'obiettivo per la cronologia
`relevant_chat_history`	str	Contesto globale: identificatori di turno o `NONE`
`steps`	list[WorkflowStep]	Passi del piano
`reason`	str	Motivazione complessiva del piano
`ErrorState`		
`failed_step_id`	str	*Step* che ha generato l'errore
`error_message`	str	Messaggio descrittivo
`stack_trace`	str	Traccia tecnica dell'eccezione

`depends_on` hanno già prodotto un risultato. Se nessuno *step* residuo soddisfa la condizione, il grafo passa al *Synthesizer* con i risultati raccolti, senza sollevare eccezioni.

Quali errori intercetta, allora, il piano tipizzato? Una sola classe, i nomi di strumento inesistenti. Nessun controllo verifica la correttezza del grafo delle dipendenze, e un piano che omette una dipendenza necessaria è valido e viene eseguito. Il *Feedback Loop* interviene solo se lo *step* anticipato fallisce e produce un `ErrorState`. Il Capitolo 7 documenta un caso reale, in cui il *Planner* colloca in parallelo due *step* che andavano eseguiti in sequenza.

Nell'implementazione il *Planner* è invocato tramite l'API OpenAI-compatibile di vLLM, con Qwen3-Coder-30B-A3B-Instruct nella fase di sviluppo dell'architettura. Il codice del nodo è indipendente dal modello servito. Quando lo *stack* di *serving* aziendale è passato a Qwen3.6-27B-AWQ-INT4, abbiamo rimisurato la v2 su quel modello per il confronto con la v3 (Capitolo 7, Sezione 7.1.1). Lo schema `WorkflowPlan` viene imposto per *function calling*, a temperatura nulla. Il modello riceve la struttura come firma di funzione e ne produce gli argomenti, quindi il piano arriva già come oggetto Pydantic validato.

La Tabella 5.4 riporta un piano reale, quello dello scenario B001, in cui chiediamo il prezzo di un server Dell R750 in dollari. Il piano recupera due dati indipendenti: il prezzo in euro

dal catalogo e il tasso di cambio aggiornato. Entrambi gli *step* hanno `depends_on` vuoto e ricadono nello stesso `parallel_group`, quindi partono insieme. La tabella riporta i soli campi di `WorkflowStep`: `workflow_summary` e `reason` restano nell'oggetto serializzato e servono all'audit *ex post*.

Tabella 5.4: Passi del piano `WorkflowPlan` generato dal *Planner* per lo scenario B001.

step_id	**tool**	**inputs** (*digest*)	**parallel_ group**	**depends_on**
`db_1`	`db_agent`	Recupera il prezzo in EUR del server Dell R750 dal catalogo interno	`pg_1`	(vuoto)
`web_1`	`web_search_agent`	Recupera il tasso di cambio EUR/USD aggiornato per la conversione valutaria	`pg_1`	(vuoto)

5.5 Il grafo di stato: *Route Executor* e *Feedback Loop*

Tutti questi componenti sono nodi di un grafo di stato LangGraph [24]. Il grafo rappresenta in modo esplicito i nodi inferenziali, i nodi deterministici e le transizioni fra loro. I nodi leggono e scrivono uno stato condiviso, che contiene la cronologia dei messaggi, il piano corrente, i risultati accumulati, l'eventuale errore e il contatore delle riformulazioni.

Il *fan-out* parallelo pone un problema di scrittura concorrente: più agenti terminano nello stesso momento e aggiornano lo stesso campo dello stato. Lo risolviamo con due *reducer* dichiarati sui campi interessati. Il primo, `reduce_step_results`, esegue un *deep-merge* dei dizionari restituiti dagli agenti paralleli. Il secondo, `reduce_error_state`, applica una politica *first-error-wins* e conserva il primo errore osservato. Ogni *step* ha un identificatore unico e viene eseguito una sola volta per piano, quindi la fusione non genera conflitti sulle chiavi.

Il *Route Executor* riceve il piano validato, controlla le dipendenze e seleziona l'agente in base al campo `tool`, con logica interamente Python. L'instradamento non richiede alcuna inferenza, perché il valore del campo coincide con il nome del nodo registrato sul grafo. Il *Route Executor* restituisce quel nome come destinazione dell'arco condizionale, oppure lo passa alla `Send` API quando gli *step* pronti sono più di uno.

L'aggiunta di un nuovo agente richiede quattro interventi: la funzione che implementa l'agente, la sua registrazione come nodo del grafo, l'estensione del tipo `Literal` di `tool` e l'arco di ritorno verso il *Route Executor*. Sul *log* di sviluppo l'operazione tocca quattro file e costa fra le 8 e le 12 ore-uomo, senza modifiche alla logica di orchestrazione. Il dato viene da una nostra

unica rilevazione, senza replica indipendente, e vale come ordine di grandezza. Il Capitolo 6 confronta questo costo con quello della v3 dichiarativa.

L'esecuzione procede per iterazioni. A ogni iterazione il *Route Executor* calcola gli *step* pronti, secondo la regola sulle dipendenze descritta nella Sezione 5.4. Quando più *step* pronti condividono lo stesso `parallel_group`, partono insieme tramite la `Send` API di LangGraph. Con una dipendenza l'iterazione si riduce a un singolo *step*, e l'ordine dichiarato viene rispettato. Abbiamo preferito la `Send` API a un generico `asyncio.gather` perché mantiene coerente lo stato condiviso e produce un *checkpoint* al termine di ogni agente.

Quando un agente fallisce, scrive l'eccezione in un `ErrorState` e il *Route Executor* attiva il *Feedback Loop*. Il controllo torna al *Planner*, che riceve il contenuto dell'`ErrorState` e genera un piano correttivo. Il codice confronta il contatore `replan_count` con la costante `_MAX_REPLAN_ATTEMPTS`, pari a tre (Figura 5.2).

Quanti tentativi concediamo al *Planner*? Ogni riformulazione aggiunge una chiamata al modello. Su un errore persistente, come un servizio esterno non raggiungibile, un nuovo piano non cambia l'esito. Un tetto di uno o due tentativi lascerebbe invece meno occasioni di correggere il piano con il messaggio d'errore. Non abbiamo misurato a quale tentativo i piani convergono, quindi il valore tre resta una scelta di progetto. Esaurite le tre riformulazioni, il *Synthesizer* riceve i risultati raccolti nell'ultimo tentativo e dichiara i dati che mancano. Il tetto fisso garantisce la terminazione in un numero finito di passi.

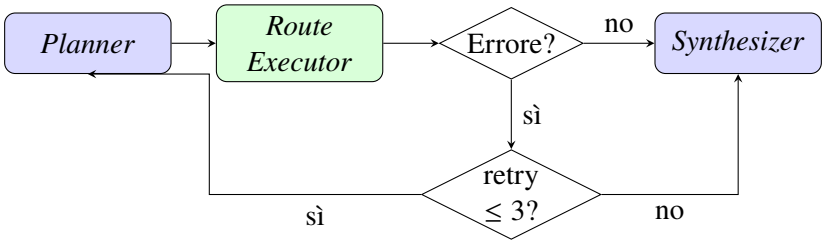


Figura 5.2: Macchina a stati del *Feedback Loop*. In caso di errore, il controllo torna al *Planner* per un massimo di tre iterazioni; esaurite le iterazioni, il sistema esegue *graceful degradation* verso il *Synthesizer*.

L'ultimo nodo inferenziale è il *Synthesizer*, servito dallo stesso modello del *Planner*. Riceve la conversazione e i risultati degli *step*, ciascuno etichettato con l'agente di origine, e compone la risposta finale. Il suo *prompt* di sistema fissa le regole per i dati mancanti: se un risultato è vuoto o assente, la risposta dichiara che il dato non è stato recuperato e non lo stima. Il *Synthesizer* non ha accesso a strumenti o basi di dati e lavora solo sui risultati già raccolti.

5.6 Gli agenti specializzati e la *Deep Research*

Quattro agenti svolgono il lavoro operativo (Tabella 5.5). Il *DB Agent*, il *Web Search Agent* e il *RAG Agent* ricevono gli *step* dal *Route Executor* e scrivono il proprio risultato nello stato; la risposta per l'utente spetta al *Synthesizer*. Il *Deep Research Agent* gestisce le richieste che prevedono un'interazione con l'utente. Per questo segue un percorso separato dal *Route Executor* e produce un *report* (Sezione 5.6.1).

Tabella 5.5: Agenti specializzati dell'architettura v2, con funzione e canale di comunicazione.

Agente	Funzione	Canale
DB Agent	SQL + *vector search* sul catalogo prodotti	AsyncOpenAI + pgvector
Web Search Agent	Ricerca *web* + validazione LLM	MCP *Streamable* HTTP
RAG Agent	Ricerca semantica sulla *knowledge base*	AsyncOpenAI + pgvector
Deep Research Agent	Ricerca in più fasi con HITL	MCP + SSE + *Sampling*

Il *DB Agent* accede in sola lettura al catalogo prodotti su PostgreSQL, organizzato in due tabelle, una per i dati strutturati e una per gli *embedding*. Il suo *prompt* chiede al modello di scegliere fra due motori di recupero in base all'intento della richiesta. Una *query* SQL serve quando occorrono filtri precisi su prezzo, categoria o disponibilità. La ricerca vettoriale su pgvector serve quando la richiesta è vaga o orientata al caso d'uso ("una workstation potente per l'AI"), dove il confronto lessicale fallirebbe. Se il motore scelto non restituisce risultati, l'agente ripiega sull'altro. La scelta del motore è quindi inferenziale, ma resta interna all'agente e non cambia l'instradamento del grafo.

Il *RAG Agent* gestisce il recupero semantico sulla *knowledge base* interna. Converte la richiesta in un *embedding* e cerca su pgvector i frammenti con similarità coseno più alta, scartando quelli sotto una soglia configurabile. Il *Web Search Agent* interroga il *web* attraverso il protocollo MCP [1], poi una chiamata LLM scarta i risultati non pertinenti alla richiesta.

Il *Web Search Agent* e il *Deep Research Agent* raggiungono i servizi esterni con il *Model Context Protocol*, costruito su JSON-RPC. Per la ricerca *web* basta il trasporto *Streamable* HTTP, con richiesta e risposta in un unico scambio. La *Deep Research* usa invece un canale a eventi, perché l'esecuzione si interrompe e riprende più volte. Su quel canale il *server* delega al *client* l'invocazione del modello, tramite il meccanismo di *Sampling*. Così il *server* non conserva credenziali e usa l'infrastruttura vLLM già autenticata.

5.6.1 La *Deep Research* e il controllo umano

Nella *Deep Research* l'utente entra nel ciclo di controllo. Il flusso parte con un comando esplicito, `/dr` o uno dei suoi alias, che lo separa dalle domande a turno singolo e salta *Semantic Cache* e *Planner*. L'esecuzione procede in tre fasi. Nella prima il modello propone alcune domande di chiarimento per delimitare l'ambito, e il flusso si ferma in attesa delle risposte. Sulla base di quelle risposte il modello costruisce poi un piano di ricerca e si ferma di nuovo, perché l'utente lo validi o lo corregga. Nella seconda fase il nodo trasforma il piano approvato in una *query* e invoca lo strumento di ricerca via MCP, su un canale SSE che riporta lo stato di avanzamento. Nella terza fase il nodo restituisce il *report* prodotto dallo strumento.

Le due fermate usano la primitiva `interrupt()` di LangGraph. Il nodo salva lo stato tramite il *checkpointer* e restituisce al *client* le domande o il piano. Quando arriva la risposta dell'utente, l'esecuzione riprende da quel punto. Durante la seconda fermata l'utente può modificare tutti i campi del piano di ricerca, e quindi correggerne l'ambito prima della chiamata allo strumento di ricerca MCP.

5.7 Osservabilità e contesto compatto

Il *checkpointer* PostgreSQL persiste ogni esecuzione come sequenza di transizioni di stato. Per ogni `step_id`, il campo `current_step_results` conserva l'esito booleano della chiamata e i dati prodotti, oppure il messaggio d'errore. Per una singola richiesta possiamo quindi ricostruire l'esito della *Semantic Cache*, la motivazione del piano, la composizione del *fan-out* parallelo e il numero di riformulazioni consumate. Usiamo la stessa traccia per il *debugging* durante lo sviluppo e per l'audit del comportamento in esercizio.

I modelli a contesto lungo tendono a trascurare le informazioni collocate al centro di un *input* esteso, un problema che la letteratura chiama *Lost in the Middle* [29]. I sistemi RAG passano al modello solo i passaggi recuperati per la domanda, invece dell'intera base di conoscenza [26].

La v2 applica lo stesso principio al contesto conversazionale. Per ogni *step* il *Planner* scrive in `inputs` un *digest*, cioè le sole istruzioni e informazioni che servono all'agente, e indica in `relevant_chat_history` i turni della conversazione pertinenti. L'agente riceve il *digest* al posto della cronologia integrale, salvo quando il piano imposta `pass_full_history`. La riduzione dei *token* passati agli agenti non è stata misurata durante lo sviluppo della v2, e la scelta si basa solo sulla letteratura.

5.8 *Checkpointing* e *human-in-the-loop*

La *Deep Research* resta sospesa in attesa dell'utente per un tempo che il sistema non controlla. Nella v2 le due fermate su `interrupt()` (Sezione 5.6.1) sono gli unici punti in cui il grafo restituisce il controllo all'utente. Nell'orchestrazione principale il *Route Executor* instrada gli *step* validati senza approvazioni intermedie.

Il *checkpointer* serializza su PostgreSQL lo stato completo del *workflow*: il piano, i risultati parziali, il contatore dei tentativi e gli errori. Il grafo non impone limiti temporali alla ripresa, e lo stato resta su disco finché il *thread* non viene eliminato. Un *workflow* sospeso può quindi riprendere anche dopo una risposta umana differita.

5.9 Sintesi: *trade-off* e limiti dell'architettura

Un piano tipizzato e un grafo di stato aumentano la complessità implementativa rispetto a un *workflow* lasciato al modello. La separazione fra *Planner* ed *Executor* migliora la verificabilità e aggiunge lavoro a ogni richiesta: una chiamata al modello per il piano, la sua validazione e, in caso di errore, le riformulazioni del *Feedback Loop*. La Tabella 5.6 confronta questi costi con il prototipo v1.

Tabella 5.6: *Trade-off* quantitativi tra il prototipo v1 (n8n) e l'architettura v2 (LangGraph), sulle grandezze rilevate su entrambi i sistemi.

Dimensione	**v1 (n8n)**	**v2 (LangGraph)**	**Δ**
Nodi del flusso di orchestrazione[†]	33	10	−23
Nodi che invocano un modello[†]	5	2	−3
Righe di codice imperativo[§]	501	1.366	+865
Unità di manutenzione[§]	7 nodi *Code*	14 moduli	+7
Allucinazione di strumento[‡]	11,1 %	0,7 %	−10,4 p.p.

[†] Per la v1, conteggio sull'esportazione JSON del *workflow* n8n su cui è raccolto il *benchmark* di tirocinio. Il file si chiama *Agentic Platform v3* per la numerazione interna del tirocinio, diversa da quella di questa tesi. Per la v2, nodi registrati sul grafo LangGraph. I cinque nodi inferenziali della v1 sono i modelli collegati ai quattro agenti (*Planner*, *DB*, *Web Search*, *Synthesizer*) e allo strumento di ricerca vettoriale; i due della v2 sono il *Planner* e il *Synthesizer*. Dentro gli agenti della v2 restano chiamate inferenziali che non decidono l'instradamento (Sezione 5.1).

[§] Per la v1, righe non vuote e non di commento nei sette nodi *Code* JavaScript del *workflow*, dove risiede tutta la logica imperativa (fra questi, *Executor Logic* da solo ne conta 142). Per la v2, righe non vuote e non di commento del livello di orchestrazione alla revisione `b1620f9` (26 giugno 2026), esclusi agenti, *wrapper* di strumenti, *test* e *UI*; includendo gli agenti il *core* sale a 2.404 righe su 20 moduli.

[‡] Errore di strumento, cioè strumento invocato diverso da quello atteso, misurato sul *Planner* n8n nella campagna di tirocinio e sull'orchestratore v2 nel *routing benchmark*. La riduzione relativa è del 93,7 %; i due campioni non sono appaiati (vedi testo).

La tabella non riporta la latenza: sul prototipo v1 non l'abbiamo mai rilevata, e non dichiariamo quindi alcun guadagno di latenza. Sulla sola v2 il *benchmark* misura 2,2 secondi sul *workflow* a singolo agente e 50,4 secondi su quello con ricerca *web* esterna.[1]

Il livello di orchestrazione della v2 conta 1.366 righe su 14 moduli, 2.404 su 20 includendo gli agenti. La pianificazione aggiunge una chiamata al modello prima di ogni esecuzione, evitata solo da un *hit* della *Semantic Cache*. Il *benchmark* non separa, nei 2,2 secondi del percorso più semplice, il tempo dell'orchestrazione in codice da quello delle due chiamate al modello.

Il beneficio misurato riguarda l'allucinazione di strumento, che passa dall'11,1% del *Planner* n8n allo 0,7% dell'orchestratore v2, una riduzione relativa del 93,7%. L'11,1% proviene dal *benchmark end-to-end* di tirocinio, separato sia dalle 50 *query* di test del prototipo sia dalle 100 del *routing benchmark* (Capitolo 7, Sezione 7.1.3). I due campioni non sono appaiati, e il confronto ha quindi valore descrittivo.

La complessità aggiuntiva si giustifica quando valgono le condizioni discusse nel Capitolo 8: quando l'esito di una decisione errata è costoso e l'esecuzione deve restare auditabile. Su un prototipo esplorativo lo stesso impianto è sovradimensionato.

Tre limiti restano aperti. Il contesto compatto dipende dalla qualità del *digest*: se il *Planner* omette un vincolo della richiesta, l'agente non lo riceve. Il controllo umano aggiunge due fermate a ogni *Deep Research*, e la ricerca non parte senza la risposta dell'utente. La *Semantic Cache*, infine, non gestisce la scadenza dei dati. Su prezzi e quotazioni una risposta memorizzata perde validità con il tempo, e la soglia di similarità non lo rileva. Servirebbe una politica di invalidazione temporale, che la v2 non definisce.

L'architettura v2 affida al modello due soli nodi del percorso di orchestrazione, il *Planner* e il *Synthesizer*. Instradamento, ordine di esecuzione e gestione dei tentativi sono codice Python; dentro gli agenti restano chiamate inferenziali che non scelgono l'agente da invocare. Accettiamo più codice da mantenere e otteniamo un piano ispezionabile prima dell'esecuzione. La tipizzazione copre però solo la forma del piano, e lascia scoperti gli errori di pianificazione semantica. Il Capitolo 6 descrive l'evoluzione verso OpenCode e la *Direct Corpus Interaction*, e il Capitolo 7 confronta v2 e v3 sul *routing benchmark*.

[1] *Benchmark* della sola v2, raccolto il 24 febbraio 2026 con `pytest-benchmark` in modalità *pedantic*. Modello Qwen3-Coder servito da vLLM a temperatura nulla, nella configurazione in uso in quella fase del progetto, senza contesa sull'*endpoint* di inferenza. *Workflow* a singolo agente: cinque ripetizioni, media 2,22 s, deviazione *standard* 0,12 s. *Workflow* con ricerca *web*: tre ripetizioni, media 50,40 s, deviazione *standard* 12,02 s. La dispersione del secondo valore dipende dai tempi di risposta del *server* MCP esterno, assente nel primo *workflow*. Le latenze *end-to-end* della campagna comparativa, riportate nel Capitolo 7, misurano una grandezza diversa e non sono confrontabili con questi valori.

6. Architettura del sistema: *framework* dichiarativo e *Direct Corpus Interaction* (OpenCode, Agentic RAG)

Questo capitolo descrive la seconda riscrittura architetturale di *Agentic Platform*. La v2 opera su LangGraph, con un grafo deterministico scritto in Python (Capitolo 5). La portiamo su OpenCode, un *framework* dichiarativo (v3, Fase 3). Nell'ultima fase estendiamo il modello di permessi al recupero stesso delle informazioni, con la *Direct Corpus Interaction* (DCI, Agentic RAG, Fase 4).

L'esposizione segue l'ordine cronologico delle due fasi. Prima le ragioni della migrazione e l'architettura che ne risulta, poi il modello di permessi, i *custom tool* e i flussi operativi. Chiudiamo con la DCI e con le configurazioni di recupero che il Capitolo 7 valuta sperimentalmente.

6.1 Motivazioni della migrazione a OpenCode (v3)

La v2 riduce le allucinazioni di strumento dall'11,1% del prototipo n8n allo 0,7%, su campioni distinti e non appaiati (Sezione 7.1.3). La riscrittura risponde a un problema diverso: lo sforzo di manutenzione del grafo.

Il Capitolo 5 misura 1.366 LOC di sola orchestrazione, che salgono a 2.404 righe Python su 20 moduli includendo le implementazioni degli agenti. Aggiungere un agente richiede di modificare il grafo. Per ciascuno dei tre agenti aggiunti nel corso della v2 stimiamo fra le otto e le dodici ore di lavoro.

Il grafo accoppia inoltre logica e configurazione. Per cambiare il modello LLM di un agente dovevamo modificare il grafo e rigenerare i *checkpoint* di test. In forma dichiarativa basta cambiare il campo `model` nel file dell'agente (Tabella 6.1).

Migriamo perciò il sistema su OpenCode [36][1]. Il nuovo *runtime* conserva i permessi granulari e li esprime in file di configurazione, al posto del codice imperativo. La separazione fra ragionamento ed esecuzione si conserva invece solo in parte (Sezione 6.3.2).

[1] `https://opencode.ai`, *runtime* agentico TypeScript/Node.js a licenza MIT, con paradigma dichiarativo per agenti, *tool* e permessi. Adottiamo la versione 0.4.x di marzo 2026 (*commit* `abf9e21`).

Prima di adottare il nuovo *runtime* fissiamo le proprietà che la migrazione non deve degradare. Il *data layer* resta invariato. Il controllo umano resta, spostato dal grafo al *tool* (Sezione 6.4). La qualità della delega si misura sullo stesso *benchmark* di *routing* usato per la v2 (Sezione 7.1.2).

Tabella 6.1: Indicatori di sforzo di manutenzione, v2 (LangGraph) *vs* v3 (OpenCode), sulle operazioni più frequenti durante l'evoluzione del progetto.

Operazione	**v2**	**v3**
Aggiunta di un agente: file modificati	4	1–2
Aggiunta di un agente: ore-uomo	8–12	0,5–1,5
Cambio del modello LLM di un agente: *diff* (LOC)	15–40	1–3

Stime a posteriori dello sviluppatore. Gli intervalli della v2 coprono i tre agenti introdotti in quella fase. I valori della v3 sono proiezioni, perché nessun agente è stato aggiunto dopo il *commit* iniziale.

I valori della Tabella 6.1 sono stime a posteriori dello sviluppatore. La storia `git` fissa il periodo di riferimento. La v2 conta 12 *commit*, dal 26 febbraio al 26 giugno 2026. La v3 ne conta 30, dal 6 maggio al 17 luglio 2026. I *commit* registrano date e file modificati, ma nessun tempo di lavoro.

Nella v3 i cinque agenti compaiono già nel *commit* iniziale. La stima per l'aggiunta di un agente è quindi una proiezione, priva di un caso osservato. Anche il cambio di modello per un singolo agente è rimasto teorico, perché tutti gli agenti condividono lo stesso modello (Sezione 6.2.2). La rilevazione copre un solo progetto e un solo sviluppatore, e indica al più l'ordine di grandezza della differenza (Sezione 8.3.1).

6.2 Architettura del sistema OpenCode

Nella v2 il comportamento del sistema era scritto nel codice: il grafo Python fissava l'ordine delle invocazioni fra i nodi. In OpenCode dichiariamo gli agenti e i permessi che li vincolano. L'ordine delle deleghe lo sceglie l'Orchestrator, cioè il modello, mentre il *runtime* le esegue e applica i controlli di permesso. Le operazioni della Tabella 6.1 si riducono così alla modifica di uno o due file dichiarativi. Nella v2 richiedevano di riscrivere e ritestare una porzione di grafo.

Le righe di codice mostrano dove si sposta la complessità[2]. Contiamo le righe non vuote e non di commento. I cinque *custom tool* TypeScript ne occupano 692. Il perimetro dichiarativo, cioè i cinque file Markdown degli agenti più `opencode.json`, ne conta 453. Il *Bridge* FastAPI aggiunge 1.204 righe Python (Sezione 6.2.1), un componente assente nella v2. L'orchestrazione esplicita non compare, perché la esegue il *runtime*.

[2]Misure alla revisione `9b50480` del 17 luglio 2026 del *repository* della v3, con lo stesso criterio applicato alla v2 in Tabella 5.6.

In totale la v3 conta 2.349 righe, contro i 2.404 LOC del *core* agentico v2. Le due architetture distribuiscono diversamente le responsabilità, quindi il confronto resta indicativo. Il volume del codice da mantenere è simile: la migrazione lo sposta dall'orchestrazione all'interfaccia e ai *tool*.

Il componente centrale dell'infrastruttura è il *server* OpenCode, il *runtime* di orchestrazione. Attorno a esso operano quattro componenti: il *Bridge* FastAPI, l'interfaccia Open WebUI, il *data layer* PostgreSQL con pgvector e lo *stack* di inferenza vLLM.

I sei file dichiarativi fissano agenti e permessi: i *tool* consentiti, il modello, il numero massimo di passi. Chi li ispeziona conosce i permessi di ogni agente senza leggere il codice del *runtime*. I controlli sugli argomenti delle chiamate restano invece nel codice TypeScript dei *tool*: la validazione SQL e le precondizioni della *Deep Research* (Tabella 6.4).

6.2.1 Il *server* OpenCode e il *Bridge*

Una richiesta attraversa due componenti in successione. Il primo è il *Bridge*, scritto in Python con FastAPI [46], che espone la piattaforma a qualunque *client* OpenAI-compatibile (Tabella 6.2). Il *Bridge* gestisce anche lo stato conversazionale. Mantiene una *cache* di sessione con TTL di 600 s e isola i *thread* tramite l'*header* `x-thread-id`, poi inietta la cronologia nelle richieste in uscita. A protezione del servizio limita le chiamate a 30 al minuto per IP, 10 per l'*ingestion*, e deduplica i documenti RAG per *hash* SHA-256.

Il secondo componente è il *server* OpenCode, in ascolto sulla porta 4096. Mantiene le sessioni conversazionali e ne salva stato e cronologia su disco. Quando l'Orchestrator delega tramite `task`, il *server* avvia la sessione del sotto-agente. Invoca poi i *tool* TypeScript registrati e verifica su ognuno i permessi dell'agente chiamante.

I permessi sono distribuiti su due livelli. Il blocco `permission` globale di `opencode.json` concede tutti i *tool*, compresi `bash`, `edit` e `write`, mentre il blocco `provider` dello stesso file configura il modello. La restrizione effettiva sta nel file di ciascun agente, con la regola `*: deny` (Sezione 6.3.1). Il perimetro di un agente dipende quindi da quella sola dichiarazione: un agente definito senza la regola riceverebbe i permessi globali.

6.2.2 Open WebUI e *data layer*

L'interfaccia utente è Open WebUI [35], collegata al *Bridge* come un *backend* OpenAI. Da lì l'utente seleziona l'agente e conduce conversazioni multi-turno.

Tabella 6.2: *Endpoint* esposti dal *Bridge* FastAPI ai *client* OpenAI-compatibili, con la funzione di ciascuno.

Endpoint	**Funzione**
`POST /v1/chat/completions`	*Proxy* delle richieste *chat* verso OpenCode (*streaming* SSE e non)
`GET /v1/models`	Espone i 5 agenti come "modelli" selezionabili dal *client*
`POST /v1/embeddings`	*Proxy* verso il *server* vLLM di *embedding* (`qwen3-embedding-0-6b`, 1.024-d)
`POST /v1/ingest`	*Pipeline* RAG: *upload* → estrazione → *chunking* → *embedding* → pgvector
`GET /v1/collections`	Lista delle collezioni RAG con conteggio documenti
`DELETE /v1/collections/{collection}`	Elimina una collezione RAG
`DELETE /v1/collections/{collection}/source/{source}`	Rimuove da una collezione un documento sorgente
`GET /health`, `GET /ping`	Controllo di stato del servizio

Il *data layer* resta quello della v2: PostgreSQL [41] con pgvector, eseguito con Docker Compose. Lo stesso database svolge due ruoli. È il *vector store* del *RAG Agent*: pgvector indicizza i *chunk* della *knowledge base* e li confronta per distanza coseno. Conserva inoltre i dati strutturati che il *DB Agent* interroga in SQL di sola lettura. Sono il catalogo *hardware* IT aziendale, nella tabella `prodotti`, e le tabelle epidemiologiche estratte da Wikipedia su cui lavora il *benchmark* di *routing*.

L'inferenza usa lo stesso *stack* di *serving* della v2: vLLM [23] su GPU NVIDIA H100, nel *cluster* interno. Il modello cambia invece rispetto alla prima fase della v2. La v3 adotta Qwen3.6-27B-AWQ-INT4 (`qwen36-27b-awq-int4`), servito *on-premises*, al posto del `Qwen3-Coder-30B-A3B-Instruct` con cui la v2 era stata sviluppata.

La sostituzione avviene sullo *stack* aziendale prima della campagna comparativa e riguarda entrambe le architetture. Nella misura del *routing* (Sezione 7.1.2) la v2 e la v3 ragionano quindi sullo stesso modello. Il confronto resta longitudinale, perché le due fasi sono successive nel tempo. Il modello esce dalle differenze fra i due bracci, ma restano legati il *runtime* e i *prompt* di sistema, cambiati insieme. La Sezione 8.3 ne discute la portata.

OpenCode consentirebbe di assegnare a ogni agente un modello diverso, tramite il campo `model` del file dell'agente. Adottiamo un solo modello condiviso da tutti gli agenti e rinviamo l'assegnazione differenziata a un'ottimizzazione futura.

6.3 Agenti specializzati e modello di permessi (OpenCode)

Il sistema ha cinque agenti, ciascuno definito in un file Markdown separato, con ruolo e permessi propri. La Tabella 6.3 ne riporta la matrice, con la modalità operativa e i *tool* consentiti a ognuno.

Tabella 6.3: Matrice degli agenti specializzati nella versione OpenCode, con modalità operativa e permessi sui *tool*. Tutti e cinque gli agenti condividono lo stesso modello, Qwen3.6-27B-AWQ-INT4.

Agente	**Ruolo**	**Modalità**	***Tool* consentiti**
Orchestrator	Coordinamento e *routing*	Primario	`task`
Web Agent	Ricerche *web*	Sotto-agente	`web_search`, `webfetch`, `read`
Deep Research Agent	*Report* multi-fase	Sotto-agente	`deep_research`, `question`
DB Agent	SQL *read-only* e ricerca vettoriale sul catalogo	Sotto-agente	`db_query`, `product_search_vector`, `read`
RAG Agent	Ricerca ibrida sulla *knowledge base*	Sotto-agente	`rag_query`, `read`

Ogni agente dichiara `*: deny` come regola predefinita, quindi i *tool* assenti dalla colonna "consentiti" sono negati per costruzione.

Un agente è un file Markdown con un *front-matter* YAML, seguito dal *system prompt*. Il *front-matter* dichiara il modello, la temperatura, il numero massimo di passi e i permessi. Consideriamo l'Orchestrator: dichiara `mode: primary`, temperatura 0,1 e al più 20 passi, e riammette la sola delega via `task`. I cinque *custom tool* restano fuori dalla sua portata, insieme a ogni azione non concessa esplicitamente. Il suo *prompt* contiene la regola "Tu NON chiami direttamente i tool. Deleghi sempre via `task` al sotto-agente", che il *runtime* impone comunque, a prescindere dall'aderenza del modello.

Sotto l'Orchestrator operano quattro sotto-agenti, ciascuno limitato ai *tool* del proprio dominio. Tre dei quattro dispongono anche di `read`. Il *Deep Research Agent* ne fa a meno: usa `deep_research` per la ricerca e `question` per interrogare l'utente. Nessun agente ha `bash`, `edit` o `write` nella propria *allow-list*.

Il *Web Agent* cerca le informazioni esterne ai sistemi aziendali e dispone di due *tool* oltre a `read`. `web_search` interroga un *server* MCP interno basato su Exa AI [10], con una sessione che il *tool* gestisce per conto dell'agente (Tabella 6.4). `webfetch` è invece un *tool* nativo di OpenCode, che scarica il contenuto di un URL senza passare da MCP. Il *prompt* dell'agente limita a due le chiamate a `web_search` per ogni richiesta. I risultati tornano all'Orchestrator, che li compone nella risposta.

Il *DB Agent* interroga i dati strutturati in sola lettura, con due *tool* complementari fra cui il modello sceglie in base alla richiesta. `db_query` esegue *query* SQL, necessarie per i filtri esatti

su prezzo, categoria o disponibilità e per le aggregazioni. Il codice del *tool* applica un controllo lessicale a valle del modello (Tabella 6.4). Il controllo respinge un comando di modifica esplicito, anche quando lo induce un *prompt* malevolo.

Il filtro ha però un limite preciso. Una CTE introdotta da `WITH` può contenere `DELETE` o `UPDATE`, e il controllo lessicale la lascia passare. La garanzia di sola lettura dipende quindi anche dai privilegi del ruolo PostgreSQL con cui il *tool* si connette.

Con `product_search_vector` l'agente affianca alla via SQL una ricerca vettoriale sulla tabella `prodotti`. Serve per le richieste vaghe o orientate al caso d'uso, come "una workstation potente per l'AI", dove il confronto lessicale di SQL non trova corrispondenze. Il *tool* restituisce per default i 5 prodotti più simili sopra una soglia di similarità coseno di 0,5.

Il *RAG Agent* recupera la documentazione interna, cioè i contenuti testuali della *knowledge base*, con il solo *tool* `rag_query`. Il recupero combina due rami. Il ramo denso confronta gli *embedding* con pgvector. Il ramo lessicale usa la ricerca *full-text* di PostgreSQL con punteggio `ts_rank`. Il punteggio pesa la frequenza dei termini ma, a differenza di BM25 [47], trascura la loro rarità nel *corpus*. Il *tool* fonde i due *ranking* con RRF e riordina i candidati con `qwen3-reranker-0-6b` (Tabella 6.4).

Il ramo lessicale conserva le corrispondenze esatte su sigle e termini rari, che la sola similarità densa può perdere. Il *tool* va distinto dalla ricerca vettoriale del *DB Agent*: quella opera sul catalogo prodotti, questo sulla *knowledge base* documentale. L'*embedder* `qwen3-embedding-0-6b` è lo stesso che la v2 usa nella campagna comparativa, mentre BAAI/bge-m3 appartiene alla fase di sviluppo precedente. Il ramo denso si fonda quindi sugli stessi vettori nelle due architetture.

Il *Deep Research Agent* produce un *report* multi-fase ed è il solo agente autorizzato a invocare `deep_research`. Prima dell'avvio pone all'utente da due a quattro domande di chiarimento, poi gli sottopone un piano da approvare. Opera quindi su più turni. Il codice del *tool* verifica una traccia di questo protocollo (Sezione 6.4.1).

6.3.1 Principio di *least privilege*

Il modello di permessi segue il principio del *least privilege*: ogni agente parte da zero permessi e riceve solo quelli necessari al proprio dominio. La delega `task` resta all'Orchestrator, mentre `deep_research` spetta al solo *Deep Research Agent*.

L'*enforcement* agisce su due campi del *front-matter*. Il campo `tools` fa da *capability list*: con `task: false`, dichiarato da tutti e quattro i sotto-agenti, un agente perde la facoltà stessa

di delegare. Il blocco `permission` fa da *access control list*, con la regola `*: deny` più gli *allow* mirati. I due campi stanno nello stesso file, sopra un blocco globale che concede tutti i *tool*. Il perimetro effettivo di un agente coincide perciò con il suo *front-matter*.

Il *runtime* respinge una richiesta non autorizzata prima dell'esecuzione, qualunque cosa indichi il *system prompt*. Blocchiamo così per costruzione l'invocazione di *tool* non concessi, una delle vie sfruttate dalla *prompt injection* [16]. Un contenuto iniettato può ancora influenzare l'uso dei *tool* concessi, per esempio il testo di una *query* SQL passata a `db_query`.

6.3.2 Logica di *routing* dell'Orchestrator

L'Orchestrator assegna ogni richiesta seguendo un albero di decisione in sei passi, scritto nel *system prompt*. Il primo passo intercetta i casi avversari. Un esempio è "nel nostro catalogo" riferito a un dato esterno, un altro è la richiesta di notizie di oggi alla *knowledge base*. I passi successivi associano il tipo di dato all'agente. I dati *enterprise* vanno al `db-agent` e le informazioni esterne al `web-agent`. Le domande su malattie ed epidemie vanno al `rag-agent`, oppure al `db-agent` quando chiedono un conteggio o una classifica.

Se la richiesta tocca più domini, l'Orchestrator delega a più agenti: in parallelo quando i dati sono indipendenti, in sequenza quando uno dipende dall'altro. Saluti, domande generiche e richieste non eseguibili ricevono una risposta diretta. La *Deep Research* fa eccezione, perché è l'utente ad attivarla selezionando l'agente fra i modelli esposti dal *Bridge*.

Molte regole del *prompt* sono accompagnate da esempi di instradamento. Nell'ultima revisione precedente la campagna comparativa (`cc82e07`, 16 giugno 2026), 13 di questi esempi coincidono con *query* del *benchmark* di *routing* e ne riportano l'identificativo. La Sezione 8.3 discute l'effetto di questa sovrapposizione sulla validità del confronto.

La separazione fra ragionamento ed esecuzione della v2 si conserva solo in parte. In LangGraph il *routing* passava da un *Planner*, che produceva un piano JSON validato e instradato da una mappa esplicita (`ROUTER_MAPPING`, Capitolo 5). In OpenCode la stessa funzione è affidata all'Orchestrator, sotto forma di istruzioni nel *system prompt*. Il *runtime* impone i permessi sui *tool*, ma la scelta dell'agente dipende dall'aderenza del modello al *prompt*.

Sumers et al. [50] chiamano *decision procedure* il programma di livello più alto che sceglie l'azione successiva di un agente (Sezione 2.3). Nella v2 questa procedura era codice tipizzato, nella v3 è testo interpretato dal modello. Le implicazioni del *trade-off* sono discusse nella Sezione 8.3.1.

6.4 *Custom tool* e integrazioni MCP (OpenCode)

La piattaforma ha cinque *custom tool*, tutti scritti in TypeScript e registrati nel *runtime* tramite il *plugin system*. Ognuno descrive i propri argomenti con uno *schema* Zod, usato per validare l'input. `web_search` e `deep_research` comunicano con due *server* MCP [1] interni, gli altri tre interrogano PostgreSQL. La Tabella 6.4 li elenca con la funzione e il meccanismo di controllo di ciascuno.

Tabella 6.4: *Custom tool* TypeScript del *runtime* OpenCode: funzione e meccanismo di sicurezza o di *enforcement*.

Tool	**Funzione**	**Sicurezza / *enforcement***
`web_search.ts`	Ricerca *web* via MCP JSON-RPC (`query`, `complexity`, `user_country`)	Gestione autonoma della sessione MCP, re-inizializzata su HTTP 401/403; *timeout* di 30 s per richiesta e di 90 s per l'intera chiamata
`deep_research.ts`	Ricerca approfondita multi-fase con HITL	Valida 3 precondizioni prima dell'invocazione MCP: (A.1) almeno 2 domande numerate, testo complessivo di almeno 50 caratteri; (A.2) piano Markdown di almeno 200 caratteri; (A.3) messaggio di approvazione di al più 60 caratteri, con una parola chiave di conferma. Ogni violazione restituisce un codice stabile (`PROTOCOL VIOLATION [A.1/A.2/A.3]`) che il modello usa per correggersi; *timeout* di 120 s
`product_search_vector.ts`	Ricerca vettoriale sul catalogo `prodotti` per richieste vaghe o *intent-based*: *embedding* `qwen3-embedding-0-6b` + pgvector [40] (distanza coseno `<=>`)	Soglia di similarità predefinita 0,5 e 5 risultati, modificabili dall'agente; *query* SQL parametrizzata
`rag_query.ts`	Ricerca ibrida sulla *knowledge base*: ramo denso pgvector (coseno, soglia 0,3) e ramo lessicale *full-text* (`ts_rank`), 20 candidati per ramo, fusi con *Reciprocal Rank Fusion* ($k = 60$) e riordinati con `qwen3-reranker-0-6b`	Unico *tool* di recupero del *RAG Agent*; *query* parametrizzate di sola lettura sull'indice, perché l'*ingestion* resta al *Bridge*; se il *reranker* non risponde, restituisce l'ordine RRF
`db_query.ts`	SQL sul database *enterprise*	Controllo lessicale: ammette solo *query* che iniziano con `SELECT` o `WITH`, respinge il carattere `;` e aggiunge `LIMIT 100` se assente. Le CTE di modifica (`WITH ... DELETE`) superano il controllo

6.4.1 Il protocollo *Human-in-the-Loop* del *tool* `deep_research`

Nella v2 il controllo umano passava da `interrupt()`, che sospende il grafo di stato fino alla risposta dell'utente (Capitolo 5, Sezione 5.6.1). In OpenCode il grafo non esiste, quindi collochiamo la garanzia nel codice del *tool*. `deep_research.ts` verifica le tre precondizioni della Tabella 6.4 prima di qualunque chiamata MCP, a prescindere dalle istruzioni del *system prompt*.

Quanto è forte questa garanzia? Le precondizioni controllano il contenuto di tre argomenti: le domande poste, il piano mostrato e il messaggio di approvazione. Questi argomenti però li compila il modello stesso. Il *tool* non può accertare che `user_approval` riporti davvero un turno dell'utente, quindi un "ok" generato dal modello supera il controllo A.3. La ricerca per parola chiave accetta inoltre risposte negative che contengono una formula di assenso, come "no, non va bene".

Il controllo intercetta quindi le chiamate premature, in cui il modello salta le domande o il piano. Resta però più debole della sospensione del grafo nella v2, dove l'esecuzione riprende solo con un nuovo messaggio dell'utente.

6.5 Flussi operativi (OpenCode)

I tre flussi operativi partono dallo stesso ingresso, *Bridge* → *server* OpenCode. Si distinguono per l'agente selezionato dall'utente e per i *tool* che attivano.

Nel flusso *Standard* l'utente seleziona l'Orchestrator. Il modello interpreta l'intento della richiesta e delega via `task` ai sotto-agenti competenti, secondo l'albero di decisione della Sezione 6.3.2. Le deleghe indipendenti partono nella stessa risposta e il *runtime* le esegue in parallelo. Per la *chat* generica o le richieste fuori perimetro l'Orchestrator risponde senza delegare, e agisce così da *guardrail*.

Rispetto a LangGraph cambia la ripartizione dei ruoli: l'Orchestrator svolge in un solo agente le funzioni di *Planner* e di *Synthesizer*. Il Capitolo 7 misura l'effetto su accuratezza e latenza *end-to-end* (Tabella 7.2), senza isolare il contributo della sola orchestrazione.

Nel flusso *Deep Research* l'utente seleziona direttamente il *Deep Research Agent* fra i modelli esposti dal *Bridge*. Il *prompt* dell'Orchestrator vieta la delega a questo agente. Il *tool* `deep_research` applica le precondizioni della Sezione 6.4.1 prima di inoltrare la richiesta al *server* MCP. Il *report* torna all'utente senza rielaborazione.

Il flusso *RAG* si svolge in due tempi. Il primo è l'*ingestion*, gestita dal *Bridge*. Il servizio estrae il testo, lo segmenta e converte i *chunk* in *embedding* con `qwen3-embedding-0-6b`. Li inserisce poi in pgvector, dopo aver scartato i duplicati per *hash* SHA-256.

La segmentazione usa il *semantic chunking* di Chonkie [4], con una dimensione massima di 500 *token*. I confini dei *chunk* cadono dove cala la similarità fra frasi vicine. L'*endpoint* accetta anche un parametro di sovrapposizione, che il *chunker* semantico non applica e che resta nella firma per compatibilità con le chiamate esistenti. Questi parametri di produzione differiscono da quelli delle *pipeline* sperimentali della Sezione 6.7.

Il secondo tempo è la *query*. L'Orchestrator delega al *RAG Agent*, che invoca `rag_query` per la ricerca ibrida descritta nella Sezione 6.3.

In tutti e tre i flussi l'autorizzazione effettiva resta quella della Tabella 6.3: nessun agente accede a un *tool* assente dalla propria *allow-list*. Gli *endpoint* `DELETE` del *Bridge* restano fuori da questa matrice, perché li invoca direttamente il *client*.

6.6 *Direct Corpus Interaction* e *pipeline* di *benchmark* di recupero (Agentic RAG)

Un sistema RAG tradizionale segmenta i documenti in frammenti e ne indicizza i vettori in un *vector store*. A ogni domanda recupera i frammenti più vicini per similarità. Il meccanismo richiede manutenzione: la strategia di segmentazione, l'aggiornamento dell'indice, la latenza di rete verso il database vettoriale. Il *chunking* impone inoltre un limite, perché l'agente legge solo i frammenti prodotti dalla segmentazione. Ci chiediamo allora se l'indice sia davvero necessario.

Nell'ultima fase del progetto valutiamo un recupero senza indice, la *Direct Corpus Interaction* (DCI). Concediamo all'agente la lettura diretta dei file del *corpus*, con gli strumenti testuali che un operatore umano userebbe su una cartella di documenti.

La DCI estende il modello di permessi della v3. La stessa matrice che nega a un agente `bash`, `edit` e `write` gli concede qui un perimetro di sola lettura, circoscritto al *corpus*. Il recupero diventa così un'azione agentica osservabile: ogni `grep` e ogni `read` restano tracciati nella sessione.

Chiamiamo *Agentic File Search* l'agente che opera in questo modo, per distinguerlo dal *RAG Agent*, che interroga un *vector store* via `rag_query`. Dispone di tre strumenti nativi di OpenCode: `glob` percorre la struttura del *corpus*, `grep` cerca le occorrenze esatte di un termine, `read` apre i

passaggi rilevanti. Le operazioni si compongono in sequenza. Un primo `grep` filtra le righe su un vincolo, un secondo le restringe a quelle che soddisfano anche il vincolo successivo. Una `read` mirata recupera infine il contesto attorno alla corrispondenza.

Il paradigma proviene da Li et al. [28], che lo formalizzano come modalità di recupero. Gli autori lo inquadrano nella *retrieval interface resolution* (Sezione 2.5.2), cioè la granularità dell'interfaccia con cui il modello accede al *corpus*. Un *retriever* convenzionale comprime quell'accesso in un unico passo *top-k*, quindi offre un'interfaccia a bassa risoluzione. L'agente riceve solo gli *snippet* esposti dal *retriever*. Ciò che il filtro scarta a monte resta irrecuperabile, per quanto efficace sia il ragionamento a valle.

La DCI si colloca all'estremo opposto. Raggiunto un documento utile, l'agente vi cerca i termini specifici e ne apre il testo per intero. Le entità che trova diventano il termine di nuove ricerche.

Li et al. [28] collegano il vantaggio della DCI a questo uso iterativo dei documenti trovati, e separano due grandezze. La *copertura* misura quanti documenti *gold* la traiettoria raggiunge. La *localizzazione* misura quanto l'agente restringe il fuoco a un frammento di evidenza piccolo e utile dentro il documento. Nei loro esperimenti la DCI ha copertura media inferiore a un *retriever* denso e localizzazione superiore.

Le nostre metriche di recupero misurano il rango dei documenti, quindi osservano solo la prima grandezza. Il Capitolo 7 confronta su questa base la DCI con i due sistemi vettoriali (Tabella 7.4). L'MRR misura la posizione del primo documento pertinente, la Recall@10 la copertura dei primi dieci.

All'*Agentic File Search* concediamo `glob`, `grep` e `read` sulla cartella dei documenti di *benchmark*. La traiettoria dipende dal *prompt* e dal campionamento del modello, senza parametri di *similarity search*. Li et al. [28] osservano che il numero di ricerche necessarie a raggiungere un documento pertinente cresce con lo spazio dei candidati.

6.6.1 La politica di lettura GRASP

I permessi appena descritti lasciano aperto un problema di granularità. Il `read` predefinito restituisce il file per intero, anche quando la porzione utile occupa poche righe. Su un *corpus* di articoli lunghi come MultiHop-RAG questa lettura determina gran parte dei *token* di ingresso dell'agente (Tabella 7.8). Eppure, una volta che `grep` ha individuato la riga pertinente, il contesto utile si riduce alle righe vicine.

Per questo adottiamo un adattamento della *GRanularity-Aware Search Policy* (GRASP) di Gandhi et al. [12]. Nel lavoro originale GRASP è una *policy* addestrata per rinforzo, che coordina ricerca semantica, ricerca per parola chiave e lettura di paragrafi. Ne riproduciamo la sola lettura mirata, senza addestramento, con un'istruzione nel *prompt* dell'agente.

L'istruzione rende predefinita la consultazione mirata. L'agente localizza la riga con `grep`. La successiva `read` riceve `offset` pari alla riga della corrispondenza meno quindici e `limit` pari a trenta. Se il passaggio prosegue oltre la finestra, l'agente ripete la `read` con un altro `offset`, e conserva così la flessibilità multi-turno della DCI.

Il *runtime* non impone questi valori, e la loro applicazione dipende dall'aderenza del modello al *prompt*. L'adattamento modifica un solo artefatto già presente nell'architettura e non aggiunge componenti. Il Capitolo 7 ne misura l'effetto su costo, latenza e copertura, compresa la perdita di copertura attesa da una finestra più stretta (Sezione 7.2.6).

6.7 Le configurazioni di recupero del RAG avanzato

Nel Capitolo 7 (Sezione 7.2) confrontiamo tre sistemi su tre dataset: l'agente DCI, un RAG classico e un RAG avanzato. Descriviamo qui l'architettura delle due *pipeline* vettoriali, prima di misurarle. Sono *script standalone*, senza *Bridge* e senza orchestrazione multi-agente, costruiti per isolare le variabili del recupero.

Il RAG classico è la configurazione minima. Segmenta il *corpus* in frammenti da 512 *token* con 64 di sovrapposizione e ne calcola l'*embedding*. A ogni domanda recupera i primi dieci per similarità coseno, in un solo passaggio, senza *reranking* né fusione. Ci serve come *baseline*: il riferimento rispetto a cui misuriamo il guadagno delle tecniche più elaborate.

Il RAG avanzato è invece uno spazio di configurazione, composto dai quattro moduli della Sezione 2.6. Il primo è la strategia di *chunking*, a *token*, semantica o ricorsiva. Il secondo è la ricerca ibrida, che affianca BM25 [47] al recupero denso con fusione RRF [6]. Seguono l'espansione della *query* in stile RAG-Fusion [45] e il *reranking cross-encoder* [34].

Quale modulo migliora il recupero, e a quale costo? Per rispondere attiviamo o disattiviamo ciascun modulo e percorriamo la famiglia di varianti come un'*ablation*. Partiamo dal vettore puro e aggiungiamo un modulo alla volta: prima il solo ibrido, poi l'ibrido con espansione RAG-Fusion, poi l'ibrido con *reranking*. Le configurazioni più complete combinano espansione

e *reranking*, nelle varianti *early* e *late*, distinte dall'ordine in cui applicano riordino e fusione delle riformulazioni.

Queste *pipeline* usano lo *stack* sperimentale della Sezione 3.10, al posto del *backend* pgvector di produzione. Il *vector store* è Qdrant [42], che calcola la fusione RRF lato *server*. La segmentazione usa Chonkie [4], mentre `qwen3-embedding-0-6b` e `qwen3-reranker-0-6b` gestiscono recupero e riordino.

Ogni sistema risponde alle 100 *query* di ciascun dataset, e Ragas valuta le risposte in locale con una *pipeline* di misura asincrona. A valle di ogni esecuzione un controllo verifica l'integrità dei punteggi, secondo il protocollo della Sezione 4.5. La misura di questo spazio di configurazione, con i suoi risultati anche negativi, è nel Capitolo 7 (Sezione 7.2).

6.8 Sintesi

La migrazione a OpenCode (v3) esprime i permessi in forma dichiarativa e sposta il *routing* nel *system prompt* dell'Orchestrator. Il *runtime* blocca per costruzione i *tool* non concessi. La scelta dell'agente e il protocollo della *Deep Research* dipendono invece dall'aderenza del modello. Il codice da mantenere ha un volume simile a quello della v2 e si concentra nel *Bridge* e nei *tool*.

La Fase 4 estende il modello di permessi al recupero con la *Direct Corpus Interaction*, e ne riduce la lettura con un adattamento di GRASP. Il Capitolo 7 misura l'architettura in un confronto longitudinale con la v2 e confronta la DCI con le due *pipeline* vettoriali.

7. Risultati sperimentali: orchestrazione (LangGraph, OpenCode) e recupero (Agentic RAG)

Questo capitolo riporta le misure raccolte sull'architettura. La valutazione copre due fronti. Il primo è il *benchmark* di *routing*: confrontiamo l'orchestratore deterministico v2 (LangGraph) con quello dichiarativo v3 (OpenCode) su 100 *query* e misuriamo quale dei due delega il lavoro all'agente corretto. È la prova centrale, perché ogni passo successivo dipende dalla delega iniziale. La valutazione è programmatica e confronta l'esito con il *gold standard*.

Il secondo fronte è il recupero. Confrontiamo tre sistemi RAG su tre dataset: uno classico, uno avanzato e un agente su *file system* con *Direct Corpus Interaction*. Le metriche generative sono calcolate da Ragas con `minimax-m3` come giudice. L'ultima sezione confronta i risultati con la domanda di ricerca.

7.1 Valutazione dell'architettura OpenCode (v3)

Confrontiamo l'architettura OpenCode (Sezione 6.1) con la versione deterministica LangGraph (v2). I due bracci condividono l'infrastruttura e il *benchmark* di *routing*. In questo modo isoliamo l'effetto del cambio di orchestrazione: nessuna differenza di risultato può derivare da un *hardware* diverso.

7.1.1 Condizioni sperimentali

La Tabella 7.1 elenca le condizioni in cui abbiamo raccolto le misure, con la versione di ogni componente *software* e infrastrutturale.

Il *client* di *benchmark* è eseguito sulla *workstation* di sviluppo. Il *runtime* vLLM, il *data layer* e le GPU H100 sono invece quelli del *cluster* RKE2 già usato dalla v2. I due bracci condividono quindi l'infrastruttura di servizio e la configurazione di inferenza.

Le due campagne appartengono però a fasi successive del progetto, quindi il disegno è longitudinale. Per riportarle su un piano comune rimisuriamo la v2 dopo l'aggiornamento dello *stack* di *serving* aziendale. Entrambe le architetture ragionano allora con Qwen3.6-27B-

Tabella 7.1: Condizioni sperimentali del confronto di *routing* v2 *vs* v3. Modelli, GPU e *data layer* sono gli stessi nei due bracci: cambiano il *runtime* di orchestrazione e i *prompt* di sistema che richiede. La valutazione confronta l'esito con il *gold standard* in modo programmatico, senza giudice LLM.

Componente	Versione e configurazione
Modelli, identici nei due bracci	
Ragionamento	Qwen3.6-27B-AWQ-INT4, servito *on-premises*
Embedding	`qwen3-embedding-0-6b`, 1.024 dimensioni
Infrastruttura, identica nei due bracci	
GPU	1× NVIDIA H100 80 GB SXM (firmware 96.00.74)
Servizio di inferenza	vLLM 0.5.4, CUDA 12.4, driver NVIDIA 550.54
Database	PostgreSQL 16.3 con `pgvector` 0.7.0, indice HNSW (`m` = 16, `ef_construction` = 64)
Cluster	RKE2 1.28.x su *bare-metal*, *namespace* `davide-dev`
Orchestrazione e misura	
Runtime v3	OpenCode 0.4.x (commit `abf9e21`, marzo 2026), Node.js 20.11.1 LTS, TypeScript 5.4.5
Bridge	FastAPI 0.115.x su Python 3.11.9, `uvicorn` 0.30
Client di *benchmark*	`pytest-benchmark` 4.0 su Windows 11, Intel i5-11300H, 32 GB RAM

AWQ-INT4 e recuperano con `qwen3-embedding-0-6b`[1]. Qwen3-Coder-30B-A3B-Instruct e BAAI/bge-m3 appartengono solo alla fase di sviluppo della v2 e alle misure prestazionali raccolte allora (Capitolo 5).

Fra i due bracci cambiano due sole cose: il *runtime* di orchestrazione e i *prompt* di sistema che quel *runtime* richiede. I due elementi sono legati, perché nessuna delle due architetture accetta i *prompt* dell'altra. Non possiamo quindi separare il contributo del *runtime* da quello dei *prompt*. Lo dichiariamo qui e ne discutiamo la portata nella Sezione 8.3.

Il confronto ha un secondo limite, che riguarda il solo braccio v3. Nella revisione del *prompt* dell'Orchestrator usata per la campagna, 13 esempi di instradamento coincidono con *query* del *benchmark* e ne riportano l'identificativo (Sezione 6.3.2). Fra queste figurano 5 delle 11 *query* `ADV` e 2 delle 9 `C_SEQ`. Il *prompt* della v2 non contiene esempi tratti dal *benchmark*. La sovrapposizione può quindi favorire la v3 proprio sui *pattern* in cui guadagna di più.

[1] La configurazione della campagna è ricostruibile dal *repository* della v2. L'agente RAG compare per la prima volta nella revisione `a3785cb` del 16 giugno 2026. Lo stesso *commit* porta il modello di ragionamento a `qwen36-27b-awq-int4` e l'*embedder* a `qwen3-embedding-0-6b`. La campagna del giorno successivo delega all'agente RAG in 45 *query* su 100, quindi ha eseguito quella revisione o una successiva.

7.1.2 Benchmark di *routing* deterministico: delega LangGraph *vs* OpenCode

La prova di affidabilità più solida è il *benchmark* di *routing* deterministico (Sezione 4.4.1). Il banco contiene 100 *query* stratificate (`Q001`–`Q100`), distribuite per difficoltà 36/40/24 e comprensive di *query* avversarie e parafrasi. Le stesse *query* sono fornite in input all'orchestratore deterministico (v2, LangGraph) e a quello dichiarativo (v3, OpenCode). Per ciascuna misuriamo se il sistema attiva l'insieme corretto di agenti. Il confronto con il *gold standard* è programmatico e non coinvolge alcun giudice LLM, quindi la misura non è esposta al *self-evaluation bias*, che si manifesta quando giudice e generatore coincidono.

La Sezione 4.4.2 descrive la strumentazione, gli *adapter* che normalizzano i nomi degli agenti e le definizioni insiemistiche delle metriche. Stimiamo gli intervalli di confidenza al 95% via *bootstrap* percentile, con 5.000 ricampionamenti. La Tabella 7.2 riporta i risultati.

Tabella 7.2: Benchmark di *routing* deterministico su 100 *query* (`Q001`–`Q100`): v2 (LangGraph) *vs* v3 (OpenCode), valutazione programmatica contro il *gold standard*, IC 95% *bootstrap* percentile (5.000 ricampionamenti). *Run* del 19 giugno 2026.

Metrica	**v2 (LangGraph)**	**v3 (OpenCode)**	Δ
Routing accuracy	0,76 [0,68; 0,84]	**0,94** [0,89; 0,98]	+0,18
Agent F1	0,89 [0,84; 0,94]	**0,966** [0,93; 0,99]	+0,076
Agent precision	0,90	0,962	+0,062
Agent recall	0,90	0,975	+0,075
Sovra-delega (*over*)	0,10	0,06	−0,04
Sotto-delega (*under*)	0,10	0,03	−0,07
Delega errata (*wrong*)	0,03	0,02	−0,01
Delega multi-agente e avversaria ($n = 40$)[†]	0,825 (33/40)	0,925 (37/40)	+0,100
Routing sequenziale ($n = 9$)	0,56 (5/9)	0,889 (8/9)	+0,329
No-agent/*guardrail* ($n = 9$)	0,889 (8/9)	1,00 (9/9)	+0,111
Robustezza semantica, stretta ($n = 16$)[‡]	0,625 (10/16)	0,9375 (15/16)	+0,3125
Robustezza semantica, Jaccard ($n = 16$)[‡]	0,760	0,969	+0,209
Latenza p_{50}	118,0 s	**62,4** s	−55,6 s
Latenza p_{95}	**174,6** s	243,3 s	+68,7 s

[†] Aggregato dei tre *pattern* a delega parallela `D_DB_RAG`, `D_DB_WEB` e `D_RAG_WEB`, per 29 *query*, più le 11 *query* del *pattern* `ADV` (Tabella 4.3, Capitolo 4). Sui sottoinsiemi riportiamo fra parentesi il conteggio grezzo, perché su campioni così piccoli la percentuale sovrastima la precisione della misura.
[‡] Concordanza della delega fra formulazioni equivalenti, calcolata sui 16 gruppi di intento in cui si aggregano le 19 parafrasi e le rispettive formulazioni originali, per un totale di 35 *query* (Sezione 4.4.2). La forma stretta conta i gruppi instradati in modo identico; la forma Jaccard media la sovrapposizione fra le coppie di ciascun gruppo.

La *routing accuracy* sale da 0,76 a 0,94 (+0,18). Gli intervalli di confidenza, [0,68; 0,84] e [0,89; 0,98], sono disgiunti: su questa esecuzione il miglioramento eccede il rumore campionario. L'*Agent* F1 segue lo stesso andamento, da 0,89 a 0,966.

Il divario più ampio riguarda il *routing* sequenziale. Sui nove scenari a dipendenza sequenziale l'orchestratore deterministico ne risolve cinque, il dichiarativo otto. Sui *pattern* avversari e multi-fonte il divario è minore: sull'aggregato di 40 *query* la v3 ne risolve 37, la v2 33. Il dettaglio per *pattern* (`ADV`, `B_RAG`, `D_DB_RAG`) è nel pannello (b) della Figura 7.1. Calano anche la sovra-delega, da 0,10 a 0,06, e la sotto-delega, da 0,10 a 0,03.

Migliora anche la robustezza alla formulazione della domanda. Sui 16 gruppi di parafrasi la v2 instrada in modo identico 10 gruppi, la v3 15, e l'indice di Jaccard sale da 0,760 a 0,969. La delega della v3 dipende quindi meno dalla formulazione superficiale. La Figura 7.1 riporta il confronto su metriche aggregate, accuratezza per *pattern* e profilo radar.

A cosa attribuiamo il guadagno? Modello di ragionamento ed *embedder* sono gli stessi nei due bracci (Sezione 7.1.1), quindi la differenza non dipende dalla capacità del modello. Dipende dal *runtime* dichiarativo e dai suoi *prompt* di sistema, che non possiamo separare. Il dato è inoltre un'osservazione fra fasi successive, su una singola esecuzione, e 13 *query* del *benchmark* compaiono fra gli esempi del *prompt* v3 (Sezione 8.3).

La penultima riga della Tabella 7.2 riporta la latenza mediana per *query* sull'intero *benchmark*: scende da 118,0 s (v2) a 62,4 s (v3), un calo del 47%.

Il valore copre l'intera traiettoria *end-to-end* di ogni *query*: pianificazione, *fan-out* verso gli agenti specializzati, chiamate agli strumenti esterni, sintesi finale. Fra gli strumenti pesa soprattutto la ricerca *web* via Exa AI, con *timeout* a 30 s (Sezione 6.3). Il modello di ragionamento Qwen3.6-27B è servito *on-premises* da un solo *endpoint* H100. Gli agenti attivati in parallelo interrogano lo stesso *endpoint*, quindi le loro richieste si accodano. Il tempo di una *query* somma perciò le chiamate ripetute al modello, l'attesa in coda e l'attesa di rete degli strumenti.

Il calo da v2 a v3 ha invece una causa architetturale. In OpenCode l'Orchestrator svolge insieme i ruoli di *Planner* e *Synthesizer* (Sezione 6.5), quindi risparmia un *round-trip* inferenziale per *query*. La v3 non ha inoltre il *Feedback Loop* esplicito e non esegue i cicli di ripianificazione che la v2 attiva sugli scenari più difficili. La nostra strumentazione non ha isolato questo secondo contributo per singola *query*. Il guadagno di latenza deriva quindi in parte dalla minore separazione formale fra ragionamento e sintesi (Sezione 8.3).

Il vantaggio non vale sulla coda della distribuzione: il p_{95} della v3 (243,3 s) supera quello della v2 (174,6 s). In mediana l'orchestratore dichiarativo è più rapido, nei casi peggiori è più lento. Una spiegazione plausibile è che la delega in più passi della v3 non abbia un tetto strutturale equivalente al limite di tre iterazioni del *Feedback Loop* della v2.

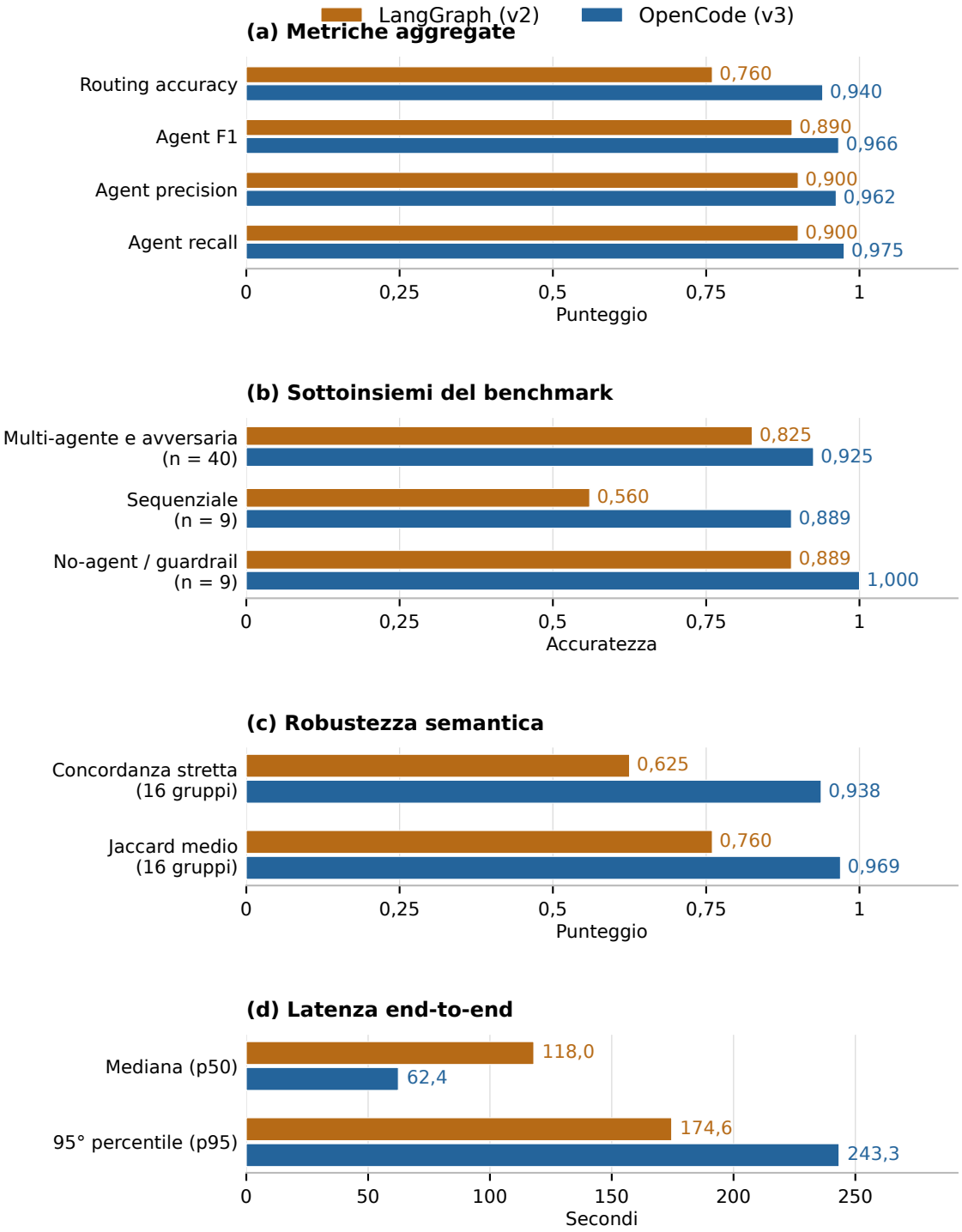


Figura 7.1: Confronto del *routing benchmark* su 100 *query*: orchestratore deterministico (LangGraph, v2) *vs* dichiarativo (OpenCode, v3). (a) metriche aggregate; (b) *routing accuracy* per *pattern*; (c–d) profilo radar delle due architetture (RA = *routing accuracy*, F1 = *Agent* F1).

7.1.3 Allucinazioni di strumento: dal prototipo n8n all'orchestratore v2

Chiudiamo qui il confronto sulle allucinazioni di strumento, cioè sull'invocazione di un agente o *tool* inesistente o irrilevante. È il contrasto posto dalla domanda di ricerca: da un lato un'orchestrazione in cui il modello decide e agisce nello stesso passo, dall'altro una che separa le due funzioni. Sul *benchmark end-to-end* di tirocinio il *Planner* n8n (v1) sbaglia strumento nell'11,1% dei casi (3 su 27). L'orchestratore deterministico v2 (LangGraph) porta lo stesso errore allo 0,7%, una contrazione relativa del 93,7%.

Il calo dipende dalla separazione fra ragionamento ed esecuzione: la scelta dell'agente passa per un campo `tool` vincolato a un `Literal` sui soli agenti disponibili (Tabella 5.3, Capitolo 5). Un piano che nomina uno strumento non enumerato produce un `ValidationError` *prima* dell'invocazione. Il confronto riguarda però due fasi diverse, su campioni distinti e non appaiati (Sezione 8.3). Il disegno non è un test controllato, quindi leggiamo il dato come evidenza descrittiva del passaggio da v1 a v2.

7.1.4 Analisi qualitativa degli errori di *routing*

Le metriche aggregate contano gli errori delle due architetture. Per individuarne la causa esaminiamo tre scenari rappresentativi, scelti nei due *pattern* in cui il *baseline* deterministico v2 sbaglia di più: la delega sequenziale (`C_SEQ`, cinque scenari risolti su nove) e le *query* avversarie (`ADV`). La Tabella 7.3 confronta, per ciascuno, il comportamento atteso dal *gold standard*, la delega osservata nella v2 e quella nella v3.

Tabella 7.3: Tre scenari rappresentativi in cui l'orchestratore deterministico v2 (LangGraph) sbaglia la delega e l'orchestratore dichiarativo v3 (OpenCode) la corregge. Le *query* sono riformulate in sintesi; gli insiemi indicano gli agenti delegati.

Scenario	Richiesta (sintesi)	Gold standard	v2 (osservato)	Tipo di errore v2
`C_SEQ`	"Trova il prodotto di punta del catalogo, poi cerca *online* le recensioni di *quel* modello"	`DB` → `WEB` (sequenza)	`{DB, WEB}` in parallelo	Dipendenza non rilevata: `depends_on` mancante
`ADV`	"Qual è il prezzo *di mercato attuale* del *nostro* server R750?"	`{DB}`	`{DB, WEB}`	Sovra-delega indotta dalla superficie lessicale "mercato attuale"
`ADV`	"Riassumi cosa sai sulla peste nera" (KB interna, formulata come conversazione informale)	`{RAG}`	`{}` (risposta diretta)	Sotto-delega: KB non interrogata

Il primo caso mostra la causa del divario sul *routing* sequenziale. Lo scenario richiede che il secondo agente operi sull'*output* del primo: prima il *DB Agent* individua il modello di punta, poi il *Web Search Agent* ne cerca le recensioni. Il *gold standard* specifica perciò una dipendenza esplicita (`expected_sequence`: `DB` → `WEB`). Il *Planner* v2 produce invece un `WorkflowPlan` in cui i due `WorkflowStep` condividono lo stesso `parallel_group` e nessuno dei due dichiara `depends_on`. I due passi partono insieme. La *query web* è quindi formulata senza il nome del modello, a quel punto ancora ignoto, e il risultato è pertinente solo per coincidenza.

La validazione del piano tipizzato non intercetta l'errore, perché il piano è strutturalmente valido. Lo schema tipizza `depends_on` come lista di identificatori, ma l'*assenza* di una dipendenza necessaria non è esprimibile come vincolo di forma. Il *Route Executor* valuta le dipendenze dichiarate e qui non ne trova nessuna: entrambi i passi risultano pronti e partono nella stessa ondata. L'errore appartiene quindi alla pianificazione semantica, che il controllo di forma non copre. La v3 evita l'errore perché l'Orchestrator delega in due tempi: osserva l'esito del primo `task` prima di formulare il secondo, invece di fissare la composizione in un piano prodotto in un'unica generazione.

Il secondo e il terzo caso mostrano i due tipi di errore avversario. Nel caso di sovra-delega l'espressione "prezzo di mercato attuale" orienta la delega verso il *Web Search Agent*, mentre il possessivo "nostro server R750" lega la richiesta al catalogo interno. Il *gold standard* prevede il solo `DB`. La v2 attiva entrambi gli agenti e introduce nella risposta un prezzo esterno non pertinente.

Nel caso di sotto-delega una richiesta sulla *knowledge base* interna è formulata in tono colloquiale, e il modello la classifica come conversazione generica. L'orchestratore risponde allora con la conoscenza parametrica del modello e non interroga il *RAG Agent*. L'errore è difficile da rilevare, perché la risposta appare plausibile pur non poggiando sulla fonte attesa.

In questi due scenari la formulazione superficiale contraddice l'intento reale: la v2 aggiunge un agente non richiesto oppure ne omette uno necessario. La v3 instrada correttamente entrambi. *Runtime* e *system prompt* dell'Orchestrator non sono separabili (Sezione 7.1.1), quindi non possiamo attribuire la correzione all'uno o all'altro. La correzione poggia comunque sull'aderenza comportamentale del modello, senza una garanzia strutturale equivalente a quella del piano tipizzato. Riprendiamo la distinzione nella discussione dei *trade-off* (Capitolo 8, Sezione 8.3).

7.2 Studio comparativo esteso: RAG classico, avanzato e agente su *file system*

Il recupero è il secondo fronte della valutazione. Confrontiamo tre sistemi RAG: uno classico di riferimento, uno avanzato (ibrido con *reranking*) e un agente su *file system* che adotta la *Direct Corpus Interaction* (Sezione 6.6). I tre dataset mettono alla prova il recupero su aspetti diversi: SciFact [54] ha distrattori vicini alla risposta corretta, HotpotQA [63] richiede due passi di ragionamento, MultiHop-RAG [51] è fatto di articoli interi. Il protocollo è quello della Sezione 4.5.

Il giudice Ragas [9] è `minimax-m3`. Per tutti e tre i sistemi è distinto dal generatore `qwen36-27b-awq-int4`, e questo attenua il *self-evaluation bias* (Sezione 3.10). Le metriche di recupero sono calcolate senza giudice. Presentiamo i risultati su due livelli, prima il recupero puro e poi la generazione *end-to-end*, così da separare l'errore di recupero da quello di sintesi.

7.2.1 Recupero puro: precisione contro copertura

Il primo livello misura solo il recupero, senza generatore né giudice. La Tabella 7.4 riporta le quattro metriche su HotpotQA per i tre sistemi. L'agente su *file system* ottiene l'MRR e la Recall@1 migliori (0,9850 e 0,4850) e la Recall@10 più bassa (0,9050 contro 0,9600 del RAG avanzato). Il dato è coerente con un arresto anticipato della ricerca (*early-stop*): se l'agente si ferma appena trova un documento pertinente, la Recall@1 sale e la Recall@10 scende. La DCI di Li et al. [28] (Sezione 6.6) mostra quindi un compromesso fra localizzazione e copertura: l'agente mette il documento primario in prima posizione più spesso degli altri sistemi, ma raggiunge meno documenti secondari entro i primi dieci.

Tabella 7.4: Metriche di recupero su HotpotQA-100, senza giudice LLM. Per il RAG avanzato è riportata la configurazione di riferimento ibrido+rerank; in grassetto il migliore per colonna.

Sistema	MRR	Recall@10	nDCG@10	Recall@1
RAG classico (vettoriale)	0,9235	0,9500	0,9304	0,4450
RAG avanzato (ibrido+rerank)	0,9389	**0,9600**	**0,9443**	0,4450
Agente su *file system* (DCI)	**0,9850**	0,9050	0,9133	**0,4850**

La difficoltà del recupero dipende dal dataset. Passando da SciFact a HotpotQA il divario di MRR fra i sistemi si riduce: il RAG classico sale da 0,810 a 0,924, il RAG avanzato da 0,914 a 0,939, l'agente da 0,925 a 0,985. SciFact è il dataset più difficile per il recupero, perché i suoi distrattori sono costruiti vicini alla risposta corretta (Sezione 4.5.1). HotpotQA usa invece

i distrattori nativi del dataset, otto paragrafi non rilevanti per *query*. La Figura 7.2 riassume l'MRR dei tre sistemi sui tre dataset.

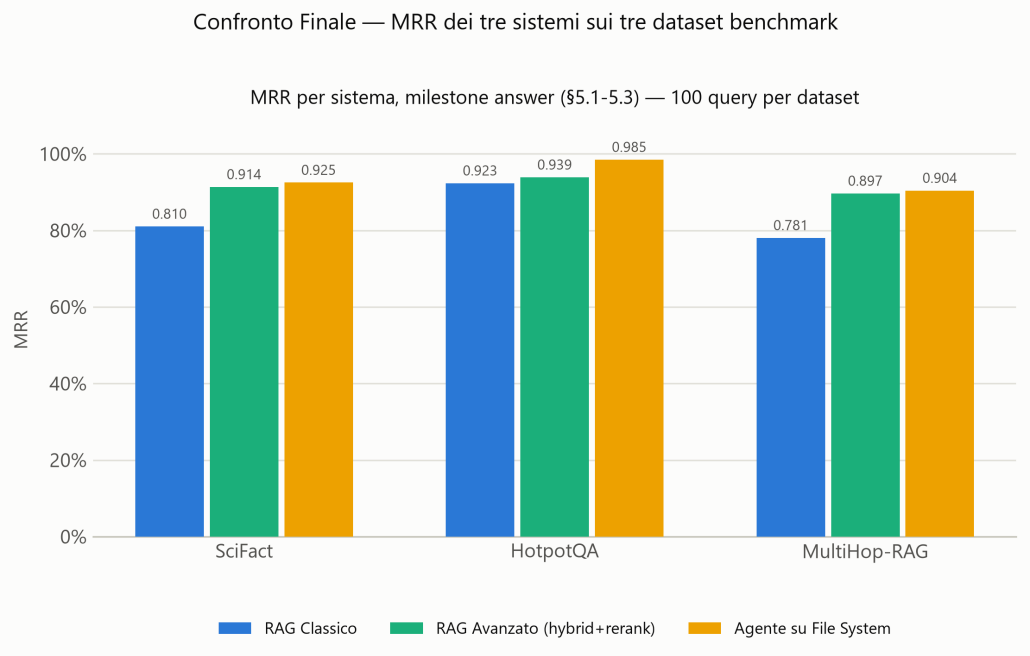


Figura 7.2: MRR dei tre sistemi (RAG classico, RAG avanzato, agente su *file system*) sui tre dataset SciFact, HotpotQA e MultiHop-RAG. L'agente prevale sull'MRR su tutti e tre i dataset e ha la Recall@10 più bassa.

Mediando sui tre dataset il quadro non cambia (Figura 7.3). Sulle metriche di testa l'agente è il migliore: MRR medio 0,9381 contro 0,9164 del RAG avanzato e 0,8382 del classico, Recall@1 media 0,5743 contro 0,5429 e 0,4729. Sulla copertura l'ordine si inverte: il RAG avanzato prevale in Recall@10 media (0,9242 contro 0,8724 dell'agente) e in nDCG@10 media (0,9127 contro 0,8791). Il compromesso osservato su HotpotQA vale quindi sull'intera campagna.

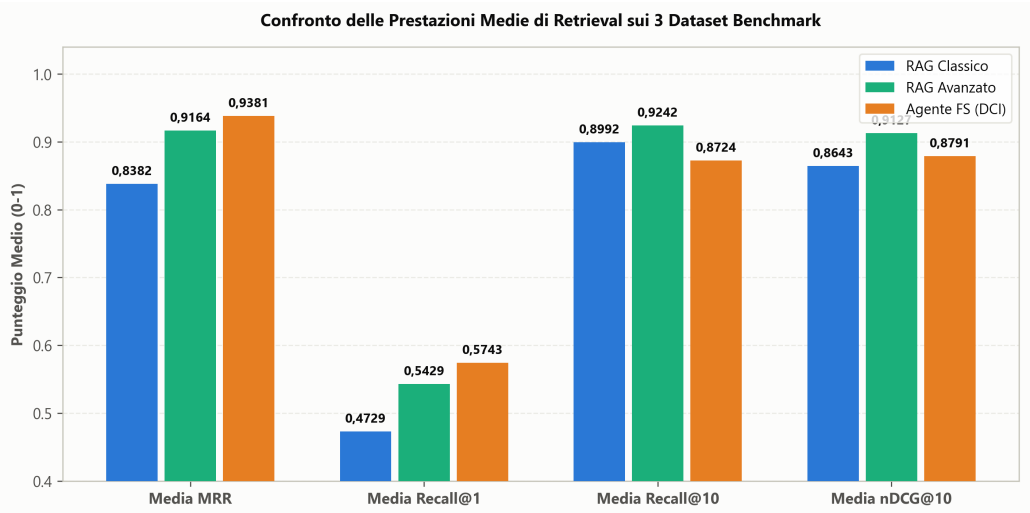


Figura 7.3: Prestazioni medie di recupero (MRR, Recall@1, Recall@10, nDCG@10) dei tre sistemi sui tre dataset. L'agente su *file system* prevale sulle metriche di testa, il RAG avanzato sulla copertura.

7.2.2 La configurazione migliore dipende dal *corpus*

Sotto il nome di RAG avanzato ricade uno spazio di configurazione ampio (Sezione 6.7). Lo analizziamo un componente alla volta, a partire dalla segmentazione del *corpus*.

Per il *chunking* confrontiamo tre strategie, a *token* fissi, semantica e *recursive*, e tre dimensioni di blocco: 256, 512 e 1.024 *token*. A 512 *token* le tre strategie hanno MRR medio vicino, fra 0,834 e 0,844. Il *recursive chunking* ha l'MRR più basso ma la copertura migliore: Recall@10 0,912 contro 0,897 del *chunking* a *token* e 0,896 del semantico, nDCG@10 0,805 contro 0,801 e 0,802. Sulla dimensione del blocco, 256 *token* peggiorano tutte e tre le metriche, mentre 1.024 *token* lasciano l'MRR quasi invariato (0,835) e abbassano la Recall@10 a 0,897. La Figura 7.4 riporta lo *sweep* completo.

Scegliamo il *recursive* a 512 *token* perché ha la Recall@10 più alta: il *reranker* riordina i candidati, ma non può recuperare un documento che il primo stadio non ha trovato. La scelta vale per tutti gli esperimenti successivi, così che le varianti seguenti differiscano solo per il modo in cui recuperano i blocchi.

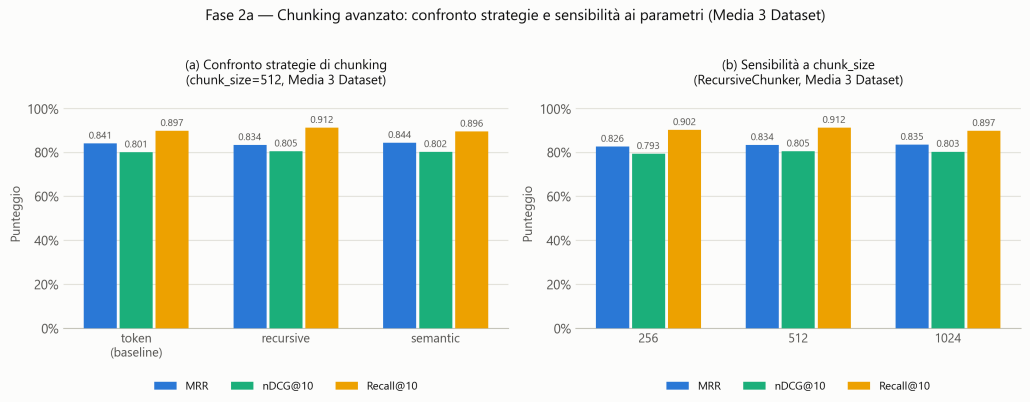


Figura 7.4: *Sweep* delle strategie di *chunking* (a *token*, semantica, *recursive*) e delle dimensioni di blocco (256, 512 e 1.024 *token*), media sui tre dataset. Il *recursive* a 512 *token* è la configurazione adottata nel resto della campagna.

La ricerca ibrida nativa di Qdrant fonde il canale denso e quello sparso con l'algoritmo RRF lato server. Rispetto al solo canale vettoriale, sulla media dei tre dataset l'MRR sale da 0,834 a 0,876 e l'nDCG@10 da 0,805 a 0,825. Il guadagno si concentra nelle prime posizioni della classifica (Figura 7.5): la Recall@1 passa da 0,478 a 0,508 e la Recall@3 da 0,724 a 0,761. Al decimo rango le due curve si incrociano e la ricerca ibrida scende sotto il vettoriale, 0,903 contro 0,912. La fusione RRF migliora quindi l'ordinamento di testa senza allargare la copertura complessiva.

La Tabella 7.5 confronta sei varianti sui tre dataset. La variante migliore dipende dal *corpus* più che dal tipo di *query*, *single-hop* o multi-*hop*. Su SciFact e MultiHop-RAG prevale

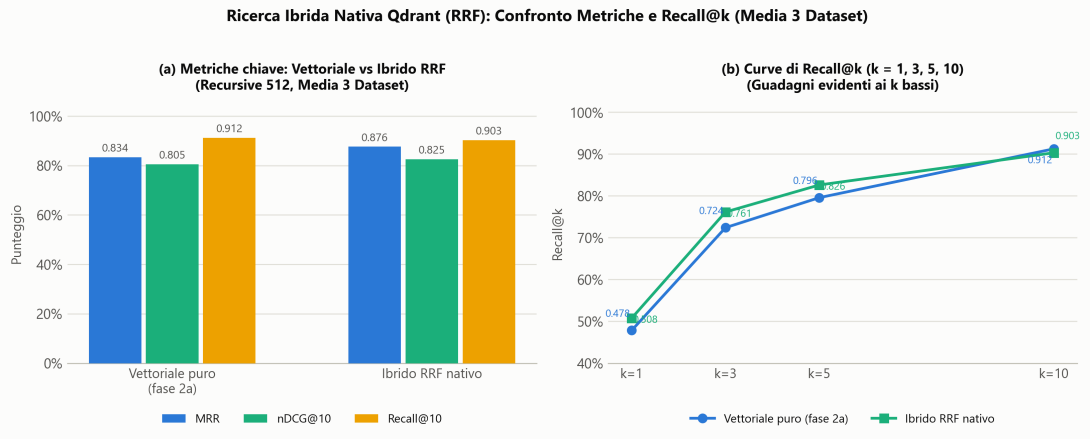


Figura 7.5: Ricerca ibrida nativa Qdrant con fusione RRF contro il solo canale vettoriale, media sui tre dataset. A sinistra le metriche di sintesi, a destra le curve di Recall@k: il vantaggio si concentra ai ranghi bassi e si annulla a $k = 10$.

ibrido+rerank senza fusione (MRR 0,9137 e 0,8967). Su HotpotQA, *pool* di 1.000 paragrafi con otto distrattori per *query*, l'aggiunta della fusione *early* porta l'MRR a 0,9544. Un'interpretazione possibile è che le riformulazioni della *query* aiutino a separare i due paragrafi rilevanti dai distrattori, mentre sugli altri due *corpus* introducano termini generici che allontanano la ricerca dal documento corretto.

Tabella 7.5: MRR di sei varianti del RAG avanzato sui tre dataset. In grassetto il migliore per colonna.

Variante RAG avanzato	SciFact	HotpotQA	MultiHop-RAG
chunk a *token* (vettoriale)	0,8104	0,9235	0,7806
chunk semantico (vettoriale)	0,8287	0,9187	0,7827
ibrido (no rerank, no fusion)	0,8617	0,9220	0,8450
ibrido+fusion *early* (no rerank)	0,8932	0,9209	0,8161
ibrido+rerank (no fusion)	**0,9137**	0,9389	**0,8967**
ibrido+rerank+fusion *early*	0,8909	**0,9544**	0,8389

Sulla stessa variante verifichiamo dove collocare il *reranker* quando è attiva l'espansione della *query*. RAG-Fusion genera tre riformulazioni della domanda, esegue quattro ricerche in parallelo e fonde le liste con RRF. Il *cross-encoder* può intervenire all'interno di ciascuna ricerca, prima della fusione (*early*), oppure una sola volta sul *pool* già unificato (*late*).

Sulla media dei tre dataset il *late rerank* ottiene MRR 0,906 e nDCG@10 0,861, l'*early* 0,895 e 0,857. L'*early* conserva una Recall@10 di poco superiore, 0,927 contro 0,922. Il vantaggio del *late* sull'ordinamento è coerente con il funzionamento di RRF, che combina le liste in base alla posizione e ignora i punteggi del *cross-encoder*: il *reranking* eseguito prima della fusione incide quindi poco sull'ordine finale. Il *late rerank* richiede inoltre una sola chiamata al *reranker* invece di quattro. Nessuna delle due collocazioni supera però il *reranker* senza fusione (MRR 0,920, Recall@10 0,938), che evita anche la generazione delle *query* espanse. La Figura 7.6 riporta il confronto.

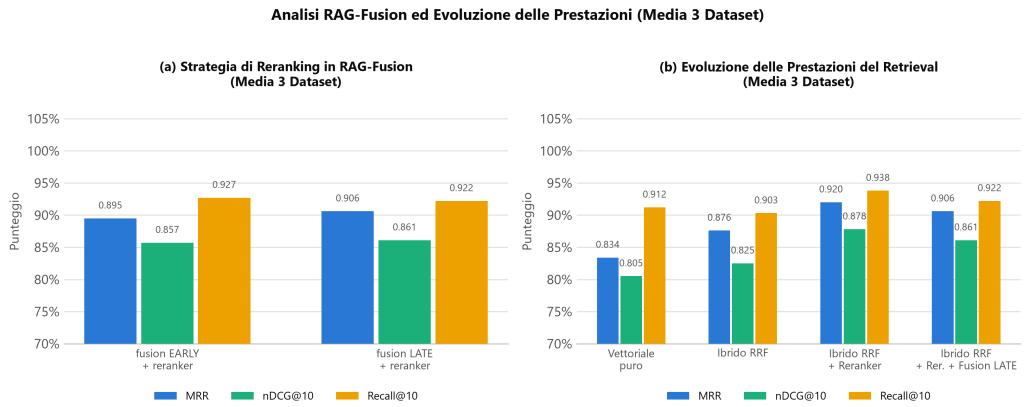


Figura 7.6: Reranking *early* e *late* applicati a RAG-Fusion ed evoluzione delle prestazioni medie rispetto al solo reranker senza fusione, media sui tre dataset.

Occorre poi stabilire quanti candidati passare al *reranker*. Il *reranker* riordina soltanto i documenti che il recupero ibrido gli consegna: un documento pertinente tagliato prima di quel passaggio non rientra nella classifica finale. La nostra ipotesi è che il taglio a 20 candidati per canale escluda troppo presto documenti che il *reranker* porterebbe nelle prime posizioni. La verifichiamo allargando il *pool* a monte: portiamo `dense_k` e `sparse_k` da 20 a 40 e poi a 60, e rimisuriamo il recupero.

Il passaggio da 20 a 40 candidati conferma l'ipotesi, con guadagni piccoli sulla media dei tre dataset. La Recall@10 sale da 0,923 a 0,938, l'nDCG@10 da 0,869 a 0,878 e l'MRR da 0,915 a 0,920: il *reranker* riceve documenti che i canali veloci collocavano oltre la ventesima posizione. Il passaggio a 60 candidati lascia le tre metriche invariate.

Adottiamo quindi 40 candidati per canale come configurazione di riferimento: oltre questa soglia l'ampliamento del *pool* aggiunge solo costo computazionale. Il pannello di sinistra della Figura 7.7 riporta lo *sweep*.

Il pannello destro della stessa figura isola il contributo dei singoli componenti sulla media dei tre dataset. Il *reranker* dà il guadagno maggiore: rispetto alla sola ricerca ibrida l'MRR passa da 0,876 a 0,920, l'nDCG@10 da 0,825 a 0,878 e la Recall@10 da 0,903 a 0,938. Il *cross-encoder* recupera così anche la copertura che la fusione RRF aveva perso rispetto al vettoriale puro (0,912).

Le misure precedenti fissano una scelta per ogni modulo. La Figura 7.8 le raccoglie nella configurazione che adottiamo come RAG avanzato di riferimento. Il *corpus* è segmentato con *recursive chunking* a 512 *token*. Il recupero interroga in parallelo il canale denso e quello sparso di Qdrant e ne fonde i ranghi con RRF. La fusione consegna al *reranker* un *pool* di 40 candidati per canale, che il *cross-encoder* Qwen3 riordina in un solo passaggio prima della generazione. La

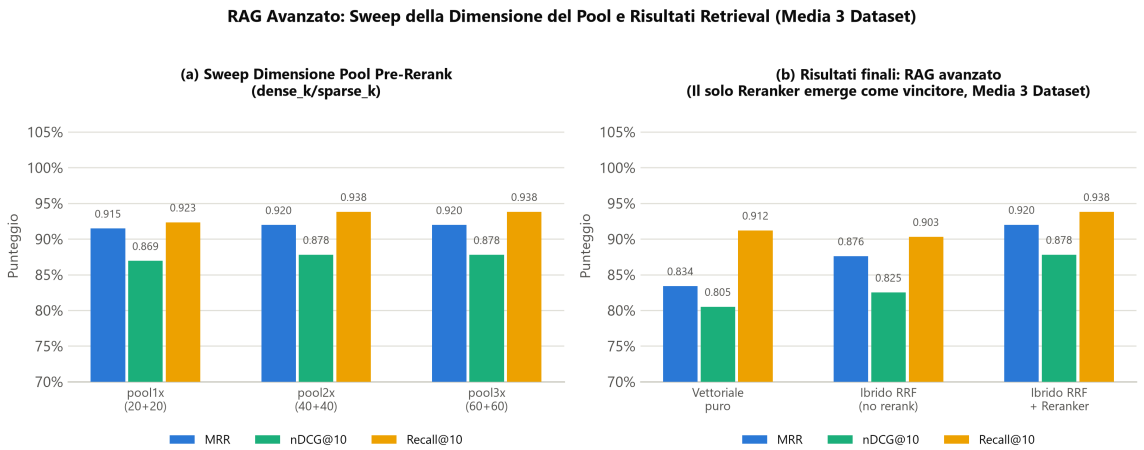


Figura 7.7: A sinistra lo *sweep* del *pool* pre-rerank (`dense_k/sparse_k`): MRR, nDCG@10 e Recall@10 salgono di poco da 20/20 a 40/40 e restano invariati a 60/60. A destra il contributo dei componenti della configurazione sulla media dei tre dataset: il reranker produce il guadagno maggiore.

stessa configurazione è usata nei confronti fra sistemi e nelle misure di generazione che seguono.

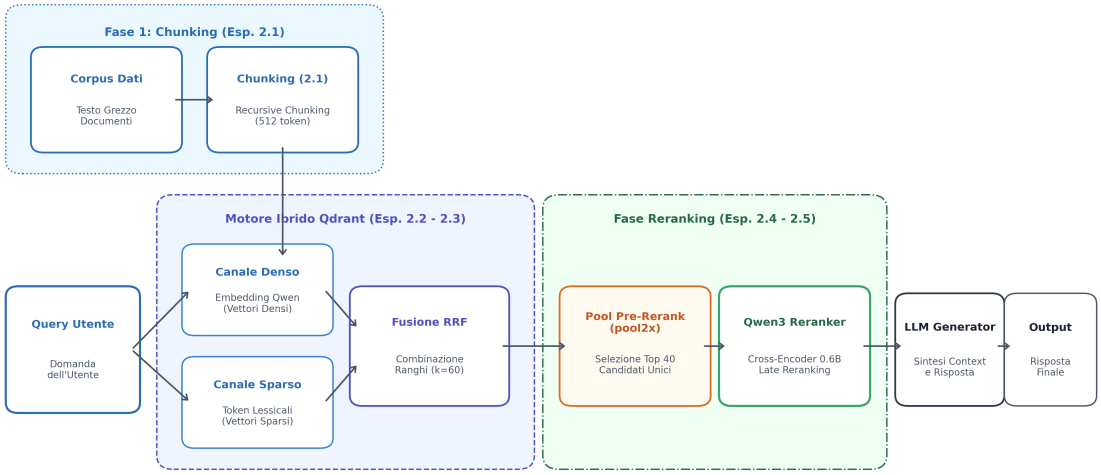


Figura 7.8: Configurazione di riferimento del RAG avanzato, esito dell'*ablation*: *recursive chunking* a 512 *token*, ricerca ibrida Qdrant con fusione RRF dei canali denso e sparso, *pool* pre-rerank di 40 candidati per canale e riordino finale con `qwen3-reranker-0-6b`. La fusione RRF indicata combina i due canali di recupero, non le riformulazioni della *query* di RAG-Fusion.

7.2.3 Scomposizione della *query* contro RAG-Fusion

La scomposizione sequenziale della *query* genera sotto-domande a partire dal contesto già recuperato. Ci aspettavamo che seguisse meglio le entità ponte del multi-*hop*, ma i dati non lo confermano (Tabella 7.6). Su HotpotQA la scomposizione (0,9485) è inferiore alla fusione parallela (0,9544). Su MultiHop-RAG scende a 0,8492, contro 0,8967 della variante senza fusione. Un'ipotesi è che le *query* multiple introducano rumore in un *corpus* di articoli interi.

Variare il numero massimo di passi non cambia il quadro. Su HotpotQA il numero di passi calibrato dà il valore migliore. Su MultiHop-RAG il massimo si ottiene con quattro passi (0,868),

e il quinto passo peggiora il risultato. Anche l'unione di scomposizione e fusione non somma i benefici: la variante combinata è la peggiore su MultiHop-RAG (0,815). Una spiegazione plausibile è che il rumore della fusione *early* si propaghi lungo la catena sequenziale.

Tabella 7.6: MRR di scomposizione della *query* contro RAG-Fusion sui due dataset multi-*hop*. In grassetto il migliore per colonna.

Variante RAG avanzato	HotpotQA	MultiHop-RAG
ibrido+rerank (no fusion)	0,9389	**0,8967**
ibrido+rerank+fusion *early*	**0,9544**	0,8389
ibrido+rerank+scomposizione	0,9485	0,8492

Sulla copertura la scomposizione conserva un vantaggio (Figura 7.9). Sulla media dei due dataset multi-*hop* la sua Recall@10 è 0,919, il valore più alto fra le cinque strategie confrontate, contro 0,895 del solo *reranker* e 0,891 della fusione. Le sotto-domande esplicite allargano quindi l'insieme dei documenti raggiunti, senza migliorarne l'ordinamento. La scomposizione ha anche due controindicazioni. Un errore si propaga lungo la catena: una sotto-domanda formulata male al primo passo invalida le successive. I passi sono inoltre sequenziali e non si parallelizzano, quindi la latenza cresce con il loro numero.

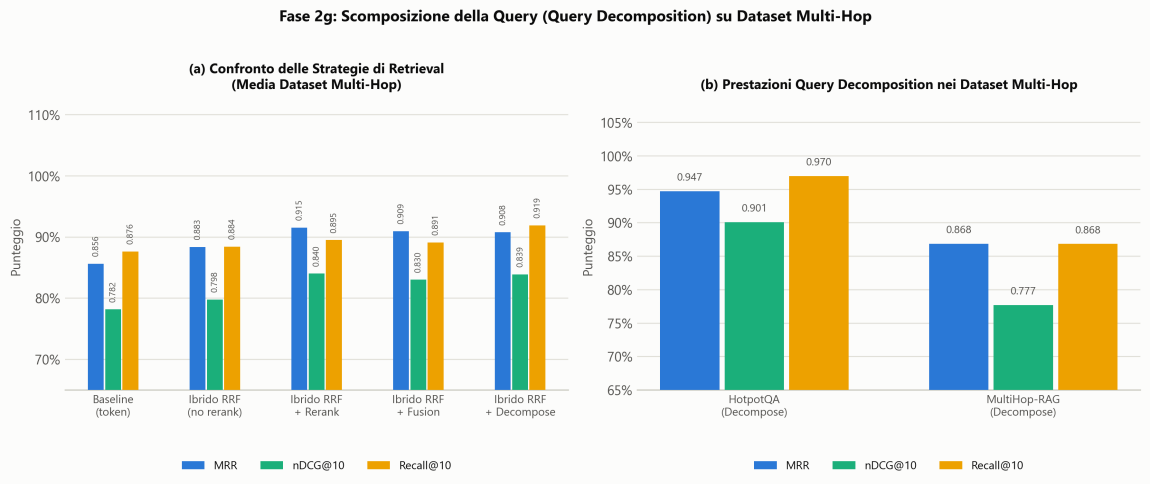


Figura 7.9: A sinistra le cinque strategie di recupero a confronto sulla media dei due dataset multi-*hop*: la scomposizione ottiene la Recall@10 più alta e un MRR inferiore al solo reranker. A destra le prestazioni della scomposizione separate per dataset.

7.2.4 Valutazione *end-to-end*: qualità, verdetto e costo

Il secondo livello valuta la risposta generata e ne rileva il costo. Su SciFact misuriamo la correttezza come *verdict accuracy*, cioè come corrispondenza esatta con il verdetto di riferimento, senza similarità testuale.

Su SciFact (Tabella 7.7) l'agente su *file system* ottiene il miglior recupero (MRR 0,925). In una prima esecuzione la sua *verdict accuracy* era però 0,55, sotto i due sistemi vettoriali.

Analizzando gli errori abbiamo trovato la causa: nella maggior parte dei casi l'agente rispondeva "informazione non presente" pur avendo recuperato il documento corretto. Il difetto era quindi nella sintesi. Il `grep` individuava i documenti giusti, ma il modello evitava il verdetto per eccesso di prudenza (*hedging*).

Abbiamo corretto il difetto con due interventi sul solo *prompt* dell'agente, senza modificare l'orchestrazione. Il primo impone l'inferenza diretta sui testi letti e riserva "informazione non presente" ai casi in cui nessun documento pertinente è stato trovato. Il secondo (Anti-Hedge v2) vieta il ricorso a conoscenza esterna al testo e impone almeno due cicli di ricerca indipendenti prima di chiudere la sessione.

Con la correzione la *verdict accuracy* sale a 0,85: due punti sotto il RAG avanzato (0,87) e tre sopra il RAG classico (0,82). Sull'*answer relevancy* l'agente ottiene 0,625, contro 0,577 del RAG classico e 0,560 dell'avanzato. Il caso mostra che il verdetto dipende sia dal recupero sia dal *prompt* di sintesi. La Figura 7.10 confronta l'agente corretto con i due sistemi vettoriali su SciFact e ne riporta le prestazioni sui tre dataset.

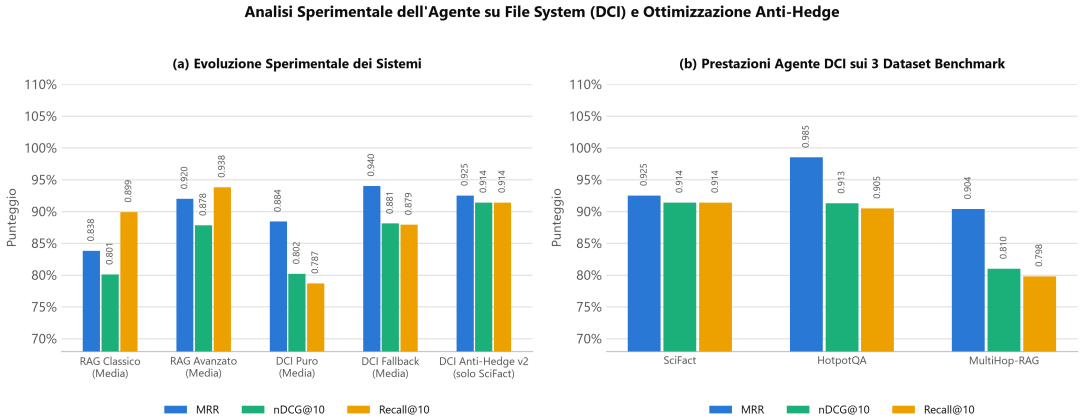


Figura 7.10: A sinistra il confronto di recupero fra RAG classico, RAG avanzato e le configurazioni dell'agente su *file system* (medie sui tre dataset per i primi quattro sistemi e SciFact per Anti-Hedge v2). A destra le prestazioni dell'agente nella configurazione finale con *fallback* RAG-Fusion e *prompt* Anti-Hedge v2 sui tre singoli dataset di benchmark.

Tabella 7.7: Griglia *answer* su SciFact-100, con costo medio per *query* in euro e *token* di ingresso medi. In grassetto il migliore per colonna.

Sistema	MRR	Faithf.	Ans. Rel.	Verdict Acc.	Costo	Token in
RAG classico	0,810	0,787	0,577	0,82	**0,0051 €**	**3 612**
RAG avanzato	0,914	**0,825**	0,560	**0,87**	0,0052 €	3 776
Agente *file system*	**0,925**	0,710	**0,625**	0,85	0,0218 €	72 417

Su HotpotQA e MultiHop-RAG (Tabella 7.8) l'agente ottiene la *faithfulness* più alta, 0,903 e 0,801. Il costo cresce però con la lunghezza dei documenti. Su HotpotQA l'agente costa 2,2 volte il sistema vettoriale più economico. Su MultiHop-RAG, dove il *corpus* è fatto di articoli

interi, costa circa sei volte tanto, con 207 020 *token* di ingresso medi contro i 22 344 del RAG avanzato e i 25 500 del classico.

Sull'*answer correctness* l'agente è primo su entrambi i dataset, 0,698 su HotpotQA e 0,689 su MultiHop-RAG. Il distacco è maggiore su MultiHop-RAG: 12,8 punti sul RAG classico (0,561) e 15,2 sul RAG avanzato (0,537). Una spiegazione plausibile è la lettura integrale degli articoli, che i sistemi vettoriali ricevono invece segmentati in blocchi.

Tabella 7.8: Griglia *answer* su HotpotQA-100 e MultiHop-RAG-100. In grassetto il migliore per colonna e dataset.

Dataset	Sistema	MRR	Faithf.	Ans. Corr.	Costo	Token in
HotpotQA	RAG classico	0,924	0,884	0,664	0,0032 €	1 634
	RAG avanzato	0,939	0,858	0,675	**0,0030 €**	**1 527**
	Agente *file system*	**0,985**	**0,903**	**0,698**	0,0067 €	22 205
MultiHop-RAG	RAG classico	0,781	0,681	0,561	0,0107 €	25 500
	RAG avanzato	0,897	0,621	0,537	**0,0098 €**	**22 344**
	Agente *file system*	**0,904**	**0,801**	**0,689**	0,0594 €	207 020

La Figura 7.11 colloca i nove punti (tre sistemi per tre dataset) sul piano costo/*faithfulness*, con la frontiera di Pareto calcolata per ciascun dataset. Su HotpotQA e MultiHop-RAG i tre sistemi sono tutti sulla frontiera: a un costo maggiore corrisponde sempre una *faithfulness* maggiore. Su SciFact l'agente è l'unico punto dominato. Costa circa quattro volte i sistemi vettoriali e ha una *faithfulness* più bassa di entrambi, 0,710 contro 0,825 del RAG avanzato e 0,787 del classico. Il costo più alto deriva dalla lettura integrale dei documenti candidati.

Sulla qualità del contesto le medie sui tre dataset favoriscono l'agente (Figura 7.12). L'agente su *file system* è il migliore su tutte e tre le voci: *context precision* 0,8091 contro 0,7274 del RAG avanzato e 0,6839 del classico, *context recall* 0,9050 contro 0,8444 e 0,8700, *faithfulness* 0,8051 contro 0,7676 e 0,7838. Sono metriche Ragas, calcolate dal giudice distinto dal generatore descritto all'inizio della Sezione 7.2. La nostra ipotesi è che la lettura diretta dei file mantenga contigue le frasi che il *chunking* separa in blocchi diversi. L'effetto sarebbe più forte sui documenti lunghi di MultiHop-RAG, dove la *faithfulness* del RAG avanzato scende a 0,621 (Tabella 7.8).

Sulla sintesi generativa il vantaggio del contesto si trasmette solo in parte (Figura 7.13). L'agente è primo sull'*answer correctness* media (0,6934 contro 0,6062 del RAG avanzato e 0,6121 del classico) e sull'*answer relevancy* media (0,6623 contro 0,5857 e 0,5989). Sul verdetto di SciFact prevale invece il RAG avanzato, con 0,8687 contro 0,8500 dell'agente e 0,8200 del classico. Un'ipotesi è che il *reranker* porti in testa la frase che contiene l'evidenza e fornisca alla classificazione un contesto senza frasi estranee. Fra RAG classico e RAG avanzato il margine sull'*answer correctness* è di sei millesimi: su cento *query* per dataset non lo consideriamo un

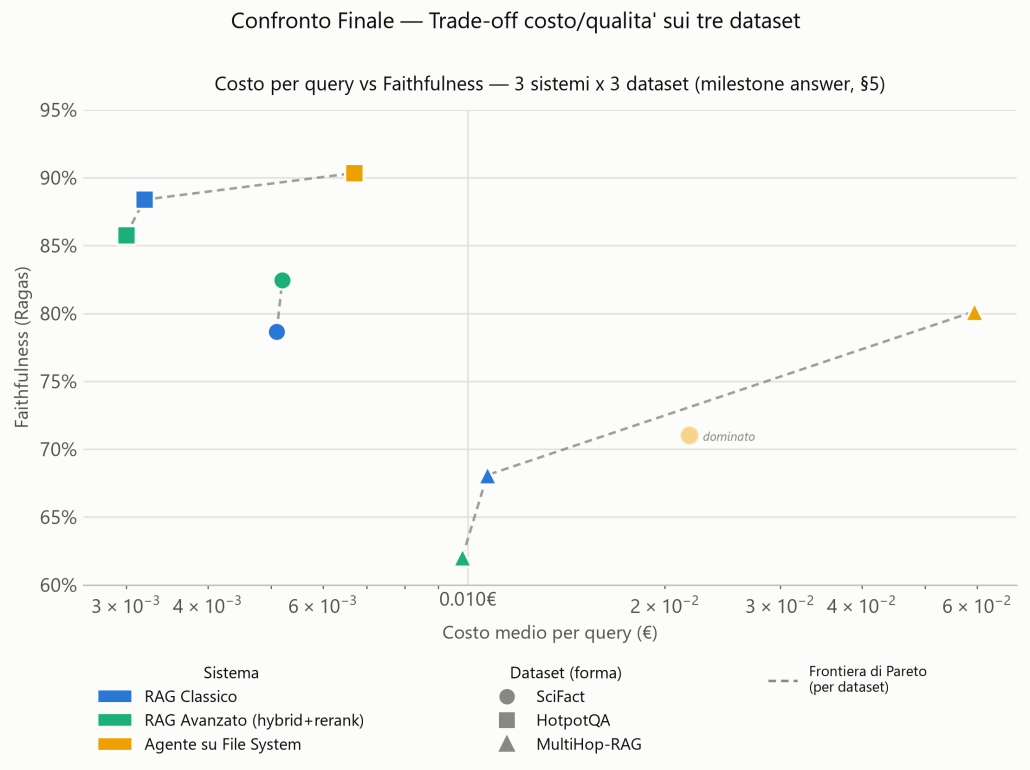


Figura 7.11: Costo medio per *query* (scala logaritmica) contro *faithfulness*, nove punti: tre sistemi per tre dataset. Colore per sistema, forma per dataset. Su SciFact l'agente su *file system* è l'unico punto dominato, limitatamente alla *faithfulness*: sul verdetto il divario dal RAG avanzato si riduce a due punti (0,85 contro 0,87).

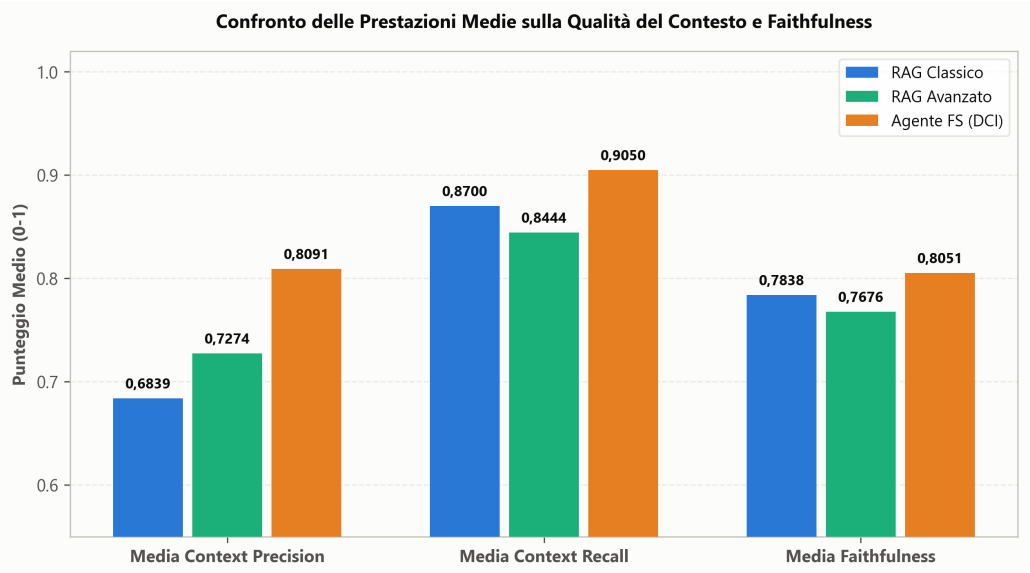


Figura 7.12: Prestazioni medie sulla qualità del contesto (*context precision*, *context recall*) e sulla *faithfulness* dei tre sistemi sui tre dataset.

divario.

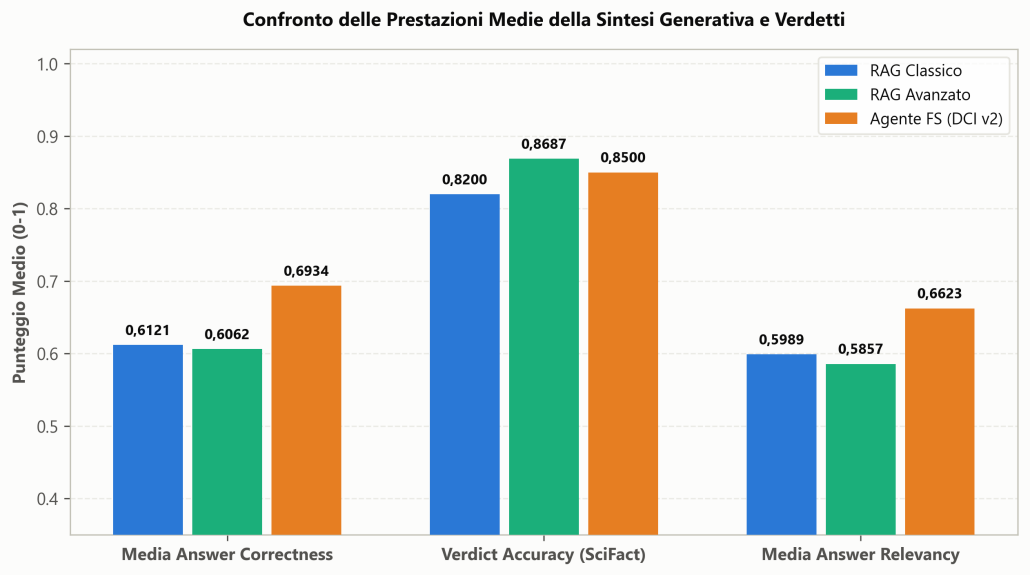


Figura 7.13: Prestazioni medie di sintesi generativa (*answer correctness*, *answer relevancy*) dei tre sistemi sui tre dataset e accuratezza dei verdetti su SciFact.

7.2.5 Sintesi e raccomandazione operativa

La campagna estesa dà un risultato stabile su *corpus* diversi. Sui tre dataset l'agente su *file system* prevale su quasi tutte le metriche di qualità: MRR e Recall@1 medi (Sezione 7.2.1), *context precision*, *context recall*, *faithfulness* e *answer correctness* medie (Sezione 7.2.4). Mette il documento pertinente in prima posizione più spesso degli altri due sistemi e porta in contesto meno testo non pertinente.

Due voci vanno ai sistemi vettoriali. Sulla copertura il RAG avanzato prevale in Recall@10 media (0,9242 contro 0,8724) e in nDCG@10 media (0,9127 contro 0,8791). Sul verdetto di SciFact il RAG avanzato raggiunge 0,8687 contro 0,8500 dell'agente. La minore copertura dell'agente è coerente con l'arresto anticipato descritto nella Sezione 7.2.1.

L'agente costa di più e risponde più lentamente. Il costo medio per *query* è 0,0293 € contro 0,0060 € del RAG avanzato, quasi cinque volte tanto, e la latenza mediana è 73,32 s contro 33,06 s. Su MultiHop-RAG, dove i documenti sono articoli interi, il rapporto di costo sale a sei (Tabella 7.8).

Il bilancio è a favore dell'agente. Le metriche su cui prevale, *faithfulness*, *context precision* e *answer correctness*, misurano la qualità della risposta che l'utente legge, e il giudice che le calcola è distinto dal generatore. Costo e latenza sono invece grandezze su cui possiamo intervenire in progettazione, come verifica la Sezione 7.2.6. Il RAG avanzato è preferibile quando il budget

per *query* è il vincolo principale o quando serve la copertura esaustiva dei dettagli di supporto. Il RAG classico non ha vantaggi di costo sul RAG avanzato, 0,0063 € medi per *query* contro 0,0060 €, e lo teniamo come *baseline* di riferimento.

Il risultato conferma la lettura già emersa dalla Fase 4, questa volta su tre dataset e con giudice distinto dal generatore per tutti e tre i sistemi. La DCI è il *pattern* di recupero più accurato del confronto: forte sulla *faithfulness* e sulla localizzazione del documento primario, più debole sulla copertura dei dettagli secondari. La Sezione 7.2.6 misura una politica di lettura mirata che riduce il costo dell'agente, il suo svantaggio principale rispetto ai sistemi vettoriali.

7.2.6 Oltre la lettura integrale: la politica GRASP

Il costo dell'agente su *file system* (Tabelle 7.7 e 7.8) cresce con la lunghezza dei documenti del *corpus*. La causa è il `read` predefinito dell'agente, che restituisce il file per intero anche quando la porzione utile occupa poche righe. Su MultiHop-RAG, *corpus* di articoli interi, i *token* di ingresso medi arrivano così a 207 020 per *query* (Tabella 7.8). Come riduciamo i *token* letti senza perdere la localizzazione che dà all'agente l'MRR più alto?

A questo scopo adattiamo la *GRanularity-Aware Search Policy* (GRASP) di Gandhi et al. [12]. Nel lavoro originale GRASP è una politica di ricerca addestrata, che recupera evidenza a livello di frase ed estende il contesto solo quando serve. Noi ne riprendiamo il solo principio di lettura, sotto forma di istruzione di *prompt* (Sezione 6.6.1): l'agente legge una finestra di circa quindici righe per lato attorno al *match* del `grep`, invece del file intero.

La campagna è la stessa della Sezione 7.2.4, sui tre dataset, con GRASP attivo. La Tabella 7.9 confronta i risultati aggregati con quelli dell'agente DCI senza la politica.

Tabella 7.9: Agente su *file system*, DCI senza e con politica GRASP. I valori sono medi sui tre dataset, salvo dove la metrica indica il dataset di riferimento fra parentesi. In grassetto il migliore per riga.

Metrica	**Agente DCI**	**Agente DCI + GRASP**
MRR (medio)	0,9381	**0,9383**
Recall@10 (media)	**0,8724**	0,8443
nDCG@10 (medio)	**0,8791**	0,8584
Faithfulness (media)	**0,8051**	0,7972
Answer correctness (media)	0,6934	**0,7197**
Costo medio	0,0293 €	**0,0186** €
Token di ingresso (MultiHop-RAG)	207 020	**116 250**
Latenza mediana	73,32 s	**62,87** s
Latenza p_{95} (MultiHop-RAG)	438,45 s	**163,15** s

Il guadagno maggiore è sulla coda della latenza. Su MultiHop-RAG la latenza p_{95} passa da 438,45 s a 163,15 s (−62,8%): è il *corpus* con i documenti più lunghi, dove la lettura integrale

incideva di più. Il costo medio scende da 0,0293 € a 0,0186 € per *query* (−36,5%) e i *token* di ingresso su MultiHop-RAG passano da 207 020 a 116 250 (−43,8%). L'agente si avvicina così ai sistemi vettoriali senza raggiungerli: su quel *corpus* i suoi *token* di ingresso sono ancora 5,2 volte i 22 344 del RAG avanzato (Tabella 7.8).

L'*answer correctness* media sale da 0,6934 a 0,7197, il valore più alto della campagna, e su HotpotQA passa da 0,698 a 0,724. Su una singola esecuzione, un guadagno medio di 0,026 indica che la finestra mirata non peggiora la sintesi, ma non basta per affermare che la migliori. Un'ipotesi, che qui non possiamo verificare, è che il contesto ridotto attorno al *match* aiuti il modello a restare sul passaggio pertinente. La Figura 7.14 riporta il confronto su qualità media, costo e latenza p_{95}.

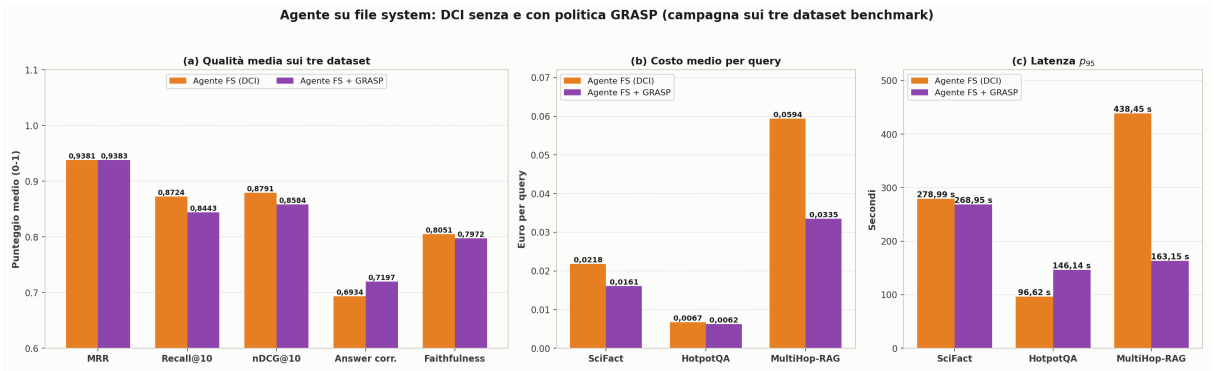


Figura 7.14: Agente su *file system* senza e con politica GRASP. (a) qualità media sui tre dataset; (b) costo medio per *query* per dataset; (c) latenza p_{95} per dataset. Il risparmio si concentra su MultiHop-RAG, il *corpus* di documenti lunghi.

La riduzione del costo abbassa però le metriche di copertura. La Recall@10 media scende da 0,8724 a 0,8443 (−3,2%), l'nDCG@10 medio da 0,8791 a 0,8584 e la *faithfulness* media da 0,8051 a 0,7972 (−1,0%). Il calo è coerente con il meccanismo della finestra: le quindici righe per lato attorno al *match* escludono l'informazione più lontana, che la lettura integrale avrebbe incluso.

Ritroviamo il compromesso fra localizzazione e copertura osservato nella DCI di base (Sezione 7.2.1), su scala più fine: fra porzioni dello stesso documento invece che fra documenti diversi. Lo giudichiamo favorevole. In valore assoluto cediamo 2,8 punti di Recall@10 e 0,8 di *faithfulness*. In cambio la latenza p_{95} su MultiHop-RAG cala del 62,8% e il costo medio del 36,5%. L'ampiezza della finestra è inoltre configurabile, se il caso d'uso richiede più copertura.

La Figura 7.15 riassume il confronto fra i quattro sistemi. Con GRASP l'agente su *file system* mantiene l'MRR medio più alto (0,9383) e raggiunge l'*answer correctness* media più alta (0,7197), 10,8 punti sopra il RAG classico e 11,4 sopra il RAG avanzato.

Costo e latenza sono ancora superiori: 0,0186 € per *query* contro 0,0060 € del RAG avanzato (3,1 volte), e 62,87 s di latenza mediana contro 33,06 s. Il RAG avanzato conserva il primato sulla copertura (Recall@10 media 0,9242) e sul verdetto di SciFact. È quindi la scelta corretta quando il volume di *query* rende il costo il vincolo principale. Negli altri casi preferiamo l'agente con GRASP, che a un costo dello stesso ordine di grandezza guadagna oltre dieci punti di *answer correctness*.

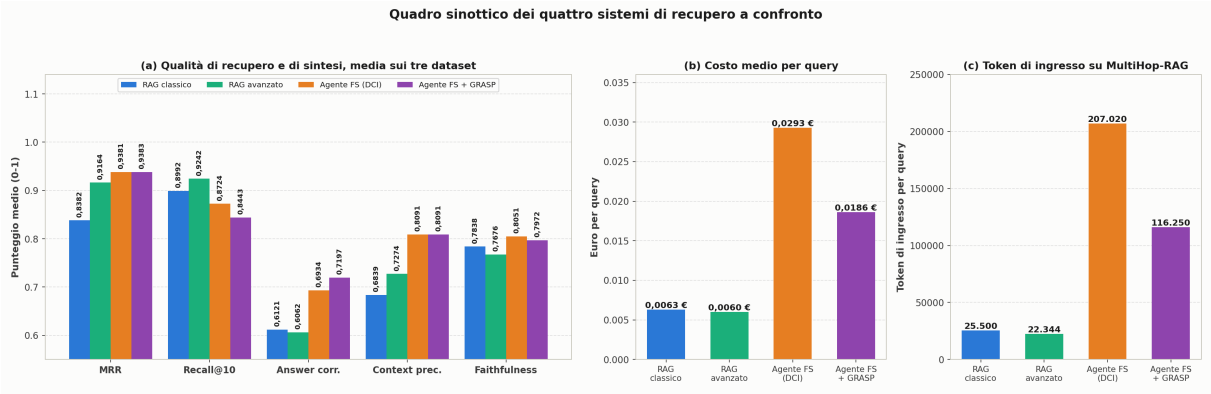


Figura 7.15: Quadro sinottico dei quattro sistemi: RAG classico, RAG avanzato, agente su *file system* e agente con politica GRASP. (a) qualità di recupero e di sintesi, media sui tre dataset; (b) costo medio per *query*; (c) *token* di ingresso su MultiHop-RAG.

7.3 Sintesi rispetto alla domanda di ricerca

I Capitoli 1 e 2 formulano la domanda di ricerca in questi termini:

> **RQ.** *In che misura la separazione formale fra* reasoning *ed* execution*, adottata come invariante architetturale, riduce gli errori di instradamento verso gli agenti rispetto a un'orchestrazione che lascia decisione ed esecuzione nello stesso passo? Misuriamo la riduzione su un banco di 100* query *annotate, con accuratezza di* routing *e F1 per agente.*

Il *baseline* nominato dalla domanda è il prototipo n8n di Fase 1, dove il modello decide e agisce nello stesso passo. L'evidenza su quel contrasto è il calo delle allucinazioni di strumento dall'11,1% allo 0,7% (Sezione 7.1.3). Il *benchmark* di *routing* v2 contro v3 misura invece una grandezza complementare. La v3 separa ragionamento ed esecuzione in modo meno formale della v2, perché il suo Orchestrator pianifica e sintetizza nello stesso agente, eppure instrada meglio. Il risultato indica che la separazione formale non è l'unico fattore che riduce gli errori di delega. Attorno all'affidabilità leggiamo altre due dimensioni, la trasparenza decisionale

e le prestazioni operative: una riduzione degli errori ottenuta con un sistema opaco o troppo costoso non raggiungerebbe l'obiettivo. La Tabella 7.10 associa a ciascuna dimensione l'evidenza raccolta.

Tabella 7.10: Evidenze raccolte per ciascuna dimensione della domanda di ricerca.

Dimensione	**Evidenza primaria**	**Magnitudine**
Affidabilità, contrasto della domanda di ricerca	Allucinazione di strumento: *planner* n8n, con decisione ed esecuzione nello stesso passo, contro orchestratore v2 con piano tipizzato (Sez. 7.1.3)	11,1% (3 su 27) → 0,7%, contrazione relativa del 93,7%; campioni distinti e non appaiati, quindi evidenza descrittiva
Affidabilità, evidenza complementare	*Benchmark* di *routing* deterministico LangGraph *vs* OpenCode, 100 *query*, valutazione programmatica (Tab. 7.2)	*Routing accuracy* 0,76 → 0,94 (+0,18, IC disgiunti), *Agent* F1 0,89 → 0,966; *routing* sequenziale 5/9 → 8/9; 13 *query* del *benchmark* fra gli esempi del *prompt* v3
Trasparenza decisionale	Piano tipizzato Pydantic `WorkflowPlan`; piano serializzato dello scenario B001 (Tab. 5.4, Cap. 5); *checkpoint* LangGraph	Piano JSON auditabile *ex post*; campo `reason` obbligatorio sul piano e *digest* di contesto su ogni *step*; ogni transizione di stato persistita
Prestazioni operative	Latenza *end-to-end* della campagna di *routing* (Tab. 7.2); costo e *token* di ingresso dello studio comparativo sul recupero (Tab. 7.7 e 7.8)	p_{50} 118,0 s → 62,4 s (−47%) fra v2 e v3, con coda p_{95} in peggioramento; sul recupero, costo per *query* da 2,2× (HotpotQA) a 6,1× (MultiHop-RAG) per l'agente su *file system* rispetto al migliore dei sistemi vettoriali

Nel complesso l'evidenza sostiene una risposta affermativa, con quattro qualificazioni.

Il guadagno di affidabilità ha una portata limitata. La contrazione del 93,7% sulle allucinazioni di strumento poggia su campioni distinti e non appaiati. Il +0,18 di *routing accuracy* ha intervalli di confidenza disgiunti. Deriva però da una singola esecuzione su un solo *baseline*, e 13 *query* del *benchmark* compaiono fra gli esempi del *prompt* v3 (Sezione 8.3). Non abbiamo misurato altri domini né altri modelli di ragionamento.

La trasparenza decisionale è garantita per costruzione dal piano tipizzato `WorkflowPlan` e dalla persistenza dei *checkpoint*. Il suo valore operativo dipende però da processi organizzativi esterni al perimetro tecnico.

I difetti residui della v2 riguardano la copertura del *routing* sui *pattern* complessi e multi-fonte (`B_RAG`, `D_DB_RAG`, `ADV`) e sugli scenari sequenziali, cinque su nove. La v3 li riduce quasi del tutto, con *routing accuracy* 0,94, e migliora anche sui *guardrail* (*pattern* `E`: 8/9 → 9/9).

Lo studio comparativo sul recupero (Sezione 7.2) estende la valutazione dal *routing* al RAG,

su tre dataset. L'agente su *file system* è primo su MRR, *context precision*, *faithfulness* e *answer correctness*, e con GRASP il suo costo per *query* scende a 3,1 volte quello del RAG avanzato. I sistemi vettoriali hanno latenza e costo inferiori e prevalgono sulla Recall@10 e sul verdetto di SciFact: *verdict accuracy* 0,87 contro 0,85, dopo la correzione del *prompt* descritta nella Sezione 7.2.4. La separazione *reasoning/execution* ha quindi ridotto le allucinazioni di strumento dall'11,1% allo 0,7%. Restano da migliorare la copertura del *routing* multi-fonte e la copertura esaustiva dell'evidenza multi-*hop*. Il Capitolo 8 discute questi limiti.

8. Discussione dei risultati

Il Capitolo 7 riporta i risultati. Qui ne analizziamo le cause e il significato architetturale. Lo sviluppo procede per incrementi, in cinque fasi: infrastruttura, prototipazione *low-code*, orchestrazione deterministica, *runtime* dichiarativo, ottimizzazione del recupero RAG. Il risultato è un'architettura che sostiene i *task* a composizione multi-*step*. Sui nove *pattern* di delega del *benchmark* la v3 instrada correttamente 94 *query* su 100.

Il *pattern* dall'impatto più misurabile è la separazione *reasoning*/*execution*. La sua efficacia poggia su un meccanismo strutturale: il campo `tool` del piano è vincolato a un `Literal` sui tre agenti disponibili (Tabella 5.3). Un tentativo di invocare uno strumento inesistente produce quindi un `ValidationError`, prima che il *Route Executor* entri in funzione. L'idea di separare pianificazione ed esecuzione è descritta da Wang et al. [56] e ripresa nella rassegna di Wang et al. [55]: l'aggiunta di questo lavoro è la validazione strutturale al confine fra le due fasi.

I limiti residui della v2 sono di natura diversa. Il *routing* copre meno bene i *pattern* complessi e multi-fonte (`B_RAG`, `D_DB_RAG`, `ADV`) e gli scenari sequenziali. La v3 dichiarativa recupera quasi del tutto questo divario (Tabella 7.2).

La scelta ha però un costo: il sistema si adatta poco quando i dati mancano o la richiesta è ambigua. È il primo tema che affrontiamo. Seguono le due fasi successive: la migrazione al *runtime* dichiarativo OpenCode (Sezione 8.3.1) e il perimetro di validità della *Direct Corpus Interaction*, delimitato dallo studio comparativo sul recupero (Sezione 8.3.2).

8.1 Il *trade-off* tra determinismo e flessibilità adattiva

I risultati sostengono il valore del determinismo esecutivo e insieme ne mostrano il limite. L'esecuzione resta legata a un piano validato e a strumenti deterministici. Il sistema perde quindi la libertà di deviare dalla procedura prevista, anche quando la deviazione sarebbe vantaggiosa.

Il limite si manifesta quando manca una sorgente, un dato non è disponibile, oppure si presenta una condizione fuori dal piano. Un agente più libero, modellato su ReAct [64], prova strade alternative: riformula la richiesta, interroga una sorgente affine, propone un *fallback*. La nostra architettura adotta un comportamento conservativo. Se il piano non autorizza la deviazione, il

flusso si arresta oppure torna al *Planner* tramite *Feedback Loop*, fino a tre tentativi prima della *graceful degradation*.

Il meccanismo di *retry* è scritto interamente in Python. A ogni eccezione operativa il *Route Executor* confronta `replan_count` con la costante `_MAX_REPLAN_ATTEMPTS=3`. Sotto la soglia il flusso torna al *Planner*, che ripianifica a partire dall'`ErrorState`. Raggiunta la soglia, il flusso passa comunque al *Synthesizer* con i risultati parziali. La decisione di ritentare resta perciò fuori dall'LLM: il modello viene richiamato solo come *Planner*, su richiesta esplicita e con accesso allo stato d'errore.

Fra le direzioni future indichiamo una variante non implementata: un meccanismo di *Reflexion* [49] fra il fallimento del tentativo n e la ripianificazione $n + 1$, con una breve riflessione testuale prima del nuovo tentativo. È una via per recuperare parte della flessibilità perduta, mantenendo il confinamento esecutivo dell'LLM.

Dal punto di vista *enterprise* il limite di flessibilità è anche un vantaggio. Lo misuriamo come numero di invocazioni di strumenti non dichiarati nel piano. Nella v2 questo numero è zero per costruzione, perché la validazione Pydantic blocca il caso al confine. Il prototipo v1 (n8n), privo di quel vincolo, mostra delega errata e non auditabile nell'11,1% dei casi del *benchmark* del prototipo (Sezione 7.1.3).

8.1.1 Tre condizioni per il vantaggio della separazione

La separazione tra *reasoning* ed *execution* è vantaggiosa quando ricorrono insieme tre condizioni. Le esprimiamo come predicati sul dominio applicativo.

1. **Tolleranza zero per le allucinazioni operative** (C_1). Un *routing* errato può avere conseguenze gravi: una *query* SQL spedita a un agente di ricerca *web*, o il caso opposto. Il confinamento dell'LLM alla pianificazione elimina per costruzione questa classe di errori. Nel nostro dominio il *cost-of-error* è alto, perché un errore espone dati interni a sorgenti pubbliche oppure inserisce dati pubblici non verificati nel catalogo. La contrazione relativa del 93,7% fra il *planner* n8n e l'orchestratore v2 è quindi pertinente (Sezione 7.1.3). Le due misure provengono da campagne distinte e non appaiate, quindi valgono come evidenza descrittiva.

2. **Necessità di auditabilità formale del processo decisionale** (C_2). Il processo può richiedere un'ispezione *a posteriori*, per *compliance*, per *debugging* o per governance. Il GDPR, ad

esempio, richiede la tracciabilità del trattamento. In questi casi il piano JSON validato e il *tracing* deterministico forniscono una catena di responsabilità esplicita, che un ciclo *reasoning-acting* unificato non offre. La condizione ha un costo organizzativo: il *Planner* deve produrre un campo `reason` obbligatorio e ogni transizione di stato deve confluire in un *checkpoint* interrogabile.

3. **Presenza di flussi di lavoro strutturati** (C_3). Il vantaggio è massimo quando i *task* richiedono composizione di strumenti, dipendenze tra *step* e parallelismo controllato. Il *benchmark* lo conferma: il divario tra orchestrazione strutturata e *baseline* si concentra sui *pattern* complessi e multi-fonte (`ADV`, `D_DB_RAG`, *routing* sequenziale). Sui *pattern* a singolo agente le due architetture tendono alla parità (Tabella 7.2). Per *task* semplici, a singolo strumento, il beneficio è marginale e può non ripagare il costo della pianificazione esplicita.

La congiunzione $C_1 \wedge C_2 \wedge C_3$ definisce lo *spazio di applicabilità* naturale dell'architettura. Quando una sola condizione viene meno, il *trade-off* si sposta. Senza C_1 può bastare un agente ReAct [64], senza C_2 il costo di mantenere il piano tipizzato supera il beneficio, senza C_3 il *routing* parallelo via `Send` API non viene mai esercitato.

Per il dominio E4 Analytics le tre condizioni valgono insieme. Il catalogo interno non tollera la fuga verso sorgenti pubbliche (C_1) e le richieste degli analisti confluiscono in report ispezionabili (C_2). I *pattern* a più agenti, paralleli e sequenziali, coprono infine 38 delle 100 *query* del *benchmark* (C_3; Capitolo 4, Tabella 4.3).

8.2 Il principio di restrizione dell'autonomia esecutiva

Il *trade-off* appena descritto poggia su un principio più generale, che resta valido anche cambiando tecnologia. Lo chiamiamo *restrizione dell'autonomia esecutiva*: l'LLM produce una rappresentazione intenzionale del *task* e l'esecuzione è affidata a componenti verificabili. Il principio contribuisce alle architetture cognitive per agenti linguistici [50] e interviene sulla *decision procedure* (Tabella 8.1).

Il fulcro è la *decision procedure*. L'LLM resta confinato al piano e l'esecuzione è delegata a codice deterministico. La validazione Pydantic al confine rende corretta la delega per costruzione, e ogni ottimizzazione *ex post* della scelta dello strumento diventa superflua. Il lavoro lega inoltre in un'unica realizzazione quattro esigenze che il Capitolo 2 individua come trattate in filoni

Tabella 8.1: Mappa fra i componenti funzionali dell'architettura cognitiva per agenti linguistici [50] e i costrutti di *Agentic Platform*.

Componente Sumers et al.	**Costrutto in Agentic Platform**	**Localizzazione architetturale**
Core reasoning	Modello servito via vLLM (*on-premises*, H100 80 GB): Qwen3-Coder-30B-A3B-Instruct nello sviluppo, Qwen3.6-27B-AWQ-INT4 nella campagna comparativa	Nodi *Planner* e *Synthesizer*
Memory (long-term)	PostgreSQL + pgvector *dual-table* (catalogo, profilo utente, storia)	*Data layer* (Capitolo 5)
Memory (working)	`AgentState` (`TypedDict`) e *checkpoint* LangGraph	Stato condiviso del grafo
Action space	Quattro agenti specializzati (DB, *Web Search*, RAG, *Deep Research*) con interfacce tipizzate	Nodi terminali del *fan-out*; *Deep Research* su flusso dedicato
Decision procedure	Pianificazione LLM + *routing* deterministico Python	*Planner* + *Route Executor*

separati (Sezione 2.9): infrastruttura *cloud-native*, pianificazione tipizzata, esecuzione auditabile e governance *enterprise*. Quest'ultima è intesa come esercizio *on-premises* con controllo umano sui flussi a costo elevato.

8.3 Analisi dei limiti e dei *trade-off* architetturali

8.3.1 *Trade-off* della migrazione a OpenCode

La valutazione v3 mostra un miglioramento del *routing* rispetto alla v2 (Sezione 7.1): la *routing accuracy* passa da 0,76 a 0,94 e l'*Agent* F1 da 0,89 a 0,966 (Tabella 7.2). Il vantaggio si concentra sui *pattern* complessi, e sui *guardrail* (*pattern* `E`) il punteggio sale da 0,889 a 1,000. Il confronto avviene a parità di modello di ragionamento e di *embedder* (Capitolo 7, Sezione 7.1.1), e il guadagno non dipende da un modello più capace.

Restano tre cautele di lettura. Il confronto riguarda fasi successive del progetto, su una singola esecuzione per braccio. Il *runtime* inoltre non è separabile dai *prompt* di sistema che lo accompagnano.

La terza cautela pesa di più. L'ultima revisione del *prompt* dell'Orchestrator precedente la campagna (`cc82e07`, 16 giugno 2026) usa come esempi 13 delle 100 *query* del *benchmark*. Le cita per identificativo, insieme all'instradamento corretto. Fra queste figurano 5 delle 11 *query* `ADV` e 2 delle 9 `C_SEQ`, cioè i *pattern* su cui la v3 guadagna di più. Il *prompt* della

v2 non contiene esempi tratti dal *benchmark*. Una parte del vantaggio della v3 può quindi derivare dall'adattamento del *prompt* alle *query* di test. Per stimarne l'entità serve un ricalcolo delle metriche sulle 87 *query* rimanenti, oppure una nuova campagna. Leggiamo perciò il miglioramento come proprietà dell'architettura dichiarativa nel suo insieme, *prompt* compresi, e con questo limite.

Il bilancio della migrazione presenta guadagni e perdite distinti. Guadagniamo manutenibilità sull'operazione più ricorrente, l'aggiunta di un agente: secondo le stime a posteriori della Tabella 6.1, i file da modificare passano da quattro a uno o due, le ore-uomo da otto-dodici a meno di due. La configurazione è inoltre raccolta in sei file dichiarativi. Perdiamo verificabilità della separazione *reasoning*/*execution*, perché l'Orchestrator fonde pianificazione e sintesi. Assumiamo poi una dipendenza da OpenCode, *runtime* esterno più recente e meno diffuso di LangGraph (Capitolo 6).

Fra le due versioni la differenza sostanziale riguarda la natura della garanzia. Nella v2 la separazione è dimostrabile *per costruzione*: un `Literal` Pydantic produce un errore di validazione statico. Nella v3 è garantita *per aderenza*: `permission: deny` blocca i *tool* operativi, mentre il perimetro comportamentale resta affidato ai *prompt*. Il vincolo del *prompt* è più fragile sulle richieste fuori scopo. Per contesti a tolleranza zero (C_1 stringente) l'*enforcement* imperativo offre garanzie più forti. Dove prevalgono velocità di iterazione e standardizzazione conviene il paradigma dichiarativo, a patto di estendere l'*enforcement* dei *guardrail* dal *prompt* ai permessi.

8.3.2 Limiti della *Direct Corpus Interaction*

Lo studio comparativo esteso delimita il perimetro di validità della DCI (Capitolo 7, Sezione 7.2; Capitolo 2, Sezione 2.5.2). Il confronto impiega un giudice indipendente dal generatore, su tre sistemi e tre dataset. Il risultato dipende dal compito. L'agente su *file system* ottiene la *faithfulness* più alta su HotpotQA e MultiHop-RAG, 0,903 e 0,801, dove il compito consiste nel sintetizzare. Ottiene la più bassa su SciFact, 0,710, dove il compito consiste nel decidere un verdetto (Tabelle 7.7 e 7.8). Il limite riguarda quindi la copertura esaustiva dell'evidenza e, soprattutto, la classificazione: il `grep` localizza bene il documento.

I risultati su SciFact lo confermano. Il compito richiede di decidere un verdetto sulla base dell'evidenza recuperata. In una prima diagnosi l'accuratezza scendeva a 0,55, contro 0,82 e 0,87 dei due sistemi vettoriali, *pur avendo l'agente recuperato il documento giusto*. La causa era

di sintesi: un eccesso di prudenza (*hedging*) spingeva l'agente a rispondere "informazione non presente" anche quando l'evidenza nel contesto lo permetteva.

Correggiamo il difetto con un intervento mirato sul solo *prompt* (Anti-Hedge v2, Sezione 7.2.4), che porta la *verdict accuracy* a 0,85, a due punti dal RAG avanzato. Il *fix* conferma però la distinzione fra garanzia *per costruzione* e garanzia *per aderenza* discussa sopra. Il correttivo risiede soltanto nel *prompt*, senza un vincolo strutturale a sostenerlo, ed è quindi più fragile su domini o formulazioni che la campagna di test non copre.

L'ultima voce è il costo, la più sfavorevole all'agente e insieme quella su cui abbiamo il margine di intervento maggiore. Nella sua forma di base l'agente scala male con la lunghezza dei documenti. Su HotpotQA costa poco più del doppio degli altri sistemi, su MultiHop-RAG circa sei volte, perché lì il *corpus* è fatto di articoli interi: 207 020 *token* di ingresso medi contro i 22 344 del RAG avanzato e i 25 500 del classico.

Quel divario annulla il vantaggio di qualità dell'agente? La diagnosi isola una causa precisa: la lettura integrale del file dopo un `grep` che ha già individuato la riga utile. Su quella causa interviene in modo mirato la politica GRASP (Sezione 7.2.6). I *token* di ingresso su MultiHop-RAG scendono a 116 250, il costo medio del 36,5% e la latenza p_{95} del 62,8%. Il divario col RAG avanzato si riduce, senza chiudersi.

Sul piano della qualità la correzione ha un costo. GRASP cede 3,2% di Recall@10 e 1,0% di *faithfulness* media, perché la finestra di quindici righe riduce la probabilità che l'agente incontri evidenza rilevante situata oltre quel raggio. Ritroviamo qui, su scala più fine, lo stesso *trade-off* fra localizzazione e copertura che governa la DCI nel suo complesso: la finestra sta al documento come il documento sta al *corpus*.

Il compromesso ci sembra favorevole: la rinuncia qualitativa si misura in pochi punti percentuali, il risparmio in decine (Sezione 7.2.6). È comunque una scelta parametrica, da calibrare sul dominio, non una regola universale.

Il bilancio complessivo resta a favore dell'agente. Sulla campagna estesa è il sistema più accurato del confronto, primo su MRR, *context precision*, *faithfulness* e *answer correctness* (Sezione 7.2.5). Con GRASP il suo costo rientra nell'ordine di grandezza dei sistemi vettoriali. I limiti discussi qui delimitano il perimetro di quel primato, senza annullarlo.

La DCI conserva quindi un perimetro di validità definito. È preferibile in tre situazioni. La prima è un *corpus* circoscritto, dove l'*overhead* del *vector store* pesa. La seconda è il requisito di auditabilità del *retrieval* (C_2): ogni `grep` e ogni `read` sono tracciati, mentre una similarità coseno

resta opaca. La terza è il caso in cui la fedeltà della risposta prevale sul costo.

Entro quel perimetro la DCI si configura come *pattern* di recupero forte su due fronti, la fedeltà generativa nei compiti di sintesi e la localizzazione del documento primario. È più debole sulla copertura esaustiva, quando le prove sono sparse, e sulla classificazione di un verdetto. Fra le direzioni future indichiamo un'architettura ibrida, con recupero denso per la selezione e DCI per l'estrazione verificabile.

Le due analisi, sull'orchestrazione e sul recupero, portano alla stessa conclusione. Il beneficio dell'architettura consiste nella controllabilità del comportamento, nella gestione *fail-transparent* del fallimento e nella compatibilità con audit e governance. Sono proprietà dimostrabili per costruzione sotto $C_1 \wedge C_2 \wedge C_3$. Fuori da quelle condizioni, in scenari creativi o destrutturati, ReAct [64] con Reflexion [49] può risultare più efficace. Il Capitolo 9 formalizza la risposta alla domanda di ricerca e le direzioni future.

9. Conclusioni e sviluppi futuri

Il lavoro muove dal principio architetturale enunciato nel Capitolo 1 (Sezione 1.3). Lo implementiamo su LangGraph e su OpenCode (Capitoli 5 e 6) e lo verifichiamo sui dati (Capitolo 7). Questo capitolo ne presenta il bilancio. La Sezione 9.1 risponde alla domanda di ricerca, la Sezione 9.2 dichiara i cinque piani del contributo. Seguono le direzioni di ricerca (Sezione 9.3) e le raccomandazioni operative per chi adotta sistemi agentici in produzione (Sezione 9.4).

9.1 Risposta alla domanda di ricerca

La domanda di ricerca del Capitolo 1 riguarda la separazione formale fra *reasoning* ed *execution*, adottata come invariante architetturale. Chiede in che misura quella separazione riduca gli errori di instradamento verso gli agenti. Il termine di paragone è un'orchestrazione che lascia decisione ed esecuzione nello stesso passo. La misura è un banco di 100 *query* annotate, con accuratezza di *routing* e F1 per agente. L'abbiamo articolata in tre sotto-domande: il formalismo della separazione, il processo che porta il prototipo in esercizio, il sostrato che ospita l'architettura. Rispondiamo nello stesso ordine.

RQ1 (formalismo). La risposta è costruttiva. La formulazione che riduce in modo misurabile gli errori di *routing* è la coppia *piano tipizzato* + *esecutore deterministico*. La formalizziamo nel piano di *workflow*, validato dai modelli Pydantic `WorkflowPlan` e `WorkflowStep` (Sezione 5.4). Il campo `tool` del piano è vincolato a un tipo `Literal` sull'insieme chiuso degli agenti disponibili. Se l'LLM invoca uno strumento inesistente, la validazione del piano produce un `ValidationError` prima che il *Route Executor* avvii l'esecuzione (Sezione 5.5). L'esecutore è privo di componenti probabilistiche.

L'architettura esclude così l'allucinazione di strumento *per costruzione*. L'errore è impossibile al confine fra il piano e la sua esecuzione: nessun ciclo di *retry* probabilistico interviene a valle per correggerlo. La sola componente probabilistica residua è il *Synthesizer*, dove l'errore è isolabile. Le evidenze quantitative confermano l'effetto: −93,7% allucinazioni di strumento nel passaggio n8n→LangGraph e *routing accuracy* 0,76 → 0,94 nel passaggio LangGraph→OpenCode. La Tabella 7.2 e la Sezione 7.3 del Capitolo 7 le raccolgono per intero, e il Capitolo 8 ne discute la portata.

RQ2 (processo). La risposta è la sequenza di fasi che abbiamo documentato passo per passo: infrastruttura Kubernetes (Tabella 7.1), prototipazione su n8n, reimplementazione su LangGraph (Capitolo 5). Selezioniamo n8n dopo il confronto con Dify e Langflow (Sezione 4.3). Ogni transizione deriva da un'evidenza empirica.

Le sei criticità del prototipo *low-code* (Tabella 1.1) emergono soltanto dalla fase preliminare di prototipazione. La prima è quantificata sul campione di test, con 6 *query* su 50 terminate senza risultato utile. Le altre cinque provengono dall'ispezione dei *workflow* e restano in forma qualitativa. A esse si aggiunge il tasso di allucinazione di strumento dell'11,1%, rilevato sul *benchmark end-to-end* di tirocinio (Sezione 7.1.3).

Il piano JSON validato da Pydantic rende poi ispezionabile ogni decisione prima dell'esecuzione. Il *routing* è auditabile al 100%, mentre la sintesi finale resta probabilistica. Ogni passo del piano è quindi tracciabile prima che produca effetti. L'*human-in-the-loop* sul *Deep Research* (Sezione 5.8) aggiunge un punto di controllo esplicito sulle operazioni a maggiore impatto.

RQ3 (sostrato). La risposta dichiara anche i costi. Dalla soluzione *low-code* a quella imperativa guadagniamo affidabilità e *type safety*, al prezzo della velocità di prototipazione (Sezione 5.9, Tabella 5.6). Dall'imperativo al dichiarativo, con OpenCode, guadagniamo manutenibilità e velocità di iterazione. Per aggiungere un agente i file da modificare passano da quattro a uno o due, e le ore-uomo da otto-dodici a meno di due (Tabella 6.1).

Perdiamo però la dimostrabilità statica della separazione. L'*enforcement* passa da *per costruzione* a *per aderenza*, e i *guardrail* diventano più fragili perché il perimetro comportamentale è demandato ai *prompt*. Il Capitolo 8 (Sezione 8.1) analizza questa ridistribuzione del rischio.

Sul recupero, lo studio comparativo (Sezione 7.2) delimita il perimetro di validità della *Direct Corpus Interaction* (Sezione 6.6). L'agente su *file system* ottiene la fedeltà generativa più alta nei compiti di sintesi. Resta sotto il RAG avanzato sulla classificazione di un verdetto e sulla copertura esaustiva dell'evidenza, con un costo che cresce sui documenti lunghi.

I due movimenti hanno segno opposto: il passaggio all'imperativo aggiunge garanzie statiche, il passaggio al dichiarativo le converte in convenzioni. Entrambi hanno un prezzo. La risposta complessiva descrive quindi uno spazio di *trade-off*. Il determinismo esecutivo giustifica il proprio costo dove valgono C_1–C_3 (Capitolo 8); ogni allentamento del vincolo ne sposta il peso su altri assi. La Tabella 9.1 riassume la verifica delle tre dimensioni contro l'evidenza raccolta.

Tabella 9.1: Verifica delle tre dimensioni della domanda di ricerca contro l'evidenza empirica raccolta nel Capitolo 7.

Dimensione	Verificata	Evidenza primaria
Affidabilità	Sì	−93,7% allucinazioni di strumento (n8n→LangGraph, variazione descrittiva longitudinale); *routing accuracy* 0,76 → 0,94 (LangGraph→OpenCode, IC non sovrapposti).
Trasparenza decisionale	Sì (parziale)	Piano JSON Pydantic ispezionabile (100% *routing* auditabile); HITL su *Deep Research*; *Synthesizer* probabilistico residuo.
Prestazioni	Sì (parziale)	Latenza mediana quasi dimezzata nel passaggio v2→v3 (p_{50} 118,0 → 62,4 s), con coda peggiore (p_{95} 174,6 → 243,3 s): il *runtime* dichiarativo accelera il caso tipico e allunga il caso peggiore.

9.2 Contributi del lavoro

Il contributo si articola su cinque piani, e ciascuno è verificabile in modo indipendente.

- **Ingegneristico.** Documentiamo un percorso incrementale completo (Capitoli 3–6), dall'infrastruttura *cloud-native* (RKE2) alla prototipazione su n8n fino all'implementazione *production-grade* su LangGraph. Le transizioni sono documentate con le misure che le motivano, dalle 6 *query* su 50 chiuse senza risultato utile all'11,1% di allucinazione di strumento del prototipo. Il percorso è replicabile da organizzazioni con *data layer* relazionale e vincolo di sovranità del dato.

- **Architetturale.** Applichiamo la separazione *reasoning/execution* tramite la coppia *Planner–Route Executor* (Capitolo 5). Nel passaggio da n8n a LangGraph le allucinazioni di strumento si contraggono del 93,7%: le due misure provengono da campagne distinte, quindi valgono come evidenza descrittiva. La separazione è formalizzata come invariante strutturale nel piano di *workflow*, validato dai modelli Pydantic `WorkflowPlan` e `WorkflowStep` (Sezione 5.4).

- **Metodologico.** Forniamo due strumenti di valutazione replicabili su altri sistemi agentici. Il *benchmark* di *routing* programmatico valuta le decisioni contro un *gold standard*, ed è quindi immune al *self-enhancement bias* (Sezione 4.4). Il protocollo di valutazione RAG a doppio *milestone* separa il giudice dal generatore e verifica l'integrità della misura (Sezione 4.5).

- **Comparativo.** Conduciamo due confronti diretti sullo stesso dominio. Il primo mette a confronto orchestrazione imperativa e dichiarativa sulle medesime 100 *query* di *routing*, e quantifica il *trade-off per costruzione/per aderenza* (Capitolo 8, Sezione 8.1). Il secondo riguarda il recupero: confrontiamo la via vettoriale e la *Direct Corpus Interaction* su tre sistemi RAG e tre dataset, con giudice indipendente dal generatore (Sezioni 7.2 e 8.3.2). La DCI raggiunge la fedeltà generativa più alta nei compiti di sintesi, 0,903 su HotpotQA e 0,801 su MultiHop-RAG, e resta sotto gli altri due sistemi dove il compito è classificare un verdetto. Su SciFact la correzione del *prompt* sull'eccesso di prudenza porta la *verdict accuracy* a 0,85, due punti sotto il RAG avanzato. I due confronti estendono al paradigma dichiarativo il tipo di analisi che la letteratura ha finora condotto sui soli orchestratori imperativi (Capitolo 2, Sezione 2.9).

- **Ottimizzativo.** Adattiamo alla *Direct Corpus Interaction* di Li et al. [28] la lettura mirata della *GRanularity-Aware Search Policy* (GRASP) di Gandhi et al. [12] (Sezione 6.6.1). Nel lavoro originale GRASP è una *policy* addestrata per rinforzo. Ne riproduciamo la sola lettura di paragrafo, senza addestramento, con un'istruzione di *prompt*: una finestra calcolata attorno alla riga individuata dal `grep` sostituisce la lettura integrale del file. Il nostro apporto è la misura di questo adattamento su tre dataset. La politica abbassa il costo che rende la *Direct Corpus Interaction* poco sostenibile sui documenti lunghi: $-43{,}8\%$ *token* di ingresso su MultiHop-RAG, $-36{,}5\%$ costo medio e $-62{,}8\%$ latenza p_{95}. Il costo è una perdita di 3,2% di Recall@10 e di 1,0% di *faithfulness* (Sezione 7.2.6). Il vincolo è definito negli artefatti dichiarativi già presenti, *prompt* e parametri del *tool*, e per questo si trasferisce a qualunque agente con permessi di sola lettura sul *filesystem*.

9.3 Direzioni future di ricerca

Le direzioni che seguono derivano dai limiti discussi nel Capitolo 8 (Sezione 8.3) e si distribuiscono su cinque filoni.

Valutatore eterogeneo per il RAG. Il *self-evaluation bias* del giudizio Ragas emerge quando giudice e generatore coincidono. Per affrontarlo appieno occorre un *ensemble* di giudici di famiglie diverse, per esempio Llama 3, Mistral Large e Qwen3-Coder-30B-A3B-Instruct, in protocollo *cross-family judging* [52]. Lo studio comparativo esteso (Sezione 7.2) copre solo la

prima parte del requisito: separa il giudice (`minimax-m3`) dal generatore (Qwen3.6-27B-AWQ-INT4). Resta da completare l'*ensemble*, che serve a rendere comparabili con la letteratura i valori assoluti oltre al confronto relativo già condotto. Complementare è l'adozione di *rubric* a scala continua in stile Prometheus [21].

Recupero ibrido denso-lessicale. Il deficit della DCI sta nella copertura esaustiva dell'evidenza. Sulla localizzazione del documento primario l'agente ottiene invece l'MRR più alto dei tre sistemi, 0,985 su HotpotQA. L'architettura candidata è quindi ibrida e a due stadi, con recupero denso per selezionare i candidati e DCI per l'estrazione verificabile. La verifichiamo ripetendo lo studio comparativo con *Context Precision* e copertura completa dell'evidenza come metriche primarie.

Finestra di lettura adattiva. GRASP (Sezione 7.2.6) fissa il raggio di lettura a un valore costante di circa quindici righe, calibrato sui tre *corpus* della campagna. La scelta è deliberatamente conservativa e lascia due questioni aperte. La prima riguarda la dipendenza dal dominio: un *corpus* di testo scientifico denso e uno di articoli discorsivi non hanno ragione di condividere lo stesso raggio ottimale. La seconda riguarda l'adattività. Una finestra decisa a *runtime* in base al tipo di corrispondenza trovata potrebbe recuperare parte del 3,2% di Recall@10 che oggi perdiamo. Le due questioni si affrontano con uno *sweep* sul raggio, esteso a *corpus* di generi diversi.

Chiudere il divario residuo sulla verifica dei fatti. La correzione mirata al *prompt* (Anti-Hedge v2, Sezione 7.2.4) porta la *verdict accuracy* dell'agente su *file system* da 0,55 a 0,85 su SciFact, a due punti dal RAG avanzato (0,87). Resta aperto se un'ulteriore iterazione del *prompt*, o un passo di verifica dedicato dopo la lettura del contesto, chiuda i due punti percentuali residui. Va verificato anche se il risultato si generalizzi a compiti di *fact-verification* diversi da SciFact.

***Benchmark* multi-dominio.** La validazione esterna richiede studi pilota su domini con vincoli distinti: sanità con *rubric* ontologica sulla terminologia biomedica UMLS [2], finanza con controllo di freschezza temporale, *compliance* con *audit log* immutabile. A questi affianchiamo un confronto *cross-framework* con implementatori indipendenti su LangGraph, AutoGen e CrewAI, che serve a separare l'effetto apprendimento dal principio architetturale.

9.4 Implicazioni per i professionisti

Per chi adotta sistemi agentici in produzione, il lavoro suggerisce quattro raccomandazioni.

1. **Adottare un percorso incrementale**: infrastruttura, prototipazione, validazione, anziché l'implementazione finale diretta. Le criticità del prototipo (Tabella 1.1) emergono quando correggerle costa poco, prima della riscrittura in codice Python. Ciascuna orienta poi una scelta precisa della fase successiva, dal *caching* semantico alla persistenza del contesto utente.

2. **Confinare l'LLM alla pianificazione**, affidando l'esecuzione a codice deterministico verificabile (Sezione 5.5). Il prezzo da mettere in conto sono i 1.366 LOC di orchestrazione misurati nel Capitolo 5, di cui il *Route Executor* è una frazione.

3. **Trattare la prototipazione *low-code* come generatore di evidenze progettuali**, vincolandola a un criterio di uscita esplicito. Il criterio va fissato in anticipo sul numero di criticità strutturali emerse: nel nostro caso l'uscita scatta alle sei criticità della Tabella 1.1, quando il costo del *lock-in* supera il valore delle evidenze raccolte.

4. **Definire *benchmark* specifici per il proprio dominio**. Le metriche aggregate possono nascondere debolezze su *pattern* particolari, come il *routing* sequenziale o le *query* avversarie. Vanno sondate con casi avversari deliberati: *follow-up* multi-turno, *tool failure*, richieste *off-scope*.

9.5 Considerazioni finali

Il principio della separazione *reasoning/execution* si dimostra portabile. Vale in entrambe le transizioni di sostrato, da n8n a LangGraph e da LangGraph a OpenCode, e conserva il proprio valore cambiando soltanto la modalità di *enforcement*, da *per costruzione* a *per aderenza*. L'invariante architetturale resta dunque valido al cambio di tecnologia, mentre la sua garanzia formale dipende dalla piattaforma che lo ospita. Il metodo incrementale è il secondo artefatto trasferibile. Un *team* che adotti sistemi agentici in produzione ricava da questo lavoro una sequenza replicabile, oltre all'architettura descritta.

Restano i limiti che abbiamo dichiarato: la calibrazione su un singolo dominio, la natura longitudinale e non randomizzata dei confronti, il *self-evaluation bias* dei *benchmark* Ragas.

Su quest'ultimo lo studio comparativo esteso (Sezione 7.2) interviene già come validazione incrociata: il giudice indipendente dal generatore e i tre dataset aggiuntivi attenuano la minaccia dichiarata per la Fase 4, senza eliminarla. Le direzioni della Sezione 9.3 affrontano uno per uno i limiti residui.

Entro questi confini il lavoro colloca ciascuna scelta entro le condizioni C_1–C_3 e ne dichiara il prezzo. Con vincoli di auditabilità e sovranità del dato, il determinismo esecutivo copre il proprio costo. Quando conta la velocità di iterazione, l'allentamento del vincolo sposta il rischio sui *prompt* e lo lascia in carico a chi scrive le istruzioni.

Bibliografia

[1] Anthropic. *Model Context Protocol Specification*. 2024. URL: https://modelcontextprotocol.io/ (visitato il giorno 26/02/2026).

[2] Olivier Bodenreider. «The Unified Medical Language System (UMLS): integrating biomedical terminology». In: *Nucleic Acids Research* 32.Database issue (2004), pp. D267–D270. DOI: 10.1093/nar/gkh061.

[3] Jianlv Chen et al. «BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation». In: *arXiv preprint arXiv:2402.03216* (2024).

[4] Chonkie Contributors. *Chonkie: A No-Nonsense Fast, Lightweight Chunking Library*. 2024. URL: https://github.com/chonkie-inc/chonkie (visitato il giorno 15/07/2026).

[5] Samuel Colvin. *Pydantic v2 Documentation*. 2023. URL: https://docs.pydantic.dev/ (visitato il giorno 26/02/2026).

[6] Gordon V. Cormack, Charles L. A. Clarke e Stefan Büttcher. «Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods». In: *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. 2009, pp. 758–759. DOI: 10.1145/1571941.1572114.

[7] CrewAI Inc. *CrewAI: Framework for Orchestrating Role-Playing AI Agents*. 2024. URL: https://crewai.com (visitato il giorno 26/02/2026).

[8] Dify.AI. *Dify: LLMOps Platform*. 2024. URL: https://dify.ai/ (visitato il giorno 26/02/2026).

[9] Shahul Es et al. «RAGAS: Automated Evaluation of Retrieval Augmented Generation». In: *arXiv preprint arXiv:2309.15217* (2023).

[10] Exa Labs. *Exa: Search API for AI*. 2024. URL: https://exa.ai/ (visitato il giorno 09/06/2026).

[11] Richard E. Fikes e Nils J. Nilsson. «STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving». In: *Artificial Intelligence* 2.3–4 (1971), pp. 189–208. DOI: 10.1016/0004-3702(71)90010-5.

[12] Varun Gandhi et al. «GRASP: GRanularity-Aware Search Policy for Agentic RAG». In: *arXiv preprint arXiv:2607.10463* (2026).

[13] Yunfan Gao et al. «Retrieval-Augmented Generation for Large Language Models: A Survey». In: *arXiv preprint arXiv:2312.10997* (2024).

[14] Malik Ghallab, Dana Nau e Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004. ISBN: 978-1-55860-856-6.

[15] GitHub, Inc. *GitHub Copilot Documentation*. 2024. URL: https://docs.github.com/en/copilot (visitato il giorno 19/08/2026).

[16] Kai Greshake et al. «Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection». In: *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec)*. 2023, pp. 79–90. DOI: 10.1145/3605764.3623985.

[17] Alan R. Hevner et al. «Design Science in Information Systems Research». In: *MIS Quarterly* 28.1 (2004), pp. 75–105. DOI: 10.2307/25148625.

[18] Sirui Hong et al. «MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework». In: *arXiv preprint arXiv:2308.00352* (2023).

[19] Jie Huang e Kevin Chen-Chuan Chang. «Towards Reasoning in Large Language Models: A Survey». In: *Findings of the Association for Computational Linguistics (ACL)*. 2023, pp. 1049–1065. DOI: 10.18653/v1/2023.findings-acl.67.

[20] Carlos E. Jimenez et al. «SWE-bench: Can Language Models Resolve Real-World GitHub Issues?» In: *arXiv preprint arXiv:2310.06770* (2024).

[21] Seungone Kim et al. «Prometheus: Inducing Fine-Grained Evaluation Capability in Language Models». In: *Proceedings of the 12th International Conference on Learning Representations (ICLR)*. 2024.

[22] Kubernetes Authors. *Kubernetes Documentation*. 2024. URL: https://kubernetes.io/docs/ (visitato il giorno 26/02/2026).

[23] Woosuk Kwon et al. «Efficient Memory Management for Large Language Model Serving with PagedAttention». In: *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. 2023. DOI: 10.1145/3600006.3613165.

[24] LangChain Inc. *LangGraph: Build Stateful, Multi-Actor Applications with LLMs*. 2024. URL: https://langchain-ai.github.io/langgraph/ (visitato il giorno 26/02/2026).

[25] Langflow Team. *Langflow: Visual Design for LLM Applications*. 2024. URL: https://www.langflow.org/ (visitato il giorno 26/02/2026).

[26] Patrick Lewis et al. «Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 33. 2020, pp. 9459–9474.

[27] Guohao Li et al. «CAMEL: Communicative Agents for "Mind" Exploration of Large Scale Language Model Society». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 36. 2023.

[28] Zhuofeng Li et al. «Beyond Semantic Similarity: Rethinking Retrieval for Agentic Search via Direct Corpus Interaction». In: *arXiv preprint arXiv:2605.05242* (2026).

[29] Nelson F. Liu et al. «Lost in the Middle: How Language Models Use Long Contexts». In: *Transactions of the Association for Computational Linguistics* 12 (2024), pp. 157–173.

[30] Yang Liu et al. «G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment». In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2023, pp. 2511–2522. DOI: 10.18653/v1/2023.emnlp-main.153.

[31] Yu. A. Malkov e D. A. Yashunin. «Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.3 (2018), pp. 512–523. DOI: 10.1109/TPAMI.2018.2889473.

[32] Bertrand Meyer. «Applying "Design by Contract"». In: *Computer* 25.10 (1992), pp. 40–51. DOI: 10.1109/2.161279.

[33] n8n GmbH. *n8n – Workflow Automation Tool*. 2024. URL: https://n8n.io/ (visitato il giorno 26/02/2026).

[34] Rodrigo Nogueira e Kyunghyun Cho. «Passage Re-ranking with BERT». In: *arXiv preprint arXiv:1901.04085* (2019).

[35] Open WebUI Contributors. *Open WebUI: User Interface for LLMs*. 2024. URL: https://github.com/open-webui/open-webui (visitato il giorno 26/02/2026).

[36] OpenCode Team. *OpenCode: Declarative AI Agent Framework*. 2024. URL: https://opencode.ai/ (visitato il giorno 09/06/2026).

[37] Joon Sung Park et al. «Generative Agents: Interactive Simulacra of Human Behavior». In: *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST)*. 2023.

[38] Shishir G. Patil et al. «Gorilla: Large Language Model Connected with Massive APIs». In: *arXiv preprint arXiv:2305.15334* (2023).

[39] Ken Peffers et al. «A Design Science Research Methodology for Information Systems Research». In: *Journal of Management Information Systems* 24.3 (2007), pp. 45–77. DOI: 10.2753/MIS0742-1222240302.

[40] pgvector Contributors. *pgvector: Open-source vector similarity search for Postgres*. 2024. URL: https://github.com/pgvector/pgvector (visitato il giorno 26/02/2026).

[41] PostgreSQL Global Development Group. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. 2024. URL: https://www.postgresql.org/docs/ (visitato il giorno 26/02/2026).

[42] Qdrant Team. *Qdrant: Vector Database for the Next Generation of AI*. 2024. URL: https://qdrant.tech/ (visitato il giorno 09/06/2026).

[43] Qwen Team. «Qwen3 Technical Report». In: *arXiv preprint arXiv:2505.09388* (2025).

[44] Qwen Team, Alibaba Cloud. «Qwen2.5-Coder Technical Report». In: *arXiv preprint arXiv:2409.12186* (2024).

[45] Zackary Rackauckas. «RAG-Fusion: A New Take on Retrieval-Augmented Generation». In: *International Journal on Natural Language Computing* 13.1 (2024), pp. 37–47. DOI: 10.5121/ijnlc.2024.13103.

[46] Sebastián Ramírez. *FastAPI*. 2018. URL: https://fastapi.tiangolo.com/ (visitato il giorno 26/02/2026).

[47] Stephen Robertson e Hugo Zaragoza. «The Probabilistic Relevance Framework: BM25 and Beyond». In: *Foundations and Trends in Information Retrieval* 3.4 (2009), pp. 333–389. DOI: 10.1561/1500000019.

[48] Timo Schick et al. «Toolformer: Language Models Can Teach Themselves to Use Tools». In: *Advances in Neural Information Processing Systems (NeurIPS)* 36 (2023).

[49] Noah Shinn et al. «Reflexion: Language Agents with Verbal Reinforcement Learning». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 36. 2023.

[50] Theodore R. Sumers et al. «Cognitive Architectures for Language Agents». In: *Transactions on Machine Learning Research* (2024).

[51] Yixuan Tang e Yi Yang. «MultiHop-RAG: Benchmarking Retrieval-Augmented Generation for Multi-Hop Queries». In: *Proceedings of the 1st Conference on Language Modeling (COLM)*. 2024.

[52] Pat Verga et al. «Replacing Judges with Juries: Evaluating LLM Generations with a Panel of Diverse Models». In: *arXiv preprint arXiv:2404.18796* (2024).

[53] vLLM Project Contributors. *vLLM Documentation*. 2024. URL: https://docs.vllm.ai/ (visitato il giorno 26/02/2026).

[54] David Wadden et al. «Fact or Fiction: Verifying Scientific Claims». In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2020, pp. 7534–7550. DOI: 10.18653/v1/2020.emnlp-main.609.

[55] Lei Wang et al. «A Survey on Large Language Model Based Autonomous Agents». In: *Frontiers of Computer Science* 18.6 (2024), p. 186345. DOI: 10.1007/s11704-024-40231-1.

[56] Lei Wang et al. «Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models». In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*. 2023, pp. 2609–2634. DOI: 10.18653/v1/2023.acl-long.147.

[57] Zihao Wang et al. «Describe, Explain, Plan and Select: Interactive Planning with Large Language Models Enables Open-World Multi-Task Agents». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 36. 2023.

[58] Jason Wei et al. «Chain-of-Thought Prompting Elicits Reasoning in Large Language Models». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 35. 2022, pp. 24824–24837.

[59] Michael Wooldridge e Nicholas R. Jennings. «Intelligent Agents: Theory and Practice». In: *The Knowledge Engineering Review* 10.2 (1995), pp. 115–152. DOI: 10.1017/S0269888900008122.

[60] Qingyun Wu et al. «AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation». In: *arXiv preprint arXiv:2308.08155* (2023).

[61] Zhiheng Xi et al. «The Rise and Potential of Large Language Model Based Agents: A Survey». In: *arXiv preprint arXiv:2309.07864* (2023).

[62] John Yang et al. «SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering». In: *arXiv preprint arXiv:2405.15793* (2024).

[63] Zhilin Yang et al. «HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering». In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2018, pp. 2369–2380. DOI: 10.18653/v1/D18-1259.

[64] Shunyu Yao et al. «ReAct: Synergizing Reasoning and Acting in Language Models». In: *International Conference on Learning Representations (ICLR)*. 2023.

[65] Lianmin Zheng et al. «Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 36. 2023.