Politecnico di Torino

Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Informatica

A.a. 2025/2026

Sessione di laurea Marzo 2026

Assisted Selling: un agente per la selezione e configurazione di macchine agricole

Relatori:

Luigi De Russis

Oscar Ioppolo

Alessandro Vidotto

Candidato:

Riccardo Simeone

Sommario

Nel contesto del commercio digitale moderno, i configuratori di prodotto rappresentano uno strumento ampiamente diffuso per la personalizzazione di beni complessi. Tali strumenti, tuttavia, presuppongono la conoscenza preliminare dell'offerta e delle caratteristiche tecniche dei prodotti da parte dell'utente, rendendo spesso difficile il loro utilizzo a chi non è familiare con il dominio applicativo. Il presente lavoro di tesi si inserisce in questo contesto con lo scopo di proporre un metodo alternativo all'interazione tradizionale con i configuratori di prodotto. In particolare, l'attenzione è rivolta alla fase iniziale del processo di configurazione, per la quale viene progettato e sviluppato un assistente virtuale in grado di sostituire la navigazione manuale del catalogo con un'interazione di tipo conversazionale, nella quale l'utente può esprimere le proprie esigenze in linguaggio naturale. L'assistente consente all'utente di descrivere il proprio contesto operativo e i requisiti di utilizzo relativi al prodotto agricolo da acquistare e, sulla base di tali informazioni, suggerisce una selezione di modelli potenzialmente idonei che possono successivamente essere configurati in modo dettagliato attraverso il configuratore di prodotto esistente. Il questo modo, lo sforzo cognitivo associato alla scelta del prodotto di partenza viene notevolmente ridotto, migliorando l'efficacia, la precisione e l'accessibilità dell'intero processo. Il lavoro intende quindi evidenziare come un approccio di Assisted Selling, basato su tecniche di Intelligenza Artificiale, possa rappresentare un'evoluzione significativa dei configuratori di prodotto tradizionali nel supporto alla selezione di prodotti complessi.

Ringraziamenti

Grazie ai miei genitori e a mio fratello, che sono stati e sono tutt'ora il mio punto di riferimento, che mi hanno spinto e sostenuto costantemente durante tutto il percorso senza mai pretendere nulla.

Grazie ai miei nonni, agli zii e ai cugini, vicini e lontani, per il supporto e l'affetto che ho sempre sentito, sia a pochi chilometri di distanza sia a migliaia.

Grazie a chi c'è stato e a chi non c'è più, in particolare grazie a zio Luigi, che anche se non è più presente fisicamente, è stata una delle ragioni per la quale ho iniziato questo percorso e di cui spero sia orgoglioso.

Grazie a Davide e Francesco, grazie a Lorenzo, Stefano ed Andrea, che hanno dovuto sopportare un coinquilino a volte un po' scontroso, testardo e non sempre di buon umore, ma che ha sempre cercato di essere un buon amico e rendere la convivenza qualcosa di più di un semplice vivere sotto lo stesso tetto.

Grazie al gruppo Piazzetta, con il quale ho condiviso anni di risate, meme e prese in giro, che hanno reso più leggeri i momenti più complicati, tra partite interminabili a Lupus e Gin Lemon e Spritz a non finire.

Grazie al mio gruppo di giù, con il quale condivido un'amicizia da anni, per farmi sentire parte integrante nonostante io sia lontano, e che ogni volta che ci rivediamo mi fa sentire come se fossimo vicini tutti i giorni.

Desidero ringraziare Coolshop e tutti i Coolpeople, in particolare Jessica, Oscar ed Alessandro, per avermi accolto all'interno di un ambiente familiare e caloroso con grande disponibilità e cordialità, offrendomi un'esperienza che mi ha arricchito molto sia sul piano umano sia su quello formativo e professionale.

Ed infine grazie al prof. De Russis, per la pazienza avuta nei miei confronti, per i consigli condivisi durante tutto il corso di quest'ultima tappa, e per l'enorme disponibilità che dimostra costantemente verso noi studenti.

Ad maiora!

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

Negli ultimi anni il commercio digitale ha subito un'evoluzione notevole, dovuta non solo al progresso tecnologico ma anche ad un cambiamento nelle aspettative degli utenti, finendo per ricoprire un ruolo fondamentale nei processi di vendita di beni complessi. I clienti sono sempre più spesso alla ricerca di soluzioni **personalizzate** e di strumenti **semplici da utilizzare**, e i configuratori di prodotto in questo senso rappresentano una soluzione ampiamente adottata per consentire agli utenti di adattare il prodotto alle proprie esigenze.

Con l'aumento della complessità dei cataloghi e della varietà delle configurazioni disponibili, tuttavia, l'interazione con questi strumenti è diventata sempre **più impegnativa**. I configuratori tradizionali presuppongono che l'utente sappia già orientarsi autonomamente: deve navigare un catalogo strutturato, capire le differenze tra le alternative e scegliere il prodotto giusto da configurare. Nei casi in cui l'utente sia inesperto o abbia una scarsa conoscenza tecnica del dominio applicativo, questo processo potrebbe risultare problematico e poco intuitivo.

A partire da queste considerazioni, il presente lavoro di tesi, sviluppato insieme a *Coolshop Srl*, si propone di affrontare il problema introducendo un approccio alternativo di interazione con i configuratori, con lo scopo di rendere più naturale la fase iniziale di scelta del prodotto da configurare.

1.1 Background dei configuratori tradizionali

I **configuratori di prodotto** sono strumenti software progettati per supportare la personalizzazione di prodotti configurabili, spesso caratterizzati da un'elevata complessità tecnica. Sono utili, ad esempio, in presenza di prodotti che vengono assemblati a partire da singoli componenti, e consentono all'utente di costruire una rappresentazione del bene **coerente** con i propri bisogni e le proprie preferenze, automatizzandone il processo.

Dal punto di vista dell'utente, i configuratori svolgono innanzitutto una funzione orientativa: aiutano a comprendere le caratteristiche del prodotto e, man mano che una selezione viene effettuata, mostrano in modo guidato le sole scelte compatibili, evitando che l'utente generi combinazioni fuori catalogo. Consentono poi di esplorare liberamente più configurazioni senza vincoli d'acquisto, spesso proponendo simulazioni di prezzo, visualizzazioni grafiche dinamiche del prodotto e feedback immediati sulle scelte fatte. In questo senso, più che semplici strumenti di acquisto, diventano il principale punto di contatto tra il cliente e il prodotto, influenzando in modo significativo la percezione complessiva dell'esperienza d'acquisto.

Sul piano aziendale i benefici riguardano sia il controllo dell'offerta sia la riduzione dei costi operativi, definendo regole di configurazione direttamente nel sistema quali vincoli di compatibilità, obbligatorietà o esclusioni, impedendo la selezione di combinazioni non realizzabili o non commercializzabili e abbassando così il rischio di resi e costi di rilavorazione. In più permettono di raccogliere informazioni sul comportamento e sulle preferenze degli utenti, che possono essere utilizzate per migliorare l'offerta e ottimizzare i processi interni, e supportano la generazione in tempo reale di preventivi affidabili riducendo il **time-to-market**[1] e velocizzando l'intero processo di offertazione.

Tuttavia, questi strumenti si basano su un modello che implica una **selezione preliminare** del prodotto da configurare. Il processo tipico si articola in **due fasi** distinte:

1. **Selezione del modello**: l'utente naviga il catalogo attraverso una struttura spesso gerarchica oppure tramite filtri parametrizzati, alla ricerca del prodotto base da personalizzare;

2. **Personalizzazione**: una volta scelto il modello, il configuratore guida l'utente nella selezione di caratteristiche tecniche e nella scelta di varianti componentistiche, con la possibilità di generare anche un preventivo.

Per quanto riguarda la fase di **personalizzazione**, questi strumenti sono in grado di ottimizzarla garantendo coerenza tecnica tramite vincoli di compatibilità, esclusioni reciproche e controlli automatici, che impediscono combinazioni non valide e riducono errori e incongruenze.

Il problema nasce nella **fase iniziale di selezione**. Entrambe le modalità di navigazione comunemente adottate - gerarchica (per macro-categorie) e parametrica (per attributi tecnici) - presuppongono che l'utente abbia già una conoscenza almeno parziale della struttura dell'offerta e dei parametri rilevanti. Si crea così un disallineamento tra il modello mentale dell'utente, che ragiona in termini di

[1] Il tempo tra l'ideazione di un prodotto e la sua effettiva commercializzazione.

problema da risolvere, e la struttura dell'offerta, organizzata secondo criteri tecnico-commerciali. Questo disallineamento e le sue implicazioni saranno analizzati in dettaglio nel Capitolo 2.

1.2 *Assisted Selling*: supporto alla selezione

La soluzione proposta, denominata **Assisted Selling**, introduce un'interazione di tipo **conversazionale** in cui l'utente può esprimere le proprie esigenze in linguaggio naturale, anziché dover navigare autonomamente il catalogo. La conversazione è per sua natura **adattiva**: ogni risposta ridefinisce le domande successive, avvicinandosi pian piano alla soluzione senza richiedere all'utente un vocabolario tecnico che non possiede. È, in sostanza, la trasposizione digitale di ciò che il concessionario fa nella vendita tradizionale. Il sistema interpreta i bisogni espressi e li traduce in una selezione di prodotti candidati, coerenti con la struttura dell'offerta, che possono successivamente essere configurati tramite il sistema CPQ esistente.

Il modulo si configura quindi come un componente complementare, integrato all'interno del prodotto *CoolPIM* (Product Information Management) di *Coolshop*, e interviene esclusivamente nella fase preliminare di selezione, senza modificare le logiche di configurazione e pricing già presenti.

1.3 Obiettivo

L'obiettivo della tesi è progettare e sviluppare un **assistente virtuale** basato su tecniche di elaborazione del linguaggio naturale, capace di supportare l'utente nella selezione del prodotto candidato alla configurazione. In particolare, il sistema deve consentire all'utente di esprimere i propri bisogni in linguaggio naturale, **riducendo il disallineamento** tra modello mentale dell'utente e struttura tecnica del catalogo, con lo scopo di generare un insieme di prodotti configurabili coerenti con le esigenze operative espresse. Il sistema deve inoltre **integrarsi** con la soluzione CPQ esistente, demandando ad essa la personalizzazione tecnica e la definizione del preventivo.

1.4 Struttura della tesi

La tesi è organizzata in sette capitoli, in modo da favorire la comprensione del problema e descrivere la soluzione adottata per risolverlo. Nello specifico:

- **Capitolo 1**: introduce il contesto del commercio digitale, i configuratori di prodotto e i loro limiti, la soluzione proposta e gli obiettivi della tesi;

- **Capitolo 2**: descrive il settore della meccanizzazione agricola come dominio applicativo e analizza il configuratore oggetto di studio, evidenziandone le criticità;
- **Capitolo 3**: presenta l'analisi del contesto d'uso, dei profili utente e definisce i requisiti funzionali e non funzionali del sistema;
- **Capitolo 4**: descrive l'architettura generale, i sistemi coinvolti, il modello dati e le scelte progettuali;
- **Capitolo 5**: descrive e motiva le scelte implementative adottate per costruire la soluzione progettata;
- **Capitolo 6**: valuta il modulo attraverso sessioni osservative sul catalogo reale, analizzando qualità delle raccomandazioni, prestazioni e soddisfacimento dei requisiti;
- **Capitolo 7**: riassume il lavoro svolto, evidenziando dei limiti ancora presenti ed eventuali sviluppi futuri.

Capitolo 2

Contesto applicativo

In questo capitolo si descrive il **contesto applicativo** all'interno del quale si colloca il lavoro di tesi, con l'obiettivo di fornire una visione chiara del **dominio di riferimento** e del sistema reale su cui è stata sperimentata la soluzione proposta. Si tratterà, in particolar modo, il settore delle **macchine agricole** e le caratteristiche che rendono il processo di selezione e configurazione del prodotto particolarmente complesso, per poi analizzare il funzionamento del configuratore oggetto di studio e le principali **criticità** emerse.

2.1 Il settore della meccanizzazione agricola

Il settore della meccanizzazione agricola è un esempio emblematico di dominio in cui i prodotti combinano **elevata complessità tecnica** e **forte grado di personalizzazione**. Trattori e macchine operatrici sono strumenti di lavoro che devono adattarsi a contesti operativi molto diversi fra loro, il che rende il processo di scelta tutt'altro che banale.

La scelta di una macchina agricola dipende innanzitutto dal **contesto in cui verrà utilizzata**, ed in particolare dall'**attività da svolgere** (lavorazione del terreno, semina, raccolta, trasporto, movimentazione) in quanto operazioni diverse richiedono macchine con caratteristiche molto diverse, dalle **condizioni del terreno** (compatto, argilloso, sabbioso) e da quelle **ambientali** (pioggia, caldo estremo, lavoro notturno), dalla **modalità d'impiego** (se solo lavoro nei campi o anche circolazione su strada, se semovente o trainata). A questi aspetti operativi si affiancano considerazioni più **tecniche** come le **prestazioni** richieste (potenza, capacità di sollevamento, produttività), la **compatibilità** con attrezzature specifiche (aratri, coltivatori, seminatrici, caricatori frontali), il **comfort dell'operatore** (cabinato, sedili pneumatici, climatizzazione), il rispetto di **vincoli normativi** come i limiti sulle emissioni, e i costi di esercizio e manutenzione.

Questa diversità di fattori è una delle cause principali della complessità dei cataloghi prodotto, ampi e strutturati, composti da diverse famiglie di macchine e in una vasta gamma di varianti e allestimenti. Ma non è la sola. Tale complessità è dovuta anche alla **struttura intrinseca** del prodotto stesso. Infatti, le macchine agricole moderne possono essere considerate **sistemi meccatronici avanzati**, in cui componenti meccanici, idraulici, elettronici e software lavorano in modo coordinato. La scelta di una soluzione può influenzarne un'altra e rende più complicata la **verifica della compatibilità** tra i vari sistemi.

Il settore è inoltre al centro di un progressivo **processo di digitalizzazione** che ha portato all'introduzione di nuove funzionalità e strumentazioni, tra cui sistemi di agricoltura di precisione, dispositivi di telemetria, soluzioni di guida assistita e piattaforme di gestione dei dati. Queste tecnologie ampliano il numero di combinazioni possibili e contribuiscono all'aumento della complessità decisionale.

C'è poi un aspetto economico che non va trascurato. L'acquisto di una macchina agricola è un investimento ad elevato impatto economico e operativo, con un orizzonte temporale pluriennale. Una scelta non adeguata può incidere sulla produttività e sui costi di gestione per anni. Nel contesto tradizionale, questa complessità è gestita dalla figura del **concessionario** o consulente tecnico, che fa da mediatore tra le esigenze operative del cliente e l'offerta commerciale disponibile. Il concessionario interpreta il bisogno reale dell'utente, lo traduce in parametri tecnici e guida il cliente verso la soluzione più adatta, basandosi sulla propria esperienza e sulla conoscenza diretta del catalogo. Nel contesto digitale, però, questa funzione di orientamento è difficile da replicare. La ragione è che la mediazione del concessionario è per sua natura **adattiva**: parte da ciò che il cliente dice, formula domande sulla base delle risposte ricevute e restringe progressivamente le alternative. Un'interfaccia a navigazione classica con menu, filtri e schede prodotto non ha questa capacità, presenta l'offerta nella sua interezza e lascia al cliente la responsabilità di orientarsi, senza un livello intermedio che interpreti il bisogno e lo traduca in una direzione concreta all'interno del catalogo.

Come si traduce tutto questo nell'interazione con un sistema reale? La sezione successiva lo mostra analizzando il configuratore Case IH.

2.2 Il configuratore del brand Case IH

Il lavoro svolto si è concentrato sull'analisi di un configuratore di prodotto del marchio **Case IH**, inserito all'interno di una soluzione di tipo CPQ (*Configure*, *Price*, *Quote*) che non si limita alla configurazione tecnica del prodotto ma integra anche la gestione delle *regole di compatibilità*, il *calcolo del prezzo* e la *generazione del preventivo*.

Case IH è uno dei marchi di riferimento mondiale nel settore della meccanizzazione agricola [1]. Il brand nasce nel 1985 dalla fusione tra *J.I. Case Company*, storico produttore americano di macchine agricole fondato nel 1842, e la divisione agricola di *International Harvester Company*, azienda la cui tradizione nella produzione di trattori e mietitrebbie risale alla fine dell'Ottocento [2]. Dalla loro unione è nato un marchio che oggi opera in oltre 160 paesi con una gamma di prodotti che copre l'intero ciclo delle attività agricole: trattori per lavori di campo e operazioni di caricamento, mietitrebbie con testate intercambiabili, imballatrici, caricatori telescopici e caricatori frontali. Il brand è parte del gruppo **CNH Industrial** [3], uno dei principali gruppi industriali mondiali nel settore dei veicoli e delle soluzioni per l'agricoltura e la costruzione.

Nel contesto della trasformazione digitale dei processi di vendita, Case IH mette a disposizione di concessionari e clienti finali strumenti di configurazione online accessibili dal portale ufficiale del brand [4]. La soluzione CPQ analizzata in questo lavoro è integrata in quel portale: il *Configure* gestisce la selezione e la personalizzazione del modello a partire dalla struttura gerarchica del catalogo; il *Price* calcola il valore della configurazione in tempo reale al variare delle opzioni selezionate; il *Quote* produce la documentazione commerciale da trasmettere alla rete di vendita o condividere con il cliente finale.

Il caso studiato fa riferimento alla *Product Offering* del brand nel mercato UK; per coerenza con il contesto analizzato, nel seguito verranno adottate la struttura e la nomenclatura ufficiali del catalogo in questione.

2.2.1 Struttura dell'offerta e processo di selezione

La proposta di *Case IH* è organizzata in famiglie di prodotto ben definite, che rappresentano il livello più alto di classificazione nel catalogo. Nello specifico, l'offerta comprende:

- **Tractors**, destinati ad attività agricole, coltivazione, paesaggistica e caricamento, con attenzione a potenza ed efficienza dei consumi;
- **Combines and Headers**, che includono macchine da raccolta e diverse tipologie di testate per operare con rapidità;
- **Balers**, dedicate all'imballaggio del foraggio;
- **Telescopic Handlers**, ossia caricatori telescopici progettati per garantire comfort operativo, tempi di caricamento ridotti e pronta risposta del veicolo;
- **Front Loaders**, una gamma di caricatori frontali pensati per velocizzare le attività in campo.

Le *famiglie* costituiscono il primo livello di una navigazione strutturata su più passaggi, che conduce alla scelta del prodotto più idoneo. In genere raggruppano macchine con funzioni primarie comuni e caratteristiche simili, generando una suddivisione che fornisce un primo orientamento all'utente e gli permette di individuare l'area dell'offerta più coerente con le proprie esigenze.

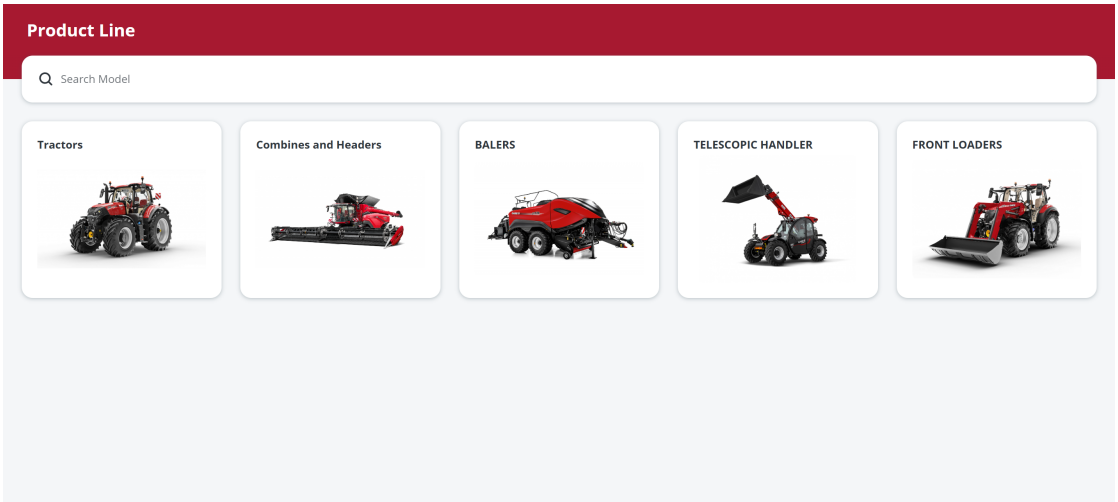

Figura 2.1: Screenshot della struttura delle famiglie di prodotto nel catalogo UK di Case IH. Fonte: [4]

Una volta selezionata la famiglia di interesse, il cliente si trova di fronte alla scelta di una **Product Series** (o **linea** di prodotto). Le *linee* introducono **un livello di dettaglio intermedio** e organizzano i prodotti in gruppi omogenei per prestazioni, dimensioni e dotazioni disponibili. In questa fase il numero di alternative è più **contenuto**, ma l'utente è comunque chiamato a confrontare caratteristiche tecniche che non sempre si rivelano immediatamente **intuitive** (Figura 2.2).

Infine, all'interno della *serie* selezionata, l'utente giunge finalmente alla scelta del **modello** configurabile, che rappresenta il punto di ingresso alla fase di personalizzazione tecnica ed è particolarmente rilevante perché determina il passaggio da una logica di esplorazione generale ad una fase di decisione più concreta: non si tratta più solo di confrontare categorie o linee, ma di individuare una macchina base coerente con il proprio contesto operativo (Figura 2.3).

La complessità dell'offerta, distribuita lungo questi tre passaggi successivi, evita che l'utente si trovi fin da subito di fronte a un numero eccessivo di alternative tecniche. Il processo di selezione è rappresentato come un percorso guidato che prepara l'accesso alla fase di *customizzazione* vera e propria.

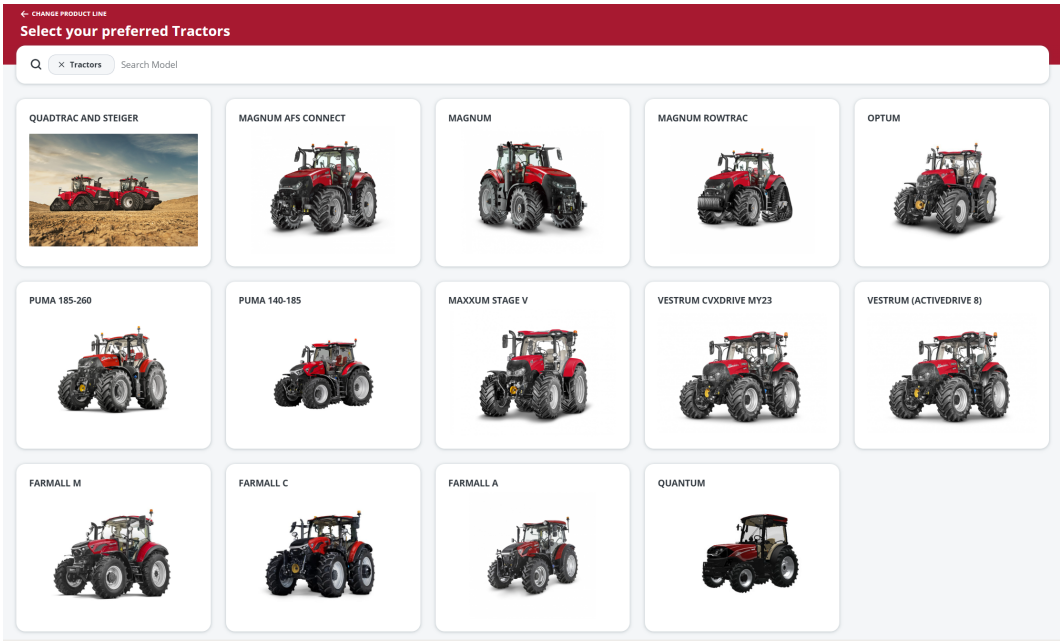

Figura 2.2: Screenshot delle *Product Series* all'interno della famiglia *Tractors*. Fonte: [4]

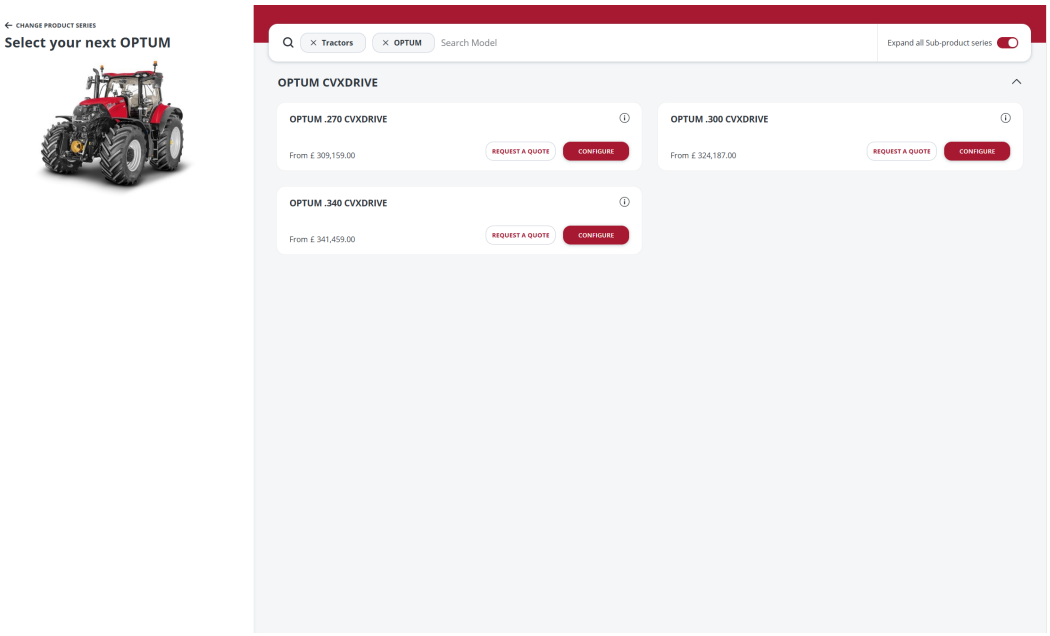

Figura 2.3: Screenshot dei modelli configurabili all'interno della serie *Optum* della famiglia *Tractors*. Fonte: [4]

2.2.2 Considerazioni e criticità

Dal punto di vista dell'interazione utente emergono alcune **criticità strutturali**. Il processo appena descritto (famiglia → serie → modello) richiede che il cliente affronti tre livelli di scelta successivi, ciascuno dei quali presuppone la capacità di distinguere tra alternative caratterizzate da parametri tecnici specifici: dalla funzione primaria della macchina (a livello di famiglia), alle prestazioni e dotazioni (a livello di serie), fino alle specifiche del singolo modello configurabile.

Il disallineamento tra modello mentale dell'utente e struttura dell'offerta, introdotto nel Capitolo 1, qui nel caso di Case IH diventa particolarmente evidente. Un agricoltore che cerca una macchina per la lavorazione di terreni argillosi su superfici medio-grandi, ad esempio, ragiona in termini di attività, condizioni operative e risultati attesi. Il configuratore, invece, gli chiede come primo passo di scegliere tra le cinque famiglie di prodotto (Tractors, Combines, Balers, Telescopic Handlers, Front Loaders), una decisione che presuppone già una conoscenza della classificazione commerciale del brand.

Nella pratica, soprattutto nel contesto della meccanizzazione agricola, questo scenario risulta spesso **problematico**. La difficoltà nasce da diversi fattori. Il primo è **terminologico**: la struttura del catalogo utilizza nomenclature specifiche (es. "Optum", "Puma", "CVXDRIVE") che possono risultare oscure o poco intuitive per chi non è esperto del settore. Il secondo è **strutturale**: la presenza di numerose famiglie, serie, modelli e varianti genera un elevato numero di alternative, aumentando il rischio di confusione e di scelte poco consapevoli. Il terzo, forse il più rilevante, è l'**assenza di mappatura diretta tra bisogni e prodotti**: il sistema non offre un meccanismo esplicito per tradurre un'esigenza operativa (es. "lavoro su terreni fangosi con attrezzi pesanti") in una gamma o modello specifico, lasciando all'utente il compito di interpretare e navigare il catalogo. Un quarto fattore, più sottile, riguarda i **vincoli di compatibilità e personalizzazione**: la scelta di un modello influenza la disponibilità di optional e la compatibilità tra sistemi, ma queste regole non sono sempre esplicite nella fase di selezione iniziale. L'utente seleziona senza sapere cosa quel modello gli consentirà o precluderà.

Il risultato è che la selezione della soluzione di partenza diventa un'attività complessa e potenzialmente soggetta a **errori**, **incertezze** o addirittura abbandono del processo. L'esperienza utente può risultare frustrante, soprattutto per chi non ha una conoscenza approfondita del catalogo o del dominio tecnico.

In sostanza, il configuratore di Case IH esegue correttamente il proprio compito nella fase di personalizzazione, ma non offre un supporto adeguato nella fase di orientamento iniziale. È uno spazio di miglioramento che vale la pena affrontare, e il capitolo successivo traduce questa lacuna in requisiti e obiettivi concreti.

Capitolo 3

Analisi del contesto e definizione dei requisiti

Questo capitolo descrive il processo di analisi che ha guidato la definizione degli obiettivi e dei requisiti del modulo di *Assisted Selling*. L'obiettivo è tradurre le criticità emerse dallo studio del configuratore reale e del contesto applicativo in un insieme coerente di requisiti **funzionali** e **non funzionali**, che costituiranno la base per le scelte architetturali e progettuali illustrate nel capitolo successivo.

3.1 Metodo di analisi

La definizione di obiettivi e requisiti è stata condotta attraverso un processo di analisi progressiva, raffinando man mano le ipotesi progettuali con lo scopo di garantire coerenza tra esigenze degli utenti, vincoli tecnici e contesto applicativo. Nello specifico, si è lavorato su tre fronti. In primo luogo, è stato analizzato il **sistema esistente**, osservando il flusso di selezione del configuratore Case IH e ricostruendo i passaggi che conducono dalla scelta della famiglia di prodotto fino all'individuazione del modello da configurare. Questa fase ha permesso di evidenziare i punti in cui l'utente è chiamato a prendere decisioni tecniche senza un supporto esplicito. In secondo luogo, è stato analizzato il **contesto tecnico**, ossia l'insieme dei vincoli architetturali e funzionali derivanti dall'integrazione con la soluzione CPQ esistente e con l'infrastruttura dati disponibile in CoolPIM. In particolare, è emersa la necessità di progettare un modulo complementare al configuratore, capace di operare senza modificarne le logiche di configurazione, compatibilità e pricing. Infine, è stato analizzato il **contesto d'uso**, considerando sia le caratteristiche del dominio della meccanizzazione agricola sia il modo in cui utenti con diversi livelli di esperienza affrontano la fase iniziale di selezione del prodotto.

A partire da questi elementi d'analisi, il primo passo ha presupposto l'identificazione dei **problemi** riscontrati nella fase di scelta del prodotto da configurare (sulla base delle criticità emerse nel Capitolo 2) e successivamente la definizione degli **obiettivi** del sistema come risposta diretta ai problemi osservati e la derivazione dei **requisiti**, funzionali e non, necessari a rendere gli obiettivi verificabili ed implementabili.

3.2 Analisi del contesto d'uso e degli utenti

L'analisi del contesto d'uso ha avuto come obiettivo la comprensione del processo decisionale che precede la configurazione tecnica del prodotto. Come discusso nel Capitolo 2, nel settore della meccanizzazione agricola la complessità non riguarda soltanto la configurazione del bene, ma inizia già nella fase di individuazione del modello di partenza.

Contesto d'uso

L'interazione con il configuratore può avvenire in diversi scenari. Il più frequente è la **consultazione autonoma** tramite piattaforma web, tipicamente nella fase esplorativa in cui l'utente valuta l'offerta disponibile. Un secondo scenario riguarda il **supporto da parte di un dealer** o consulente commerciale, che utilizza il configuratore come strumento operativo durante il confronto con il cliente. Il terzo corrisponde alla **fase preliminare di richiesta preventivo**, in cui l'utente ha già un'idea generale del prodotto desiderato e vuole arrivare ad una stima economica.

In tutti e tre i casi l'interazione avviene in condizioni caratterizzate da tempi decisionali contenuti, confronto fra più alternative, presenza di vincoli sia economici sia operativi e, soprattutto, la difficoltà di collegare il bisogno reale alla classificazione tecnica del catalogo.. Ne consegue che il problema non riguarda solo la disponibilità di informazioni, ma anche il modo in cui queste informazioni sono organizzate e rese accessibili.

Profili utente

Il modulo si rivolge in modo specifico agli utenti che non riescono ad orientarsi in autonomia nel catalogo, non perché manchino di esperienza nel proprio lavoro, ma perché la struttura dell'offerta non è pensata per chi parte da un bisogno operativo o da un punto di riferimento concreto. In particolare, all'interno di questo gruppo, si è potuto distinguere tre profili utente in base al proprio punto di partenza. Il primo profilo è il profilo più comune e quello che lo scenario d'uso descritto di seguito rappresenta, ovvero **chi conosce le proprie esigenze operative ma non il catalogo**, un utente che sa cosa deve fare (tipo di terreno, attività, attrezzature

in dotazione) ma non sa come muoversi all'interno del catalogo. Un secondo profilo può **avere una macchina di riferimento**, un utente che conosce già un modello (proprio o di un concorrente) e lo usa come punto di riferimento: *"cerco qualcosa di simile al John Deere 6120M"*. Il terzo è il profilo di **chi ha le idee ancora vaghe**, un utente che ha un bisogno generico ma non lo ha ancora definito con chiarezza.

Per gli utenti esperti, il cui modello mentale è generalmente allineato alla classificazione tecnico-commerciale dell'offerta, la navigazione gerarchica del catalogo è coerente con il proprio modo di ragionare. La soluzione proposta non si rivolge primariamente a loro, nonostante possa offrire un supporto anche in questo contesto.

Analisi del processo decisionale

Il profilo più comune, e quello che genera le implicazioni progettuali più rilevanti, è il primo, ovvero chi ha chiaro il proprio contesto operativo ma non sa come tradurlo in una scelta di catalogo. Il suo processo decisionale può essere schematizzato come segue:

1. definizione del **bisogno operativo** (quale attività svolgere, in quali condizioni);

2. identificazione delle **caratteristiche funzionali** da ricercare;

3. ricerca di una soluzione coerente nel catalogo disponibile.

Questo processo non coincide con il flusso imposto dal configuratore, che richiede come primo passo la selezione di una famiglia di prodotto (cfr. Capitolo 2). Di seguito si presenta uno scenario d'uso rappresentativo per concretizzare le implicazioni di questo divario.

Scenario d'uso: agricoltore autonomo. Marco è un agricoltore con 15 anni di esperienza nella coltivazione di cereali su terreni collinari argillosi, per un'estensione di circa 50 ettari. Possiede già un aratro reversibile e una seminatrice, e necessita di un nuovo trattore compatibile con queste attrezzature. Conosce bene le proprie esigenze operative ma non ha familiarità con il catalogo Case IH né con la terminologia commerciale delle serie e dei modelli. Accedendo al configuratore, si trova a dover scegliere tra cinque famiglie di prodotto senza un criterio chiaro per orientarsi. Lo scenario è rappresentativo del primo profilo: Marco sa *cosa* deve fare ma non sa *quale prodotto* selezionare. Il configuratore non gli offre un percorso che parta dal bisogno operativo, costringendolo a interpretare da solo la struttura del catalogo.

Implicazioni per la progettazione

Lo scenario rimarca il fatto che le informazioni sono organizzate secondo una logica che non corrisponde al modo in cui l'utente ragiona. Marco sa cosa deve fare; il catalogo sa cosa offre; ma i due non parlano la stessa lingua, e nessuno traduce. Da questa osservazione discendono tre indicazioni progettuali. La prima è che il sistema deve accettare come input il linguaggio dell'utente, non quello del catalogo: se si chiede a Marco di scegliere tra «Optum», «Puma» e «Maxxum», il problema si ripropone identico. La seconda è che la traduzione tra i due linguaggi deve essere compito del sistema, non dell'utente: è il sistema a dover ricondurre un contesto operativo a una selezione di prodotti, non l'utente a dover imparare la tassonomia del brand. La terza, conseguenza delle prime due, è che il sistema deve guidare attivamente la fase di orientamento iniziale, riducendo il numero di decisioni che l'utente è chiamato a prendere in mancanza di informazioni sufficienti. Queste indicazioni guideranno la definizione degli obiettivi e dei requisiti nelle sezioni successive.

3.3 Obiettivi del sistema

Gli obiettivi generali introdotti nel Capitolo 1 vengono qui articolati alla luce dell'analisi del profilo utente e dello scenario d'uso appena descritto. L'obiettivo di fondo è **ridurre il disallineamento tra il modello mentale dell'utente e la struttura tecnico-gerarchica del catalogo prodotto**, supportando la selezione del modello base più coerente con le esigenze operative espresse. Dallo scenario d'uso emerge che l'ostacolo principale non è la complessità del catalogo in sé, ma il fatto che l'utente deve affrontarla da solo. Il primo obiettivo è quindi la **riduzione del carico cognitivo**: un utente come Marco dovrebbe poter descrivere il proprio contesto di lavoro senza dover prima imparare la segmentazione commerciale del catalogo. Questo presuppone che il sistema sia capace di colmare la distanza fra i due linguaggi. Da qui il secondo obiettivo, la **mediazione semantica**: il sistema deve tradurre le esigenze espresse in linguaggio naturale. Ma la traduzione da sola non basta se i risultati non sono affidabili: dato che una scelta sbagliata in fase di selezione si propaga fino alla configurazione finale, i modelli suggeriti devono essere limitati ai soli prodotti presenti nel catalogo e coerenti con i vincoli espressi. È ciò che qui si definisce **coerenza decisionale**. Infine, il configuratore esistente funziona già bene nella fase di personalizzazione. Il modulo deve quindi garantire un'**integrazione non invasiva**: operare come componente complementare al CPQ, producendo una lista di modelli configurabili compatibile con il flusso già in essere, senza modificarne le logiche.

3.4 Individuazione dei requisiti

3.4.1 Criteri per definire le priorità

La priorità di ciascun requisito è stata definita sulla base di tre criteri. Il primo considera l'**impatto sugli obiettivi**, ovvero quanto il requisito contribuisce direttamente alla riduzione del disallineamento tra bisogno operativo e struttura del catalogo, che è l'obiettivo generale del sistema. Il secondo valuta il **rischio in caso di assenza**, analizzando le conseguenze sul funzionamento del sistema se il requisito non venisse implementato; quelli la cui mancanza comprometterebbe la coerenza del suggerimento o l'integrazione con il CPQ sono stati classificati ad alta priorità. Il terzo tiene conto dei **vincoli di integrazione tecnica** ed in particolare considera la necessità di mantenere compatibilità con l'architettura esistente e con i sistemi coinvolti, evitando modifiche invasive. I requisiti con priorità **Alta** sono quindi quelli indispensabili per la funzionalità minima del modulo, mentre quelli a priorità **Media** migliorano l'esperienza utente e la completezza dell'interazione, ma il sistema può funzionare anche senza di essi se non implementati in una prima fase.

3.4.2 Tracciabilità tra evidenze e requisiti

La Tabella 3.1 riassume il collegamento tra le principali evidenze emerse dall'analisi e i requisiti che ne derivano. Si è tenuto anche conto di aspetti intrinseci al dialogo utente-sistema, come la presenza di informazioni incomplete o ambigue. La tracciabilità tra problemi osservati e requisiti garantisce coerenza tra analisi del contesto e soluzione proposta.

	Evidenza osservata	**Implicazione progettuale**	**Requisito derivato**
E1	Gli utenti meno esperti conoscono il bisogno operativo ma non la classificazione nel catalogo	Necessità di mediare tra linguaggio naturale e struttura del catalogo	Estrazione di informazioni dalle esigenze dell'utente
E2	Il configuratore non prevede interazione adattiva: ogni selezione è definitiva e indipendente dalle precedenti	Necessità di gestire un'interazione multiturno mantenendo il contesto	Gestione dell'interazione conversazionale; Richiesta chiarimenti; Robustezza
E3	Le scelte effettuate non vengono riepilogate né validate rispetto alle esigenze operative	Necessità di fornire un riepilogo delle informazioni estratte	Generazione di un recap.
E4	Il configuratore richiede una selezione preliminare e accurata dalla famiglia al modello	Il sistema deve supportare la fase di orientamento iniziale e migliorare l'allineamento tra esigenze e modelli suggeriti	Suggerimento di famiglia e di modelli coerenti; Accuratezza.
E5	Il CPQ esistente gestisce già configurazione e pricing	La soluzione deve integrarsi senza modificarne le logiche interne	Integrazione con il CPQ; Modularità dell'architettura
E6	Il configuratore non prevede alcun messaggio o percorso alternativo in assenza di modelli idonei	Il sistema deve gestire esplicitamente l'assenza di risultati compatibili	Gestione eccezioni
E7	Una parte di utenza conosce già una macchina di riferimento e può partire da quella per descrivere le proprie esigenze	Il sistema deve poter recuperare il profilo d'uso della macchina citata per orientare la conversazione	Ricerca del prodotto di riferimento

Tabella 3.1: Dall'analisi del contesto ai requisiti progettuali

3.4.3 Requisiti funzionali

A partire dagli obiettivi individuati e dall'analisi del contesto descritta nelle sezioni precedenti, sono stati definiti i seguenti **requisiti funzionali**:

Titolo	**Descrizione**	**Priorità**
Gestione interazione	Gestione di un dialogo conversazionale mantenendo il contesto tra i messaggi.	Alta
Richiesta chiarimenti	Capacità di porre domande quando le informazioni risultano incomplete o ambigue.	Alta
Estrazione informazioni	Traduzione del linguaggio naturale in attributi strutturati del catalogo.	Alta
Suggerimento modelli	Proposta di famiglia e modelli coerenti con i vincoli espressi.	Alta
Generazione recap	Sintesi delle informazioni raccolte durante l'interazione.	Media
Gestione eccezioni	Comunicazione esplicita all'utente quando nessun modello risulta compatibile con le esigenze espresse, con proposta di avviare una nuova ricerca.	Alta
Ricerca del prodotto di riferimento	Supporto a utenti che citano una macchina di confronto: il sistema ne recupera il profilo d'uso per orientare la conversazione senza richiedere domande aggiuntive.	Media
Integrazione CPQ	Trasferimento del modello selezionato al configuratore esistente.	Alta

Tabella 3.2: Requisiti funzionali del modulo di Assisted Selling

La **gestione dell'interazione** e la **richiesta di chiarimenti** sono requisiti strettamente legati: il primo impone al sistema di mantenere il contesto della conversazione tra un messaggio e il successivo, consentendo all'utente di raffinare progressivamente la propria richiesta senza dover ripetere informazioni già fornite; il secondo specifica cosa fare quando quelle informazioni non bastano o quando le informazioni fornite sono poco chiare. In quel caso il sistema deve formulare domande mirate, non generiche: se l'utente indica semplicemente "lavoro nei campi", la risposta giusta non è presentare l'intero catalogo, ma chiedere quale tipo di lavorazione, su quale estensione e con quali attrezzature. E se le domande sono

ambigue, chiedere chiarimenti per permettere all'utente di esprimere le proprie esigenze in modo chiaro. È la differenza tra un questionario precompilato e una conversazione che si adatta a ciò che l'utente ha già detto.

L'**estrazione delle informazioni** e il **suggerimento dei modelli** sono i due requisiti che traducono il dialogo in un output concreto. Il primo richiede che il sistema riconosca nel linguaggio naturale gli attributi strutturati rilevanti per la selezione, ad esempio tipo di attività, caratteristiche del terreno, attrezzature, vincoli operativi, senza che l'utente li debba esplicitare nella forma tecnica del catalogo. Il secondo stabilisce che i prodotti suggeriti debbano appartenere al catalogo reale e che le loro caratteristiche siano affini ai bisogni espressi.

La **gestione delle eccezioni** aggiunge un livello di credibilità del sistema nel caso in cui nessun modello soddisfi i vincoli espressi. Il sistema non deve mai forzare un suggerimento inadatto: dichiarare esplicitamente l'assenza di risultati e offrire all'utente la possibilità di ridefinire le proprie esigenze è preferibile a proporre qualcosa di incoerente.

La **ricerca del prodotto di riferimento** riguarda uno dei tre profili utente descritti ed in particolare chi parte da una macchina già conosciuta anziché da un bisogno operativo astratto. In questo caso il sistema deve essere in grado di reperire informazioni sul prodotto citato e usarle per ridurre il numero di domande necessarie, migliorando la fluidità della conversazione.

L'**integrazione CPQ** chiude il flusso: il modello selezionato dall'utente deve poter essere trasferito al configuratore esistente per la fase di personalizzazione senza interventi manuali intermedi, in modo che l'Assisted Selling si comporti da punto d'accesso per il CPQ e non da percorso alternativo ad esso.

3.4.4 Requisiti non funzionali

Oltre agli aspetti funzionali, il sistema deve soddisfare una serie di **requisiti non funzionali**, che indicano gli standard e le qualità che il sistema deve soddisfare, piuttosto che funzionalità specifiche. In particolare, sono stati individuati:

Categoria	**Descrizione**	**Priorità**
Accuratezza	Suggerimenti coerenti con la struttura del catalogo.	Alta
Robustezza	Gestione di input incompleti o ambigui: il sistema deve saper proseguire il dialogo, formulando domande di chiarimento mirate, senza interrompersi né produrre suggerimenti incoerenti.	Alta
Prestazioni	Tempo di risposta compatibile con un'interazione in tempo reale.	Alta
Modularità	Integrazione con più cataloghi e senza modificare la logica del CPQ esistente.	Alta

Tabella 3.3: Requisiti non funzionali del modulo di Assisted Selling

L'**accuratezza** è il requisito più vincolante sul piano architetturale. Ogni modello suggerito deve essere un prodotto effettivamente presente nella *Product Offering* attiva: il sistema non può fare riferimento a prodotti inesistenti né proporre modelli non compatibili con i vincoli espressi. Questa scelta è dettata anche dal contesto applicativo: nel settore della meccanizzazione agricola, un acquisto è un investimento importante con un orizzonte pluriennale, e un suggerimento impreciso può portare a decisioni sbagliate da cui difficilmente si può tornare indietro.

La **robustezza** è una proprietà necessaria in qualsiasi sistema conversazionale. Un utente che descrive le proprie esigenze in modo approssimativo o incompleto non deve trovarsi di fronte a un comportamento inatteso del sistema: il dialogo deve poter continuare, eventualmente con una domanda di chiarimento.

Il requisito di **prestazioni** deriva dal contesto d'uso descritto nella Sezione 3.2. Un'interazione conversazionale presuppone tempi di risposta compatibili con il ritmo di un dialogo: attese superiori a qualche secondo interrompono la percezione di continuità dell'interazione. Il sistema deve rispondere ad ogni messaggio nell'ordine di pochi secondi.

La **modularità** riguarda due aspetti distinti. Da un lato, il modulo deve operare senza interferire con le logiche del CPQ esistente. Dall'altro, deve poter essere applicato a cataloghi diversi senza richiedere modifiche strutturali: è questa proprietà che ne rende possibile l'utilizzo all'interno di CoolPIM su contesti e clienti differenti, al di là del caso di studio qui analizzato.

I requisiti funzionali e non funzionali individuati costituiscono la base per le scelte architetturali e progettuali descritte nel capitolo successivo.

Capitolo 4

Progettazione e architettura del sistema

In questo capitolo si presentano le scelte progettuali e architetturali attraverso cui si è cercato di soddisfare i requisiti e gli obiettivi individuati nel Capitolo 3. Più che limitarsi a descrivere la struttura della soluzione, l'intento è spiegare il ragionamento dietro ogni decisione partendo dai vincoli concreti imposti dal configuratore esistente e dal contesto in cui il sistema si inserisce.

Il nodo centrale del progetto sta nel colmare la distanza fra il modo in cui un cliente ragiona - per esigenze operative, tipo di impiego, condizioni di lavoro - e il modo in cui il catalogo Case IH è organizzato, cioè per gerarchie tecniche e specifiche di prodotto. In pratica, il sistema deve prendere una serie di esigenze formulate in linguaggio naturale e arrivare ad esporre un insieme di modelli configurabili all'interno di una gerarchia con centinaia di varianti.

Per affrontare il problema si è scelto di scomporlo in quattro aspetti, ognuno trattato in una sezione dedicata:

1. **Rappresentazione dei dati** (Sezione 4.2): come dare ai dati di catalogo una struttura che ne conservi le relazioni e si presti ad essere interrogata semanticamente;

2. **Confrontabilità semantica** (Sezione 4.3): come mettere in comunicazione due linguaggi diversi per natura, quello dell'utente e quello del catalogo, in modo che una descrizione operativa possa essere ricondotta a una specifica tecnica;

3. **Orchestrazione del dialogo** (Sezione 4.4): come condurre la conversazione con l'utente e coordinare gli accessi ai dati fino a proporre un insieme coerente di modelli configurabili;

4. **Integrazione con il CPQ** (Sezione 4.5): come inserire il modulo nel flusso di configurazione già esistente senza modificarne le logiche di configurazione e pricing.

Prima di entrare nel merito dei singoli aspetti, serve però inquadrare dove il modulo si colloca all'interno dell'infrastruttura esistente.

4.1 Contesto architetturale

Il modulo di Assisted Selling non nasce come sistema indipendente: è stato pensato fin dall'inizio come **componente integrato** nella piattaforma *CoolPIM* (Product Information Management) sviluppata da *Coolshop*, la stessa piattaforma che già gestisce i dati di catalogo su cui il modulo lavora. Dietro questa scelta ci sono due ragioni concrete.

La prima è di natura tecnica e risponde al requisito di **modularità** (Tabella 3.3): il modulo deve poter funzionare su cataloghi diversi senza richiedere modifiche alla sua struttura. Appoggiandosi a un PIM che già tratta dati di prodotto per clienti e mercati differenti, questa flessibilità viene quasi gratis. Basta infatti qualche leggero adattamento e che i dati del catalogo siano presenti nel sistema perché il modulo conversazionale possa operarci sopra, a prescindere dal dominio.

La seconda ragione è di carattere aziendale: integrare il modulo in una piattaforma che i clienti già utilizzano ne semplifica la distribuzione, evitando di introdurre un prodotto separato con i suoi costi di integrazione, manutenzione e formazione.

Nel caso di studio qui analizzato, i dati di prodotto arrivano dal sistema legacy *Pricebook Back Office* (PBO), che fa da fonte primaria per la *Product Offering* del mercato UK di Case IH. PBO funziona come CMS (*Content Management System*) e contiene le informazioni di classificazione, i contenuti e i media dei prodotti a catalogo. Un processo di importazione si occupa di estrarre i dati da PBO, trasformarli e caricarli in CoolPIM, dove vengono riorganizzati secondo il modello dati descritto nelle sezioni successive.

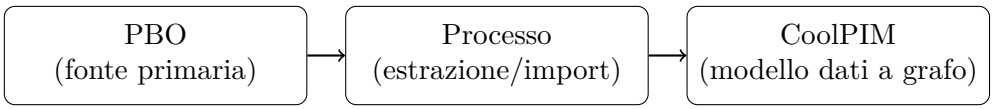

Figura 4.1: Flusso di importazione dei dati da PBO a CoolPIM

4.2 Rappresentazione dei dati di catalogo

Il primo problema da affrontare è come organizzare i dati di prodotto perché il modulo conversazionale riesca a interrogarli in modo efficace. La decisione non

è banale: da come si strutturano i dati dipende quali domande il sistema può formulare, quanto velocemente ottiene le risposte e con quanta facilità riesce a seguire i collegamenti fra entità diverse. Per spiegare le scelte fatte conviene partire da come i dati si presentano nella fonte primaria e capire quali vincoli quella struttura porta con sé.

4.2.1 I dati nella fonte primaria: struttura e limiti

In PBO ogni prodotto è memorizzato come un **record piatto**, vale a dire un insieme di attributi che ne descrivono la posizione dentro la classificazione del catalogo:

- **Gamma**: la famiglia di prodotto principale (es. *Tractors*);
- **Range**: la serie o linea commerciale (es. *Optum AFS Connect*);
- **priceBookName**: sotto-segmentazione della serie, detta anche **Subrange** (es. *Optum CVXDrive*);
- **FDP**: codice tecnico che suddivide i prodotti all'interno dello stesso Subrange;
- **Model**: il modello specifico configurabile (es. *Optum .270 CVXDrive*);
- **Techtype**: la tipologia tecnica del prodotto.

Product	
...	
Gamma	TRACTORS
Range	OPTUM AFS CONNECT
priceBookName	OPTUM CVXDRIVE
FDP	T1LX
Model	OPTUM .270 CVXDRIVE
Techtype	696196646
...	

Figura 4.2: Esempio di record prodotto in PBO

La gerarchia qui è solo *implicita*: è contenuta nei valori degli attributi, non nella struttura del dato. Per sapere che il modello *Optum .270 CVXDrive* appartiene alla famiglia *Tractors* bisogna andare a leggere il campo Gamma del suo record, in quanto non esiste una relazione diretta fra un'entità «Tractors» e un'entità «Optum .270 CVXDrive» che si possa percorrere come tale.

Accanto agli attributi di classificazione, PBO gestisce anche un secondo livello informativo che per il processo di suggerimento si rivela fondamentale. Per ogni **Range** sono definite una o più **Key Feature**, ovvero caratteristiche di natura descrittiva e commerciale che sintetizzano i punti di forza dei prodotti della serie. Ogni *key feature* può essere associata a specifici **Model** del Range, così da indicare esattamente a quali modelli si applica (quando queste associazioni mancano, la feature vale per tutti i modelli della serie). Le *key features* possono a loro volta abilitare degli **Optionals**, cioè equipaggiamenti opzionali legati alla feature e non direttamente al prodotto.

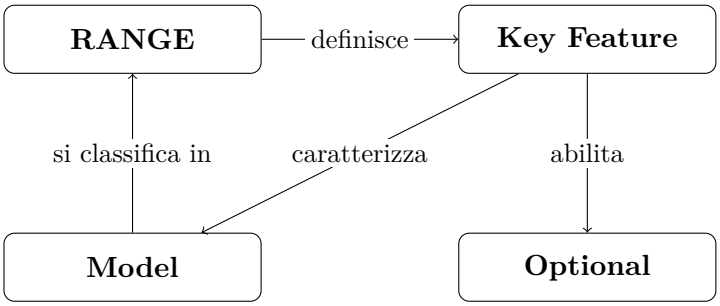

Figura 4.3: Struttura concettuale in PBO: relazioni tra Range, Key Features, Optionals e Model

Dall'analisi della fonte emergono due proprietà del dominio che pesano sulla scelta del modello dati:

1. i dati hanno una natura **gerarchica**: esiste un ordinamento naturale Gamma → Range → Subrange → Model → Techtype, dove ogni livello rappresenta un raffinamento progressivo della classificazione;
2. esistono **relazioni trasversali fra entità di categorie diverse**: le *key features* si riferiscono ad un Range ma possono essere associate a specifici Model al suo interno, e gli *optionals* sono a loro volta legati alle singole key feature. Si tratta di dipendenze incrociate che non sono interne alla struttura gerarchica stessa dei record di prodotto, ma sono relazioni tra strutture diverse.

4.2.2 Perché un database a grafo

Una struttura che è allo stesso tempo gerarchica e percorsa da relazioni trasversali corrisponde, nella sua essenza, a un **grafo**: un insieme di nodi collegati da archi dove le relazioni fra entità sono parte integrante del modello dati, non un sottoprodotto di chiavi esterne [5]. Partendo da questa osservazione si è scelto di rappresentare i dati nel **database a grafo Neo4j**, che CoolPIM già utilizza come infrastruttura di storage.

Il motivo della scelta sta nel modo in cui i due tipi di database gestiscono le relazioni. In un RDBMS le connessioni fra tabelle sono implicite nelle chiavi esterne e vengono materializzate a tempo di query con operazioni di *join*, il cui costo scala con la dimensione delle tabelle coinvolte [6]. In un database a grafo, invece, ogni relazione è un arco fisico fra due nodi: percorrerne uno ha costo costante, indipendente dal numero totale di nodi nel database [5]. La differenza si fa sentire quando le query attraversano più livelli di relazione, che è esattamente il caso del processo di suggerimento.

Per il modulo di Assisted Selling questa differenza è rilevante. Una domanda tipica del processo di suggerimento ha la forma: «date certe caratteristiche ritenute rilevanti, quali modelli vi sono collegati?». In un grafo si tratta di percorrere direttamente gli archi che legano feature e modelli; in un database relazionale servirebbe unire più tabelle - prodotti, feature, associazioni prodotto-feature - con un costo che sale a ogni livello aggiuntivo.

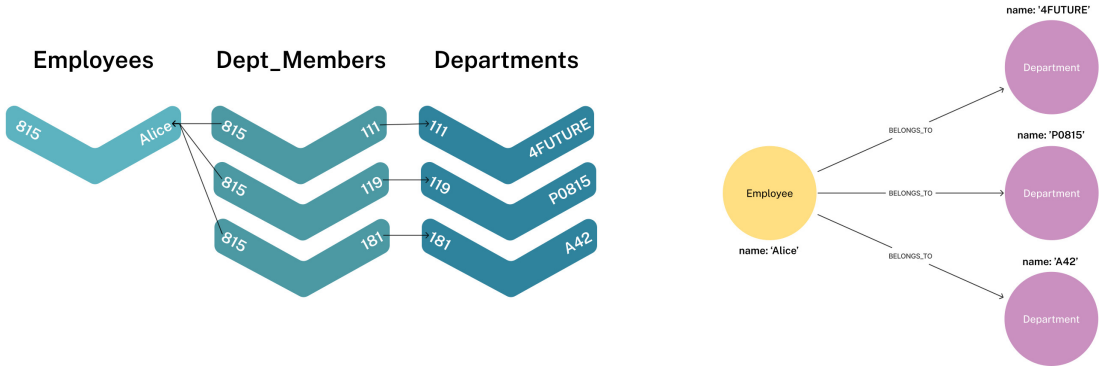

Figura 4.4: Confronto tra la rappresentazione delle stesse informazioni in un database relazionale e in un database a grafo. Fonte: [6]

La scelta del grafo non nasce quindi da una preferenza tecnologica astratta, ma dalla **natura delle interrogazioni** che il sistema deve eseguire e dal fatto che CoolPIM, la piattaforma su cui la soluzione è costruita, dispone già dell'infrastruttura di persistenza adatta. Il passaggio al grafo rende esplicito ciò che in PBO resta implicito: la gerarchia diventa un albero di nodi con relazioni padre-figlio, le associazioni trasversali diventano archi fra nodi di categorie diverse, e percorrere queste strutture diventa un'operazione nativa del sistema di storage.

Neo4j mette inoltre a disposizione due funzionalità che torneranno utili nelle sezioni successive: il linguaggio **Cypher**, che permette di esprimere in modo dichiarativo pattern di attraversamento anche complessi [7], e il supporto nativo per l'**indicizzazione vettoriale** sulle proprietà dei nodi, grazie al quale è possibile combinare ricerca semantica e navigazione del grafo all'interno della stessa query.

4.2.3 Modellazione gerarchica in CoolPIM

CoolPIM [8] organizza i dati come **alberi orientati ai nodi foglia**, dove gli attributi del padre vengono ereditati dai figli e possono essere sovrascritti a qualsiasi livello. I dati provenienti da PBO sono stati distribuiti su tre gerarchie separate:

- **Product Offering**: cinque livelli - *Gamma* → *Range* → *Subrange* → *Model* → *Techtype*;
- **Key Features**: nodi monolivello, ciascuno corrispondente a una singola caratteristica descrittiva o commerciale;
- **Optionals**: nodi monolivello, ciascuno corrispondente a un equipaggiamento opzionale.

A collegare le tre gerarchie ci sono **link cross-gerarchia**: i nodi della Product Offering (Range o Model) puntano verso le Key Features, e le Key Features puntano verso gli Optionals. Viene naturale chiedersi perché tre gerarchie e non un'unica struttura che contenga tutte le entità. Le ragioni sono tre, e ciascuna risponde a un'esigenza concreta.

1. **Indipendenza di aggiornamento.** Il catalogo viene importato periodicamente da PBO, ma i dati di prodotto, le key features e gli optionals non cambiano necessariamente insieme. Se Key Features e Optionals fossero nodi interni alla stessa gerarchia del catalogo, un aggiornamento della Product Offering rischierebbe di invalidare o spostare anche i nodi delle feature. Tenendoli separati, ogni gerarchia può essere aggiornata per conto suo senza effetti collaterali sulle altre, il che rende il modello più facile da mantenere nel tempo;

2. **Ricerca vettoriale separata.** Come si vedrà nella Sezione 4.3, il modulo esegue ricerche per similarità semantica su tipi di entità diversi: talvolta sulle Key Features, per individuare caratteristiche rilevanti, talvolta sui Model, per confrontare profili d'uso complessivi. Avere gerarchie distinte permette di costruire indici vettoriali separati per ciascun tipo di nodo e di interrogarli senza interferenze reciproche, con vantaggi sia in termini di precisione dei risultati sia di prestazioni;

3. **Omogeneità dello schema.** La piattaforma è pensata per gestire gerarchie distinte, ognuna con il proprio schema di nodi e le proprie regole di ereditarietà. Prodotti, Key Features e Optionals hanno strutture e attributi molto diversi fra loro: i prodotti seguono una classificazione a cinque livelli, le Key Features sono entità monolivello con contenuti descrittivi, gli Optionals sono entità monolivello con poco più di un codice identificativo. Costringerli in un'unica

gerarchia avrebbe significato imporre uno schema comune ad entità eterogenee, con nodi a livelli diversi che si sarebbero trovati a condividere attributi privi di senso per buona parte di essi.

4.2.4 Scelte di modellazione dei nodi

Ogni nodo del grafo è identificato da un codice univoco e organizza i propri attributi in *Content Type Group*, raggruppamenti semantici definiti dall'architettura di CoolPIM che separano le informazioni secondo la loro funzione. Alcune scelte di modellazione hanno un impatto diretto sul funzionamento del modulo e vale la pena discuterle esplicitamente.

Denormalizzazione del contesto gerarchico. Nei nodi della Product Offering, attributi come `gammaId`, `gammaName` o `rangeName` sono **denormalizzati**: compaiono come proprietà dirette di ogni nodo figlio invece di essere ricavati risalendo l'albero. In un modello normalizzato, per conoscere la famiglia di appartenenza di un nodo Model basterebbe risalire l'albero fino al nodo Gamma - un'operazione con costo $O(d)$ per ciascun nodo, dove d è la profondità dell'albero.

La ridondanza è voluta e risponde al requisito di **prestazioni**(Tabella 3.3). Quando il sistema esegue una ricerca vettoriale e ottiene decine di nodi Model candidati, deve riuscire a filtrarli per famiglia senza dover risalire l'albero per ciascuno di essi. Denormalizzando, un'operazione che costerebbe $O(n \cdot d)$ (con n candidati) si riduce a un filtro sulle proprietà locali del nodo, con costo $O(n)$. Lo spazio occupato in più è trascurabile rispetto al volume dei dati in gioco e ampiamente ripagato dalla velocità di interrogazione.

CoolPIM rende questa scelta particolarmente naturale grazie al suo meccanismo di **ereditarietà degli attributi**: nelle relazioni padre-figlio gli attributi del padre vengono propagati automaticamente ai discendenti, per cui la denormalizzazione non richiede manutenzione manuale ma è gestita dall'infrastruttura stessa.

Gestione dell'FDP. L'analisi della Product Offering ha evidenziato che lo stesso codice FDP può comparire in Subrange diversi, rendendo impossibile trattarlo come livello gerarchico autonomo. L'FDP è stato perciò modellato come attributo del nodo Subrange, con un nodo creato per ciascuna coppia Subrange-FDP (la loro combinazione determina il codice identificativo del nodo).

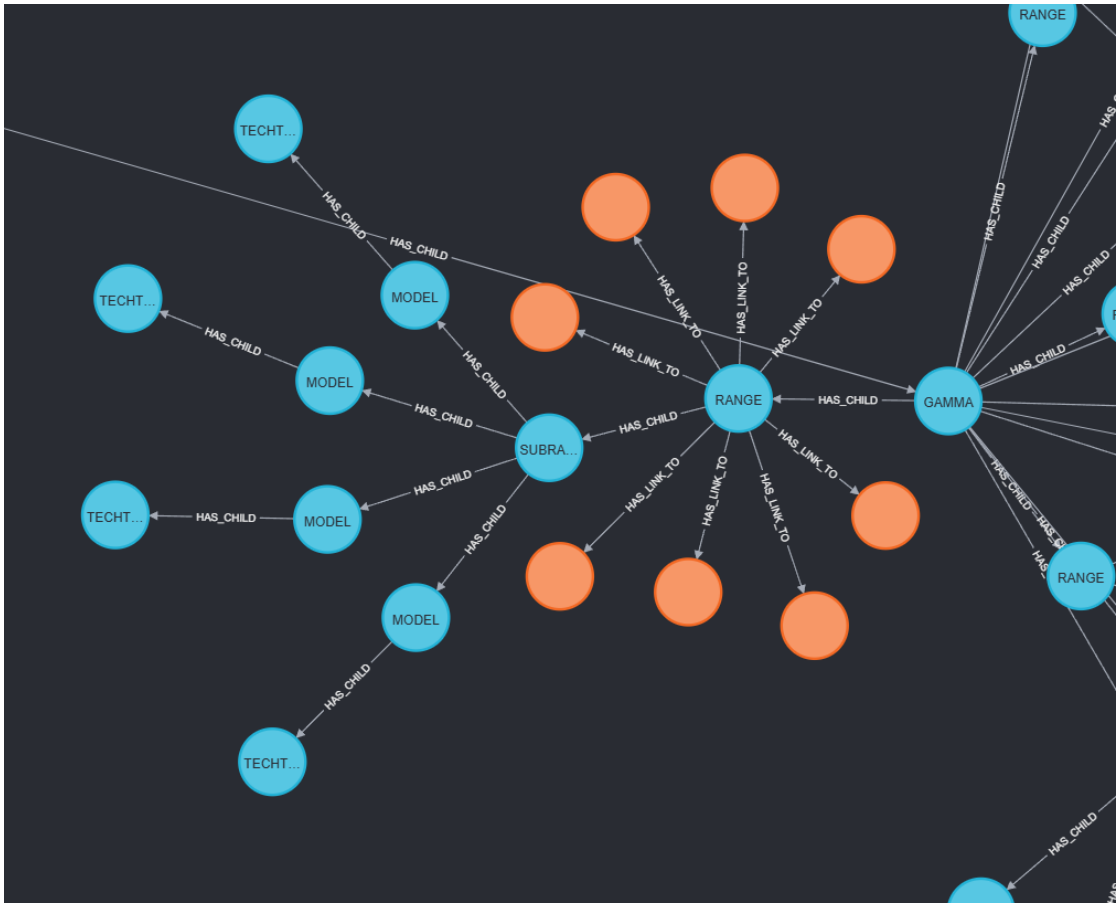

Figura 4.5: Un ramo della gerarchia Product Offering (azzurro) con link cross-gerarchia verso Key Features (arancio) in CoolPIM. *Fonte: elaborazione propria.*

Inversione della direzione dei link cross-gerarchia. In PBO il collegamento fra key feature e modello va nella direzione *Key Feature* → *Model*: è il record della feature a contenere il riferimento al modello. Nel grafo CoolPIM la freccia va nel verso opposto: `(Model)-[:HAS_LINK_TO]->(KeyFeature)`. Si è preferita questa direzione perché rispecchia il rapporto di possesso fra le due entità - un modello *ha* certe caratteristiche distintive, non il contrario - e rende il grafo leggibile quasi come una frase in linguaggio naturale. Robinson et al. considerano questa leggibilità uno dei criteri guida per decidere l'orientamento delle relazioni in un property graph [5].

Sul piano delle prestazioni l'inversione non ha alcun costo. Neo4j indicizza ogni relazione in entrambe le direzioni, per cui l'attraversamento da una Key Feature verso i Model che la possiedono (operazione frequente quando si parte da un insieme di feature candidate) è altrettanto veloce di quello nel verso opposto [9]. In sostanza, si guadagna in chiarezza senza rinunciare a nulla in efficienza.

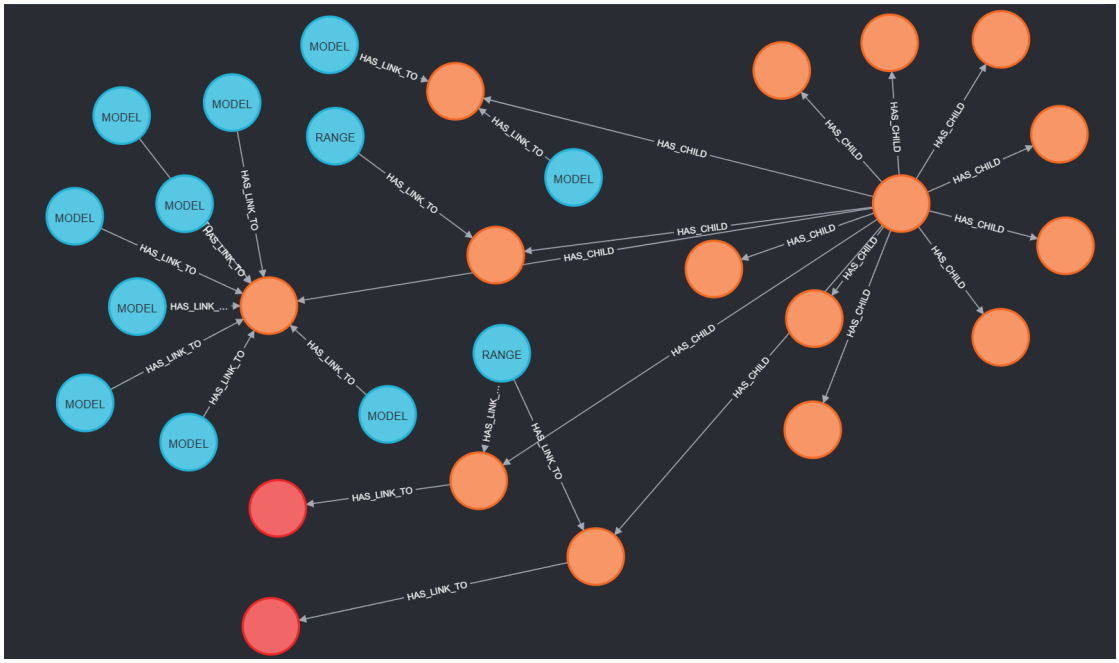

Figura 4.6: Gerarchia Key Features (arancio) con link da Product Offering (azzurro) e verso Optionals (rosso) in CoolPIM. *Fonte: elaborazione propria.*

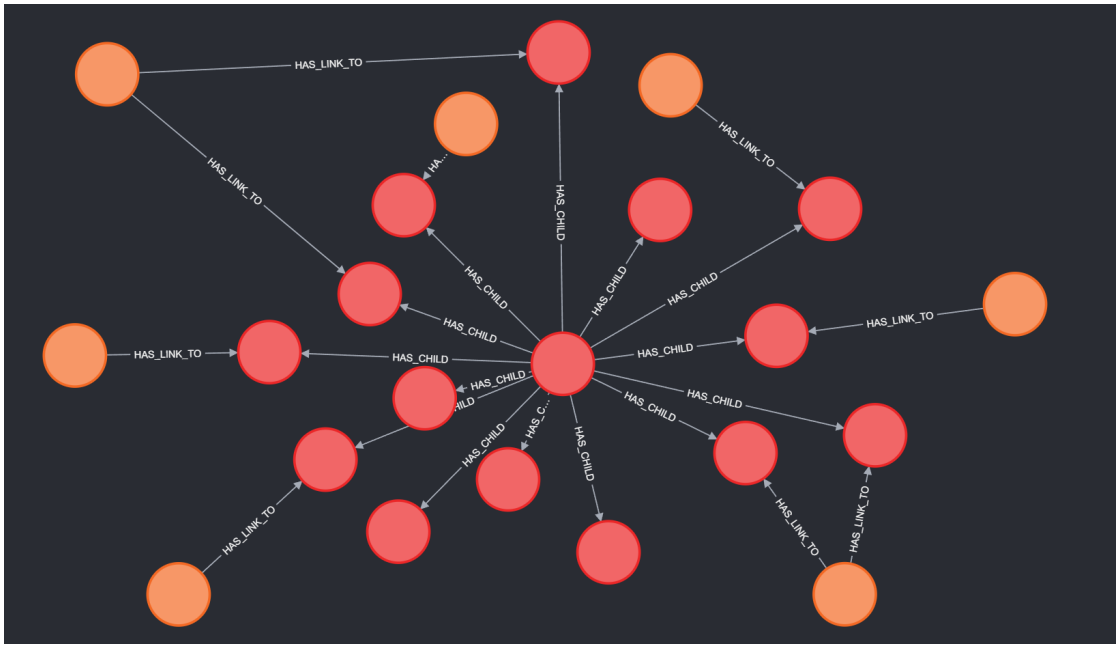

Figura 4.7: Gerarchia Optionals (rosso) con link da Key Features (arancio) in CoolPIM. *Fonte: elaborazione propria.*

A questo punto il catalogo è un grafo con relazioni esplicite, gerarchie diverse e nodi popolati con attributi descrittivi. Ma questi attributi parlano il linguaggio tecnico del catalogo (specifiche, sigle, denominazioni) mentre l'utente parla un altro linguaggio, quello dei bisogni operativi. Per questo motivo il problema successivo è trovare un modo per rendere le due rappresentazioni confrontabili.

4.3 Rendere il catalogo confrontabile con il linguaggio naturale

Il requisito di **estrazione informazioni** (Tabella 3.2) chiede che il sistema sappia tradurre ciò che l'utente esprime in linguaggio naturale in attributi strutturati del catalogo. Il requisito di **mediazione semantica** (Sezione 3) va oltre: il sistema deve fare da ponte fra il bisogno operativo e la struttura tecnica del catalogo. Resta da capire nella pratica con quali strumenti costruire questo ponte.

Si consideri un esempio concreto. Un agricoltore che dice di aver bisogno di «lavorare su terreni ripidi con attrezzi pesanti» sta esprimendo un'esigenza perfettamente chiara. Ma il catalogo non contiene quella frase: contiene specifiche come «controllo di trazione per pendenze» o «sollevamento posteriore categoria III». Le due formulazioni dicono sostanzialmente la stessa cosa, ma non condividono nemmeno una parola. Una ricerca per parole chiave (*keyword matching*) non troverebbe nulla: «terreni ripidi» e «pendenze» non hanno sovrapposizione lessicale. Nemmeno approcci basati su dizionari di sinonimi o ontologie lessicali [10] riuscirebbero nell'intento, considerata la grande distanza fra il lessico operativo dell'utente e la terminologia tecnica del catalogo.

Il problema è di natura **semantica**: serve una rappresentazione in cui sia il *significato* del testo a determinare la corrispondenza fra entità, non la sua forma superficiale.

4.3.1 Text embedding: dal testo allo spazio vettoriale

Il problema posto nella sezione precedente ha una proprietà precisa: i testi da confrontare esprimono concetti affini ma non condividono vocabolario. In questi casi serve una rappresentazione che codifichi il *significato* del testo in modo indipendente dalle parole usate, così da poter misurare la vicinanza semantica fra stringhe lessicalmente distanti. I **text embedding** fanno esattamente questo: proiettano il contenuto semantico di una stringa in uno spazio vettoriale ad alta dimensionalità, dove la distanza fra punti riflette la somiglianza di significato e non la sovrapposizione lessicale. Due testi che dicono la stessa cosa con parole diverse finiscono vicini; due testi che condividono parole ma parlano di cose diverse finiscono lontani.

L'idea che il significato di una parola dipenda dai contesti in cui compare risale alla linguistica distribuzionale di Harris [11]. I modelli Word2Vec [12] e GloVe [13] hanno tradotto questa intuizione nella pratica, imparando a collocare ogni parola in uno spazio vettoriale dove la distanza fra punti riflette la vicinanza di significato. Il problema era che ciascun termine riceveva un vettore unico, uguale a prescindere dalla frase in cui compariva: «banco», ad esempio, aveva la stessa rappresentazione sia nel contesto scolastico sia in quello finanziario.

L'architettura Transformer [14] ha risolto questo limite con il meccanismo di *self-attention*, che calcola la rappresentazione di ogni token tenendo conto di tutti gli altri nella stessa sequenza. Modelli come BERT [15] e la sua variante per la similarità testuale Sentence-BERT [16] producono **embedding contestuali**, ovvero vettori che catturano il significato di frasi intere, non di singoli termini, e il cui risultato pratico è la possibilità di confrontare testi formulati con parole completamente diverse, purché esprimano concetti affini.

Nel sistema di Assisted Selling il meccanismo è semplice: il modello di embedding prende in input un testo, che sia una descrizione dal catalogo o una richiesta dell'utente, e ne restituisce un vettore numerico. Se due testi hanno un significato affine, i loro vettori finiscono vicini nello spazio; se non c'entrano nulla l'uno con l'altro, finiscono lontani. Questo permette di misurare quanto il bisogno operativo espresso dall'utente sia «vicino» alle caratteristiche di un prodotto, anche quando i due testi non hanno una sola parola in comune.

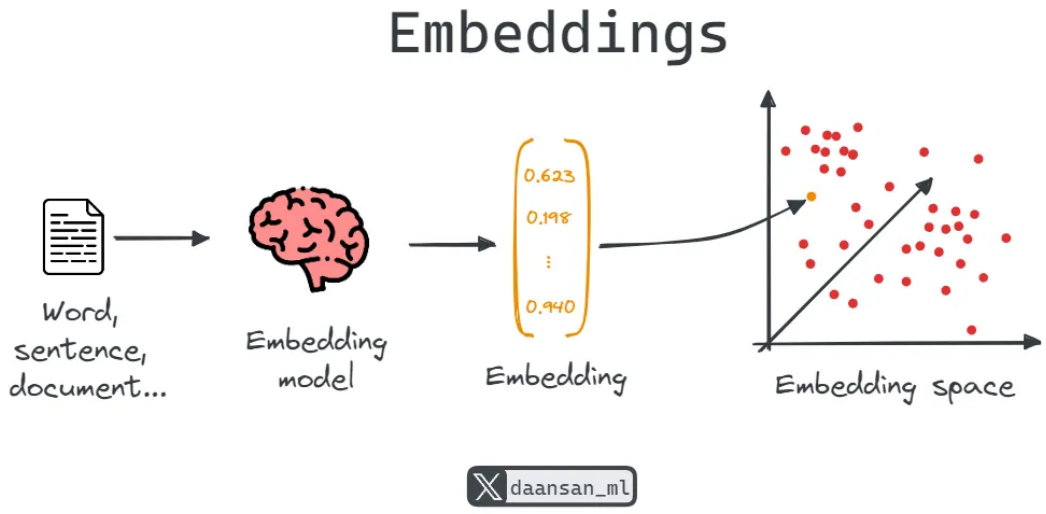

Figura 4.8: Rappresentazione del processo di embedding. *Fonte:* [17]

4.3.2 Quali contenuti indicizzare e perché

Stabilito che gli embedding sono lo strumento giusto per la mediazione semantica, resta da decidere *su quali contenuti calcolarli*. Un nodo Model ha parecchi attributi - `name`, `type`, `standard_features`, `additional_description`, `product_image` - ma non tutti servono allo stesso modo quando si tratta di confrontarli con ciò che chiede l'utente.

Il criterio seguito è stato: **indicizzare gli attributi che l'utente userebbe se gli chiedessero di spiegare a cosa serve il prodotto**. Come emerso dall'analisi nella Sezione 3.2, l'utente tipo ragiona in termini di «cosa devo fare», non di «quale codice prodotto mi serve». Nessuno direbbe «ho bisogno del codice T1LX»; direbbe piuttosto «mi serve un trattore potente per lavorare il terreno con attrezzi pesanti». Questo genere di informazione è contenuto negli attributi descrittivi, non in quelli identificativi o di classificazione, ed in particolare nei nodi di tipo *Model* (per la gerarchia di Product Offering) che rappresentano le entità da suggerire all'utente, e nei nodi di tipo *Key Feature*, che rappresentano il raccordo tra esigenze operative e prodotto: descrivono caratteristiche funzionali in un linguaggio vicino a quello con cui l'utente descrive il proprio bisogno.

Per i nodi **Model** l'embedding è calcolato sulla **concatenazione** di due attributi, `standard_features` e `additional_description`: mettendo insieme le funzionalità standard e la descrizione aggiuntiva si ottiene una rappresentazione che racchiude sia le capacità tecniche del prodotto sia il suo contesto d'uso tipico.

Per i nodi **Key Feature** si concatenano invece `title` e `description`: il nome della caratteristica e la sua spiegazione restituiscono un quadro semanticamente completo di cosa la feature offre e in che contesto è rilevante.

In entrambi i casi il vettore ottenuto viene memorizzato come **proprietà del nodo** nel grafo, così che ogni nodo sia interrogabile semanticamente senza bisogno di calcoli al volo.

La scelta di concatenare più attributi in un unico embedding, anziché calcolarne uno per attributo, nasce dall'esigenza di catturare il **significato complessivo** del nodo. Un modello le cui *standard features* parlano di «alta potenza» e la cui *additional description* precisa «ideale per terreni collinari» deve produrre un solo vettore che tenga conto di entrambe le cose, non due vettori separati che andrebbero poi combinati con logiche aggiuntive.

4.3.3 Indicizzazione vettoriale nel grafo

Gli embedding memorizzati nei nodi sono stati indicizzati attraverso il **vector indexing nativo** di Neo4j, che consente di effettuare, ad esempio, operazioni di *vector search* dei k nodi più simili a un vettore di query (*k-nearest neighbors*, KNN) con un costo computazionale molto inferiore rispetto a una ricerca esaustiva su tutti i nodi [18].

La scelta di indicizzare i vettori *dentro* il database a grafo invece che in un sistema vettoriale esterno (Pinecone, Weaviate, Milvus o simili) è una scelta dettata dalla forma dei dati e da come il sistema li interroga. Il processo di suggerimento richiede quasi sempre due operazioni in sequenza: prima trovare le entità semanticamente più vicine a ciò che l'utente ha descritto (per le key feature ad esempio), poi percorrere le relazioni del grafo per capire a quali modelli quelle entità sono collegate. Tenendo tutto dentro Neo4j, ricerca vettoriale e attraversamento del grafo avvengono nello stesso sistema e nella stessa query Cypher, eliminando così la **latenza** fra i due servizi (che sarebbe controproducente per il **requisito prestazionale**) e, come effetto collaterale, semplificando anche l'architettura complessiva.

4.3.4 Metrica di similarità

La vicinanza tra vettori è misurata tramite **cosine similarity**, una metrica ampiamente utilizzata in information retrieval e NLP (Natural Language Processing) per confrontare rappresentazioni vettoriali di testi [19]:

$$\text{sim}(\vec{q}, \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{\|\vec{q}\|\|\vec{d}\|}$$

dove $\vec{q}$ è l'embedding della query dell'utente, $\vec{d}$ l'embedding di un nodo del catalogo, $\vec{q} \cdot \vec{d}$ il prodotto scalare e $\|\vec{q}\|$, $\|\vec{d}\|$ le rispettive norme euclidee. Il risultato, anche chiamato `score`, cade nell'intervallo $[-1, 1]$, anche se negli embedding semantici utilizzati in NLP i valori negativi sono relativamente rari: più è vicino a 1, più i due testi sono semanticamente affini.

Si è preferita la cosine similarity alla **distanza euclidea** o ad altre misure per una ragione legata alla natura degli spazi di embedding. In uno spazio semantico ad alta dimensionalità, ciò che conta per la somiglianza di significato è la *direzione* del vettore, non la sua *lunghezza*. Due frasi che esprimono concetti semanticamente simili tendono a produrre vettori con direzione simile nello spazio degli embedding, ma le loro norme possono differire per motivi interni al modello di embedding, come ad esempio lunghezza del testo in input, distribuzione dei token, normalizzazione interna. La cosine similarity è indifferente alla norma e confronta solo l'orientamento, il che la rende più affidabile per questo tipo di confronto [19]. La distanza euclidea, invece, risente della lunghezza dei vettori e potrebbe giudicare distanti due testi che hanno lo stesso significato ma rappresentati da embedding di lunghezza diversa. Va osservato che molti modelli di embedding recenti restituiscono vettori già normalizzati a norma unitaria, il che rende le due metriche matematicamente equivalenti; la scelta della cosine similarity resta comunque preferibile perché non dipende da questa assunzione e risulta valida anche con modelli che non normalizzano, oltre a essere la metrica supportata nativamente dall'indicizzazione vettoriale di Neo4j.

A questo punto il catalogo è un grafo i cui nodi possono essere confrontati semanticamente con il linguaggio naturale dell'utente. Resta il problema centrale: come gestire il dialogo, decidere quali interrogazioni lanciare e quando, e trasformare una conversazione in un suggerimento coerente con i bisogni espressi.

4.4 Dal dialogo al suggerimento

Nelle sezioni precedenti si è costruita l'infrastruttura dati del sistema: un grafo interrogabile, con nodi confrontabili semanticamente con il linguaggio naturale. Ma l'infrastruttura, da sola, non basta. Serve un componente che **conduca la conversazione** con l'utente: che sappia quali domande fare, come interpretare le risposte, quando ha senso interrogare il catalogo e come mettere insieme i risultati in un suggerimento.

Questi compiti corrispondono ai requisiti funzionali del Capitolo 3: **gestione dell'interazione** conversazionale (tenere il filo del contesto fra un messaggio e l'altro), **richiesta di chiarimenti** (porre domande quando le informazioni sono insufficienti), **estrazione di informazioni** (ricondurre il linguaggio naturale ad attributi strutturati) e **suggerimento di modelli** (proporre prodotti coerenti con le esigenze raccolte).

4.4.1 Un LLM come orchestratore del processo

Si è scelto di affidare questi compiti a un **Large Language Model** (LLM), un modello generativo basato sull'architettura Transformer [14] e addestrato su corpora dell'ordine di centinaia di miliardi di token. L'addestramento conferisce a questi modelli capacità che i loro predecessori non avevano, come mantenere il filo di un dialogo su più turni, adattare il registro linguistico al contesto, e prendere decisioni non banali sulla base di istruzioni testuali [20, 21].

Usare un LLM generico invece di costruire un modello specifico per il dominio non è una scelta scontata, e vale la pena spiegarne le ragioni nel dettaglio.

1. **Non esistono dati di addestramento per questo dominio.** Non ci sono dataset pubblici di conversazioni di vendita di macchinari agricoli, e raccoglierne uno internamente non sarebbe stato realistico: servirebbe trascrivere interazioni reali fra operatori e clienti, annotarle con intenti, entità e azioni di sistema - un lavoro il cui costo in tempo e risorse sarebbe stato sproporzionato rispetto al contesto del progetto. Un LLM generico, invece, può essere guidato attraverso un *system prompt*, un insieme di istruzioni testuali che ne definiscono il comportamento, il dominio e le regole del dialogo, senza alcun addestramento aggiuntivo [20]. Questa tecnica, detta *in-context learning*, permette di adattare il modello al dominio semplicemente fornendo istruzioni

e contesto in input, sfruttando le capacità di generalizzazione che il modello ha acquisito durante il pre-addestramento;

2. **Il linguaggio dell'utente è imprevedibile.** Come emerso dall'analisi del profilo utente (Sezione 3.2), un agricoltore può descrivere le proprie esigenze nei modi più diversi: con più o meno precisione, usando termini tecnici o colloquiali, mescolando requisiti operativi e condizioni ambientali. Un sistema supervisionato richiederebbe pattern espliciti per ogni possibile formulazione, un insieme in pratica illimitato. Un LLM gestisce questa variabilità in modo naturale, grazie alla comprensione contestuale data dall'essere stato addestrato su domini molto diversi fra loro;

3. **La conversazione non segue uno script.** Il sistema non deve percorrere un percorso predefinito: deve reagire a ogni risposta dell'utente e decidere sul momento cosa fare dopo (se fare un'altra domanda, se chiedere un chiarimento, se ha già abbastanza informazioni per cercare nel catalogo). Un LLM tiene traccia del contesto su più turni e prende queste decisioni sulla base dell'intera storia della conversazione. Con un sistema a regole bisognerebbe modellare esplicitamente un automa a stati finiti con transizioni per ogni combinazione possibile di input, con una complessità che crescerebbe in modo esponenziale col numero di variabili del dominio.

4.4.2 Il rischio di allucinazione e il paradigma RAG

Affidarsi a un LLM generico porta con sé un rischio che va affrontato a monte. Un modello pre-addestrato non sa nulla del catalogo Case IH, infatti il suo addestramento è avvenuto su informazioni testuali generiche e non comprende la Product Offering trattata nel dettaglio. Se gli si chiedesse di suggerire un trattore, produrrebbe una risposta che suona credibile ma che potrebbe essere **completamente inventata**: modelli che non esistono, specifiche attribuite al prodotto sbagliato, famiglie confuse fra loro.

In letteratura questo fenomeno si chiama **allucinazione** (*hallucination*) [22]: il modello genera contenuto che sembra affidabile ma fattualmente scorretto o privo di fondamento. Ji et al. [22] distinguono fra allucinazioni *intrinseche*, in cui il modello contraddice il materiale che ha ricevuto in input, e *estrinseche*, in cui produce informazioni non verificabili, e classificano il fenomeno come uno degli ostacoli principali all'uso di sistemi generativi in contesti dove l'affidabilità conta.

Nel contesto della vendita di macchinari industriali, dove un suggerimento errato può portare a decisioni di acquisto sbagliate con conseguenze economiche importanti, questo rischio non è accettabile. Il requisito di **accuratezza** (Tabella 3.3) impone che ogni modello suggerito esista effettivamente nel catalogo al momento dell'interazione. Non basta che il sistema sia «di solito corretto»: deve essere

strutturalmente impossibile che il sistema proponga prodotti inesistenti. Va precisato che cosa si intende: la garanzia architetturale riguarda l'*esistenza* dei modelli suggeriti, poiché l'output finale viene recuperato direttamente dal grafo e non generato dal modello linguistico. Il testo libero che il modello produce attorno ai dati (ad esempio una frase di accompagnamento o un commento sulle caratteristiche) resta invece soggetto al rischio di allucinazione, come qualsiasi output di un LLM, e viene mitigato ma non eliminato dalle istruzioni del system prompt. La garanzia non può dipendere dalla probabilità statistica del modello ma deve essere architettata nel sistema.

Per ottenerla il sistema adotta il paradigma **RAG** (*Retrieval-Augmented Generation*), introdotto da Lewis et al. [23]. L'idea è che il modello non risponda attingendo alla propria conoscenza interna, ma vada prima a recuperare le informazioni rilevanti da una fonte esterna e generi la risposta esclusivamente sulla base di ciò che ha trovato. In questo modo la conoscenza in questione, memorizzata in una raccolta esterna che può essere aggiornata, viene separata dalla capacità di ragionamento e generazione linguistica, che rimane compito del modello. Nella sua forma canonica [23] l'architettura RAG prevede tre componenti: un *retriever* che cerca frammenti rilevanti in un insieme testuale, un meccanismo di *augmentation* che li inserisce nel contesto del modello, e un *generator* che produce la risposta basata su quei frammenti. Questo schema funziona bene quando l'insieme di riferimento è fatto di documenti testuali omogenei (articoli, manuali, FAQ) dove ogni frammento si può interpretare da solo. Il catalogo Case IH, però, non è una raccolta di documenti: è un **grafo con nodi tipizzati e relazioni esplicite**. Trattarlo come un corpus testuale significherebbe appiattire la struttura e perdere proprio le relazioni fra entità, cioè le informazioni che servono per rispondere a domande come «questo modello possiede questa caratteristica?». Un contenuto che descrive una key feature, preso da solo, non dice nulla su quali modelli la possiedono: quell'informazione sta negli archi del grafo, non nel testo. Qui entra in gioco il **function calling**, una capacità dei modelli linguistici più recenti che permette al modello di produrre, come parte del suo output, una richiesta di invocazione di una funzione esterna, con nome e parametri *machine-readable* [24]. Il sistema esegue la funzione, ne restituisce il risultato al modello, e il modello incorpora quel risultato e può utilizzarlo nelle risposte successive. In pratica, il modello passa da generatore passivo ad **agente** capace di interagire con sistemi esterni in modo autonomo.

Questa architettura è stata recentemente definita come **GraphRAG** in letteratura [25], un pattern emergente in cui la fonte di conoscenza del sistema RAG non è un corpus documentale ma un knowledge graph, e il retrieval sfrutta tanto la similarità semantica quanto la struttura relazionale del grafo. Il vantaggio rispetto al RAG documentale sta nella capacità di rispondere a domande che richiedono di comporre informazioni distribuite su più nodi e relazioni, che è esattamente il tipo di interrogazione del processo di suggerimento, dove per dire «quali modelli

corrispondono a queste caratteristiche» bisogna attraversare gli archi che legano Key Feature e Model.

Il modulo è stato inoltre progettato per essere **indipendente dal provider** del modello linguistico. L'interfaccia verso il modello è definita in modo astratto e si specificano le operazioni richieste (generazione di risposte, calcolo di embedding, invocazione di strumenti) senza legarle a un'implementazione concreta. Questo consente di cambiare il modello sottostante, ad esempio passando da un servizio cloud a uno locale o da un provider a un altro, senza dover riprogettare il sistema. Una scelta dettata dalla velocità con cui il settore evolve e dalla necessità di non vincolare il sistema a decisioni tecnologiche modificabili solo con una riprogettazione. Proprio perché non tutti i provider supportano il function calling, l'architettura prevede anche un **flusso alternativo** in cui conversazione e matching avvengono in fasi separate: nella prima fase il modello linguistico gestisce i turni di dialogo, mentre nella seconda le informazioni vengono estratte dalla conversazione senza l'ausilio dell'LLM e utilizzate per ricercare una corrispondenza, definendo un pattern molto più rigido. I dettagli implementativi di entrambi i flussi sono descritti nella Sezione 5.7.

4.4.3 Gli strumenti a disposizione del modello

Il modello linguistico ha a disposizione quattro strumenti, ognuno legato a una capacità specifica che serve nel processo di suggerimento. La loro individuazione è partita da una domanda pratica: quali sono le operazioni che il modello non può svolgere con la sola generazione di testo e per le quali ha bisogno di accedere a dati esterni? Dall'analisi del flusso decisionale dell'utente e dalla struttura del modello dati sono emerse quattro capacità distinte e irriducibili: (1) contestualizzare un prodotto già noto all'utente, che richiede informazioni esterne al catalogo; (2) cercare corrispondenze puntuali fra singole esigenze e caratteristiche del catalogo, che richiede la ricerca vettoriale sulle Key Feature; (3) confrontare il profilo d'uso complessivo con le descrizioni dei modelli, che richiede la ricerca vettoriale sui Model; (4) recuperare i dati verificati dei modelli selezionati, che è l'operazione che chiude il cerchio della garanzia contro l'allucinazione. Nessuna di queste operazioni è riconducibile a un'altra, e ognuna corrisponde ad uno strumento che il modello può utilizzare tramite function calling.

Strumento 1 - Ricerca del prodotto di riferimento. Dall'analisi del profilo utente (Sezione 3.2) è emerso che spesso chi si rivolge al sistema parte da un contesto operativo che già conosce: ha una macchina, ci lavora da anni, e vuole sostituirla o affiancarla con qualcosa di simile. Quando l'utente cita una macchina che possiede o che conosce, per esempio «ho un John Deere 9RX», l'informazione è

molto utile, perché permette al sistema di iniziare a capire il contesto operativo senza doverlo ricostruire da zero.

Questo strumento cerca il profilo d'uso tipico della macchina menzionata: scenari operativi, tipo di impiego, posizionamento di mercato. La fonte è il **web**: poiché l'utente può nominare qualsiasi prodotto, inclusi quelli di concorrenti non presenti nel catalogo Case IH, le informazioni non possono provenire dal grafo interno ma vengono recuperate tramite ricerca web, con modalità che dipendono dal provider AI in uso (come si vedrà nella Sezione 5.2). Il risultato non viene mostrato all'utente ma serve al modello per orientare meglio la conversazione. Se l'utente nomina un trattore da frutteto, il modello non starà a chiedere «lavori in campo aperto o in frutteto?», dovrebbe averlo già capito. Il numero di domande si riduce e l'esperienza migliora, in linea con l'obiettivo di **riduzione del carico cognitivo**.

Strumento 2 - Matching per caratteristiche chiave. Questo è il primo dei due percorsi di matching e si appoggia ad entrambe le infrastrutture costruite nelle sezioni precedenti: la ricerca vettoriale per trovare le key feature rilevanti e l'attraversamento del grafo per risalire ai modelli che le possiedono.

Nella pratica il modello linguistico, sulla base di quanto emerso dalla conversazione, estrae una lista di requisiti puntuali o *caratteristiche*, ad esempio «alta potenza di sollevamento», «trazione integrale», «comfort in cabina». Per ciascuno di essi lo strumento calcola l'embedding, lo confronta con gli embedding calcolati sulle key feature indicizzate nel grafo tramite cosine similarity, e individua le feature semanticamente più vicine. Successivamente percorre le relazioni del grafo per trovare i modelli collegati a ciascuna feature.

Il ragionamento è di tipo *analitico*: il bisogno dell'utente viene scomposto in componenti singoli, e per ognuno si cerca la corrispondenza più precisa nel catalogo. Il limite sta nel fatto che non tutte le esigenze dell'utente hanno necessariamente una Key Feature corrispondente: può capitare che il catalogo non classifichi esplicitamente una caratteristica che per l'utente è importante.

Strumento 3 - Matching per profilo d'uso. Il secondo percorso di matching lavora in modo complementare al primo. Invece di scomporre le esigenze in requisiti puntuali, prende l'*intero profilo d'uso* dell'utente (una descrizione sintetica che il modello genera a partire dalla conversazione) e lo confronta direttamente con gli embedding dei nodi Model.

Il confronto è fra il quadro complessivo dell'utente e le descrizioni dei prodotti, e cattura affinità *globali* che al matching puntuale sfuggirebbero. Un modello descritto come «ideale per lavorazioni pesanti su terreni collinari» potrebbe non avere una singola key feature che corrisponda a «terreni ripidi con attrezzi pesanti», ma la sua descrizione complessiva è semanticamente vicinissima al profilo dell'utente.

Strumento 4 - Recupero delle specifiche complete. Una volta individuati i modelli candidati attraverso i percorsi di matching, il sistema deve presentarli all'utente con informazioni complete e verificate. Questo strumento riceve una lista di identificativi e restituisce le specifiche di ciascun modello: nome, famiglia, serie, immagine di prodotto.

La separazione fra matching e recupero delle specifiche è voluta. Gli strumenti 2 e 3 restituiscono gli identificativi dei candidati ma non le loro schede complete: caricare tutti gli attributi di ogni nodo già in fase di matching sarebbe inutile, dal momento che la maggior parte dei candidati iniziali viene scartata dal modello prima di arrivare all'utente. Tenendo i due passaggi distinti, il sistema paga il costo del recupero completo solo per i modelli che effettivamente verranno presentati.

Il suo ruolo è centrale anche per il requisito di **accuratezza**: l'output finale non è generato dal modello linguistico ma recuperato direttamente dal grafo. Ogni modello che compare nel suggerimento è un prodotto reale del catalogo, con dati verificati. È questo strumento a chiudere il cerchio della garanzia contro l'allucinazione: il modello linguistico decide *quali* prodotti suggerire, ma i dati che l'utente vede arrivano sempre e soltanto dal catalogo.

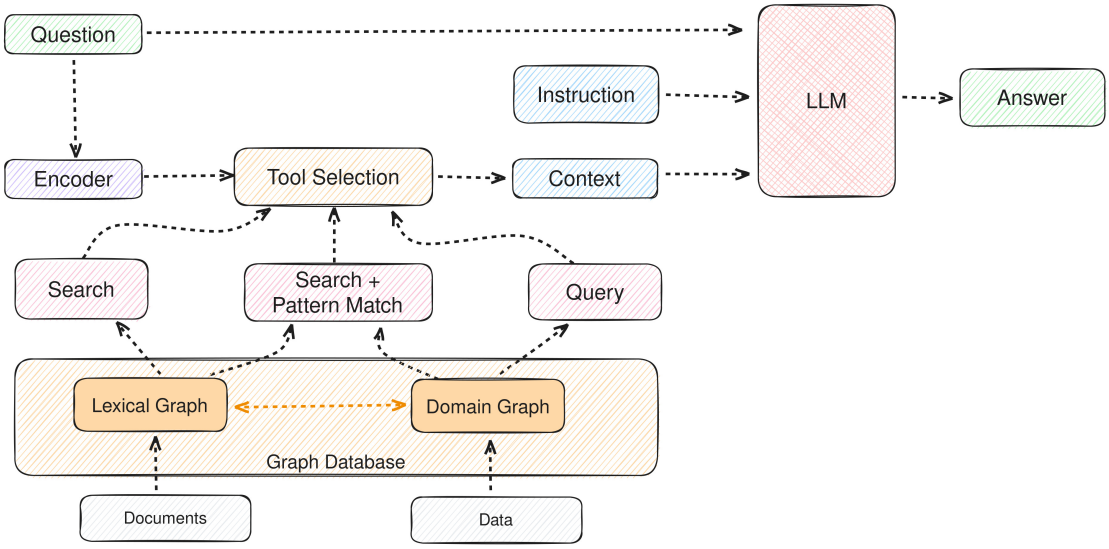

Figura 4.9: Panoramica dei pattern GraphRAG. Fonte: [26]

4.4.4 La strategia di matching duale

Avere due strumenti di matching separati (strumenti 2 e 3) non è una ridondanza: è una scelta progettuale precisa. I due percorsi colgono **aspetti diversi** della corrispondenza tra bisogni e prodotti, e usarli insieme produce suggerimenti più solidi di quelli che ciascuno darebbe da solo, un po' come nel principio di *triangolazione*,

dove fonti o metodi indipendenti convergono sullo stesso risultato aumentandone l'affidabilità.

Per capire perché servono entrambi, basta un esempio concreto. Si supponga che l'utente chieda un trattore per «lavorazioni pesanti su terreni collinari». Il matching per caratteristiche (strumento 2) trova Key Feature come «controllo di trazione per pendenze» e «sollevamento posteriore ad alta capacità», e da lì risale ai modelli che le possiedono: A, B, C. Il matching per profilo d'uso (strumento 3) confronta l'intero profilo con le descrizioni dei Model e trova B, C, D - dove D è un modello la cui descrizione complessiva è molto affine al profilo dell'utente, ma che non ha una Key Feature esplicitamente legata alle pendenze. Combinando i due insiemi il modello può confermare B e C (presenti in entrambi), includere D sulla base della forte affinità globale, e scartare A se il suo match puntuale era debole. Senza il secondo percorso, D sarebbe stato invisibile; senza il primo, non ci sarebbe stato modo di verificare la corrispondenza su singole caratteristiche specifiche.

A combinare i risultati non c'è un algoritmo deterministico: ci pensa il modello linguistico. Le istruzioni che riceve gli indicano dei criteri di massima, come ad esempio dare la precedenza ai match per caratteristiche quando sono netti, appoggiarsi di più al profilo d'uso quando quelli puntuali sono deboli, togliere dalla lista i modelli che ci sono finiti per vicinanza generica ma che in realtà non c'entrano con quello che l'utente ha chiesto. La scelta finale, però, resta sua: è il modello che decide cosa tenere e cosa scartare, alla luce di tutto quello che è emerso nella conversazione. Questa delega comporta un trade-off esplicito: si guadagna la capacità di gestire casi imprevisti con flessibilità, ma si perde il determinismo di un algoritmo a regole fisse, il cui output sarebbe identico a parità di input. Nel contesto di un dialogo di vendita il trade-off è accettabile, perché le combinazioni possibili di esigenze, formulazioni e contesti operativi sono troppo variabili per essere catturate da regole statiche, e perché l'output è comunque soggetto alla validazione dell'utente nella fase successiva.

Nella pratica questo produce un processo che si adatta da solo. Se entrambi i percorsi restituiscono gli stessi modelli, il suggerimento è solido e il modello lo presenta senza esitazioni. Se i risultati divergono, il modello valuta se la differenza indica che i due percorsi hanno colto aspetti complementari (e in questo caso probabilmente li unisce) oppure se uno dei due ha prodotto candidati poco convincenti (e allora li esclude). È il tipo di valutazione che richiede buon senso più che regole fisse, e che funziona meglio quando chi decide ha sotto mano l'intera storia della conversazione.

4.4.5 Il flusso conversazionale

Il dialogo fra utente e sistema si articola in tre fasi. Non è una sequenza rigida imposta all'utente ma una struttura pensata per **restringere lo spazio di ricerca**

un passo alla volta: si parte dall'intero catalogo, si arriva a una famiglia, e da lì a un gruppo ristretto di modelli. Il motivo è preciso: il catalogo Case IH comprende cinque famiglie, decine di serie e centinaia di modelli. Lanciare una ricerca semantica sull'intero catalogo produce un problema di *precisione*, non di efficienza computazionale: il vector search restituisce comunque i k nodi più vicini in tempo sub-lineare, ma quando l'indice contiene entità di famiglie radicalmente diverse - trattori, mietitrebbie, telescopici - gli embedding di prodotti con descrizioni genericamente simili (ad esempio «alta potenza», «comfort in cabina») possono risultare vicini nello spazio vettoriale pur appartenendo a categorie incompatibili. Filtrare prima per famiglia elimina questa interferenza e fa sì che i k candidati restituiti siano tutti confrontabili fra loro. La prima fase serve proprio a questo, ovvero delimitare il campo prima di attivare i meccanismi di matching semantico.

Fase 1 - Identificazione della famiglia di prodotto. La conversazione comincia con l'utente che descrive quello di cui ha bisogno. Le cinque famiglie di prodotto (Tractors, Combines, Balers, Telescopic Handlers, Front Loaders) corrispondono al primo livello di classificazione dell'offerta e coprono scenari d'uso fondamentalmente diversi. L'obiettivo di questa fase è capire quale famiglia si adatta meglio alle esigenze espresse, senza che l'utente debba sapere come è organizzato il catalogo, il che risponde al concetto di **riduzione del carico cognitivo**.

Il modello linguistico, attraverso il *system prompt*, è istruito a fare **domande discriminanti**: domande che aiutino a escludere una o più famiglie oppure a confermarne una, basandosi su ciò che distingue le famiglie l'una dall'altra. Le descrizioni di ogni famiglia sono incluse nel system prompt e vengono generate in automatico a partire dai dati reali del catalogo quando il sistema si inizializza: per ciascuna famiglia, il sistema produce una sintesi con gli scenari d'uso tipici, le caratteristiche funzionali e gli elementi che la differenziano dalle altre.

La scelta di porre **una sola domanda per volta** invece di presentare un questionario è voluta. Ogni risposta dell'utente può cambiare la direzione delle domande successive: se qualcuno dice «raccolta del foraggio», la famiglia Balers diventa subito la candidata principale e le domande dopo possono concentrarsi su conferme e dettagli, senza perdere tempo a esplorare famiglie già escluse. Un questionario fisso, al contrario, continuerebbe a porre domande rese inutili da risposte precedenti, ottenendo l'effetto opposto di quello desiderato.

Quando l'utente menziona una macchina che possiede già, il modello invoca lo strumento 1 (ricerca del prodotto di riferimento) e usa il profilo ottenuto per saltare le domande più scontate, concentrandosi su quelle che effettivamente servono a discriminare. In questo modo la conversazione converge più in fretta verso la famiglia giusta.

Fase 2 - Riepilogo e conferma. Quando il modello ritiene di avere abbastanza elementi per identificare la famiglia, genera un **recap**: una sintesi delle esigenze raccolte e della famiglia scelta, che viene presentata all'utente perché la confermi o la corregga.

Questa fase è un **punto di controllo esplicito** e risponde direttamente all'evidenza E3 della Tabella 3.1: nel configuratore tradizionale le scelte fatte durante la navigazione non vengono mai riepilogate né confrontate con le esigenze operative, e l'utente può arrivare a un modello senza alcuna garanzia che corrisponda davvero a quello che gli serve.

Il costo di qualche domanda in più, nel caso in cui il riepilogo non sia corretto, è irrilevante rispetto al costo di un suggerimento costruito su premesse sbagliate: il sistema cercherebbe nella famiglia sbagliata, restituirebbe modelli inadatti e perderebbe la fiducia dell'utente. Il recap funziona quindi come **validazione a doppio senso**: l'utente verifica che il sistema abbia capito bene, e il sistema si assicura di avere il via libera prima di procedere con la ricerca.

Fase 3 - Dal profilo ai modelli. Con la conferma dell'utente, il modello lancia gli strumenti di matching (strumenti 2 e 3), ne combina i risultati secondo la strategia descritta nella Sezione 4.4.4, recupera le specifiche complete dei candidati tramite lo strumento 4 e restituisce la lista dei modelli suggeriti. Ogni modello nell'output è un prodotto reale del catalogo, con gamma, nome, serie e immagine che provengono direttamente dal grafo.

Se nessun modello risulta compatibile con le esigenze espresse, il sistema lo dice apertamente e propone all'utente di eseguire una nuova ricerca, il che è una scelta deliberata: il sistema non deve mai forzare un suggerimento inadatto pur di dare una risposta. Meglio un'assenza motivata di risultati che un suggerimento incoerente.

4.5 Integrazione con il CPQ

Il quarto ed ultimo problema progettuale riguarda il punto di contatto tra il modulo di Assisted Selling e il flusso di configurazione esistente. L'analisi del Capitolo 2 ha evidenziato che il CPQ di Case IH gestisce già in modo efficace la fase di personalizzazione tecnica del prodotto: regole di compatibilità, calcolo del prezzo, generazione del preventivo. La criticità identificata non risiede nel CPQ ma nella fase che lo precede, ovvero la selezione del modello base. Il modulo di Assisted Selling interviene esclusivamente in questa fase, e la sua integrazione con il CPQ deve rispettare un principio fondamentale: **non alterare un processo che già funziona correttamente**.

L'output del modulo è una **lista di modelli configurabili**, ognuno identificato tra gli altri da nome, gamma e range. Il configuratore online di Case IH è un'applicazione web esterna, alla quale CoolPIM non ha accesso se non attraverso la struttura pubblica degli URL: ogni modello configurabile corrisponde a un indirizzo composto da gamma, range e nome del modello. Questa è l'unica interfaccia di integrazione disponibile, e il modulo la sfrutta componendo l'URL corrispondente a ciascun modello suggerito, così che l'utente possa passare dal suggerimento alla configurazione con un clic, senza che le logiche di compatibilità e pricing vengano toccate in alcun modo.

Capitolo 5

Implementazione

Questo capitolo racconta come le scelte progettuali descritte nel capitolo precedente sono state tradotte in codice. Non si tratta di una rassegna esaustiva di ogni funzione scritta, ma piuttosto di una discussione ragionata sulle scelte implementative più significative, sulle difficoltà incontrate e sulle soluzioni adottate per risolverle.

5.1 Struttura del modulo nel monorepo

La prima decisione, apparentemente banale, ha riguardato dove e come strutturare il codice. CoolPIM è sviluppato in TypeScript su runtime Node.js e organizzato come monorepo con tre pacchetti principali - il server (API GraphQL e REST), il frontend React e un proxy di autenticazione - gestiti tramite npm workspaces. Poiché il modulo ha bisogno di accedere al grafo Neo4j, allo stato delle gerarchie di prodotto, alle impostazioni aziendali, al sistema di autenticazione, replicare tutte queste dipendenze in un servizio esterno avrebbe significato duplicare infrastruttura e mantenere sincronizzazioni fragili, senza guadagnare nulla in cambio. Il modulo è stato quindi sviluppato come componente interno del monorepo, suddiviso fra **server** e **ui** seguendo la stessa struttura modulare già consolidata nel progetto.

Lato server, la logica è organizzata in cinque sottodirectory all'interno di `server/src/assistedSelling`: la cartella `ai/` dedicata all'interazione con il modello linguistico (ricerca vettoriale, matching, estrazione delle caratteristiche), `database/` per la persistenza dei dati di log su database relazionale, `events/` per il sistema di eventi in tempo reale che comunica col frontend durante le operazioni più lunghe, `scripts/` per gli script di indicizzazione vettoriale e `utils/` per le funzioni ausiliarie.

Lato frontend, il modulo risiede in `ui/src/_new/modules/assisted-selling/`, la directory riservata ai nuovi sviluppi dell'interfaccia. La struttura riflette la separazione fra le due fasi dell'esperienza utente: il componente `AssistedSellingChat`

gestisce l'interfaccia conversazionale, `AssistedSellingResults` la presentazione dei modelli raccomandati, ed entrambi sono orchestrati da un pannello contenitore (`AssistedSellingPanel`) che governa le transizioni di stato. Una cartella `APIs/` centralizza le chiamate REST verso il server, e una cartella `styles/` raccoglie gli stili condivisi del modulo.

5.2 Indipendenza dal provider AI

Nella Sezione 4.4 si è motivata la scelta di rendere il modulo indipendente dal provider del modello linguistico: vincolarsi ad un singolo fornitore, tenendo conto della velocità a cui il settore evolve, costituirebbe un rischio concreto per la manutenibilità del sistema. Il meccanismo è costruito attorno a una classe astratta che dichiara i metodi fondamentali: gestione della conversazione, calcolo degli embedding, traduzione dei messaggi di tool call nel formato del provider, e ricerca web approfondita. Ogni implementazione concreta deve fornire questi metodi, adattandoli alle API del proprio fornitore. La scelta di una classe astratta permette di integrare nuovi provider in futuro senza modificare la logica di business, ma semplicemente implementando i metodi richiesti anche per i nuovi provider. Il Listato 5.1 mostra la firma della classe e i contratti che ciascun provider deve rispettare.

```
export abstract class AIProvider {
    protected defaultChatModel: string;
    protected defaultEmbeddingModelConfig: EmbeddingConfig;

    abstract getEmbeddingDim(): Promise<number>;
    abstract getEmbedding(text: string):Promise<EmbeddingResponse
   >;
    abstract getAiEmbeddingPrefix(): string;

    abstract getChatCompletion(params: {
        messages: ChatMessage[];
        tools?: ToolRequest[];
        temperature?: number;
    }): Promise<ChatResponse>;
    abstract mapToolMessages(answer: ChatResponse
    ): Promise<{
        toolCallsMessage: ChatMessage;
        toolResultsMessages: ChatMessage[];
    }>;
    abstract runDeepResearch(
        query: string,
    ): Promise<DeepResearchUsageResult | null>;
}
```

Listing 5.1: Classe astratta `AIProvider`: contratto comune a tutti i provider.

Un aspetto meno ovvio ma importante è la gestione della coesistenza di embedding generati da provider diversi. Come discusso nella Sezione 4.3, il sistema mantiene indici vettoriali separati per i diversi tipi di nodo; a questo si aggiunge una separazione per provider. Ogni implementazione espone un prefisso identificativo che viene utilizzato nella denominazione delle proprietà e degli indici vettoriali in Neo4j. In pratica, gli embedding di OpenAI e quelli di Cohere, così come quelli di Ollama, possono convivere sugli stessi nodi, ognuno con il proprio indice, senza interferenze. Questo meccanismo è nato per supportare scenari di **A/B testing**, ovvero confrontare la qualità dei suggerimenti ottenuti con modelli di embedding diversi, senza dover rigenerare l'intero dataset ad ogni cambio.

Ad oggi sono state sviluppate tre implementazioni concrete, riassunte nella Tabella 5.1.

Provider	**Chat Model**	**Embedding Model**	**Vector Dim.**
OpenAI	GPT-4o	text-embedding-3-small	1536
AWS Bedrock	Claude 4.5 Sonnet[1]	Cohere Embed v4	1536
Ollama	LLaMA 3.1	nomic-embed-text	768

Tabella 5.1: Provider AI implementati e relativi modelli di default.

La scelta dei tre provider non è casuale e riflette tre scenari di deployment distinti. OpenAI è stato il primo ad essere integrato: il suo ecosistema è il più maturo per quanto riguarda il function calling, e *GPT-4o* offre un buon equilibrio fra qualità delle risposte e velocità di esecuzione; il modello di embedding *text-embedding-3-small*, pur non essendo il più performante della famiglia, produce vettori di qualità adeguata a un costo contenuto. *AWS Bedrock* risponde a un'esigenza di deployment enterprise: i dati non transitano attraverso API di terze parti ma restano all'interno dell'infrastruttura AWS del cliente, un vincolo frequente in contesti aziendali; *Claude 4.5 Sonnet* è stato scelto per la qualità del ragionamento, *Cohere Embed v4* per le sue capacità multilingue, rilevanti in un contesto dove il catalogo è in inglese ma l'utente potrebbe interagire in altre lingue. *Ollama*, infine, consente l'esecuzione interamente locale: nessun dato lascia la macchina, non ci sono costi di API, e l'ambiente è ideale per lo sviluppo e il testing. La differenza di dimensionalità degli embedding (768 per *nomic-embed-text* contro i 1536 degli altri due) ha validato concretamente il meccanismo di separazione degli indici per provider descritto poco sopra.

La selezione del provider avviene a runtime attraverso una factory (Listato 5.2) che ispeziona le variabili d'ambiente e istanzia il provider corrispondente, con una

[1]Tramite inference profile ARN configurabile, il modello può variare.

priorità predefinita (Bedrock, poi OpenAI, poi Ollama). L'istanza è un **singleton**: viene creata una sola volta e condivisa per tutta la durata del processo. In termini pratici, passare da un provider all'altro richiede soltanto di modificare la configurazione d'ambiente, senza toccare una riga di codice applicativo. Questa scelta implementativa ha due vantaggi principali. Da un lato, la factory disaccoppia il codice applicativo dalle implementazioni concrete dei provider, centralizzando la logica di selezione e rendendo il sistema facilmente estendibile: l'aggiunta di un nuovo provider richiede soltanto l'implementazione dell'interfaccia comune e l'aggiornamento della factory. Dall'altro lato, il pattern singleton garantisce che il provider venga inizializzato una sola volta, evitando la creazione ripetuta di client verso i servizi di inferenza e assicurando che tutta l'applicazione operi con la stessa configurazione. Nel loro insieme, i due pattern permettono di gestire in modo robusto la variabilità dei provider mantenendo il codice semplice, configurabile e privo di dipendenze dirette da specifici servizi di inferenza.

```
let aiProviderInstance: AIProvider | null = null;
export function getAIProviderInstance() {
    if (aiProviderInstance) return aiProviderInstance;
    if (BEDROCK_ACCESS_KEY_ID && BEDROCK_SECRET_ACCESS_KEY) {
        aiProviderInstance = new BedrockProvider({ ... });
        return aiProviderInstance;
    }
    if (OPENAI_API_KEY) {
        aiProviderInstance = new OpenAIProvider({ ... });
        return aiProviderInstance;
    }
    if (!!OLLAMA_URI) {
        aiProviderInstance = new OllamaProvider({...});
        return aiProviderInstance;
    }
    throw new Error('No valid AI provider configured.');
}
```

Listing 5.2: Factory di selezione del provider AI.

La priorità Bedrock-OpenAI-Ollama riflette l'ordine di preferenza per un ambiente di produzione. Bedrock è privilegiato perché, in uno scenario enterprise, la residenza dei dati all'interno dell'infrastruttura del cliente è il vincolo più stringente; se le credenziali AWS non sono configurate, il sistema ricade su OpenAI, che offre la migliore esperienza di function calling; Ollama interviene come ultima risorsa, tipicamente negli ambienti di sviluppo locale dove nessuna API esterna è disponibile.

L'implementazione Bedrock merita inoltre una nota a parte per quanto riguarda la ricerca web, lo strumento 1 descritto nella Sezione 4.4.3. La ricerca web è l'unica funzionalità del modulo la cui realizzazione si differenzia in modo netto fra i due provider principali, e la ragione è infrastrutturale. OpenAI prevede

la ricerca web nativamente, attivabile tramite le proprie Responses API: il tool `web_search_preview` consente al modello stesso di navigare il web, analizzare le fonti e sintetizzare un risultato strutturato. Bedrock non offre niente di simile: i modelli ospitati non hanno accesso alla rete e non dispongono di strumenti di browsing integrati. La soluzione (basata sul protocollo **Model Context Protocol**, MCP) è stata appoggiarsi al servizio **Amazon Bedrock Agents**, un componente separato dell'ecosistema AWS che consente di creare agenti con accesso a fonti esterne. Un agente pre-configurato nella console AWS riceve la query sul prodotto di riferimento e, sfruttando il proprio accesso al web, restituisce un profilo d'uso strutturato del prodotto cercato (scenari tipici, compiti abituali, posizionamento di mercato e un livello di accuratezza numerico). L'invocazione (Listato 5.3) avviene tramite il client `BedrockAgentRuntimeClient` con un semplice comando `InvokeAgentCommand`, e il risultato viene mappato nello stesso formato usato da OpenAI, per garantire il comportamento agnostico dei provider citato in precedenza. Questo profilo viene poi utilizzato dal modello linguistico per formulare domande più mirate, come si vedrà nella Sezione 5.4.

```
1  async runDeepResearch(query: string): Promise<
       DeepResearchUsageResult> {
2          let deepSearchResponse = '';
3          try {
4              const command = new InvokeAgentCommand({
5                  agentId: AGENT_ID,
6                  agentAliasId: AGENT_ALIAS_ID,
7                  sessionId: randomUUID(),
8                  inputText: query,
9              });
10             const response = await this.agentClient.send(command);
11             if (response.completion) {
12                 for await (const chunk of response.completion) {
13                     if (chunk.chunk?.bytes) {
14                         deepSearchResponse += new TextDecoder().
       decode(chunk.chunk.bytes);
15                     }
16                 }
17             }
18             return JSON.parse(deepSearchResponse);
19         } catch (err) {
20             throw new AIProviderError(`Bedrock deep research error
       : ${err.message}`);
21         }
22 }
```

Listing 5.3: Invocazione dell'agente Bedrock per la ricerca web.

5.3 Pipeline di indicizzazione vettoriale

Nella Sezione 4.3 si è spiegato perché il catalogo deve essere reso confrontabile con il linguaggio naturale dell'utente, quali contenuti indicizzare e con quale metrica misurare la vicinanza semantica. Questa sezione descrive come quella pipeline è stata implementata: due script dedicati, concepiti per essere lanciati periodicamente o in occasione di aggiornamenti significativi del catalogo, si occupano di precalcolare tutti i vettori e di creare gli indici necessari.

5.3.1 Indicizzazione di prodotti e key feature

Il primo script calcola gli embedding per le due tipologie di nodi identificate nella Sezione 4.3: i modelli di prodotto e le key feature. Per ciascun nodo, il testo da vettorializzare viene ottenuto concatenando i campi più descrittivi disponibili (*standard_features* e *additional_description* per i prodotti, *title* e *description* per le key feature) seguendo il criterio stabilito in fase di progettazione: indicizzare gli attributi che l'utente userebbe se gli chiedessero di spiegare a cosa serve il prodotto. Solo i nodi di tipo **Model** vengono indicizzati fra quelli della Product Offering: i modelli sono le entità che il sistema vuole raccomandare e che l'utente può successivamente configurare nel CPQ, mentre i livelli superiori della gerarchia (gamma, range, subrange) sono contenitori organizzativi la cui presenza nei risultati vettoriali potrebbe rallentare il processo di matching e inquinare i conseguenti suggerimenti. Per ogni nodo, lo script invoca il servizio di embedding del provider AI configurato, ottiene il vettore risultante e lo salva come proprietà del nodo stesso in Neo4j. Al termine, viene creato un indice vettoriale HNSW con funzione di *cosine similarity*, che consente poi ricerche per prossimità vettoriale in tempo sub-lineare [18], essenziale per rispettare il requisito di **prestazioni** (Tabella 3.3). L'uso del prefisso identificativo del provider, descritto nella sezione precedente, fa sì che lo stesso script possa essere eseguito con provider diversi senza sovrascrivere gli indici esistenti.

```
1 const { embedding, totalTokens } =
2     await ai.getEmbedding(node.combinedText);
3
4 await neo4jClient.executeQuery(`
5     MATCH (n:${type} {id: $nodeId})
6     SET n.${ai.getAiEmbeddingPrefix()}_embedding_${fields} =
    $embedding,
7         n.${ai.getAiEmbeddingPrefix()}_last_updated_${fields} =
    timestamp()`,
8         { nodeId: node.id, embedding }
9 );
```

Listing 5.4: Calcolo e salvataggio dell'embedding per un nodo.

Il Listato 5.5 mostra la query Cypher che implementa la ricerca per prossimità, invocando la procedura nativa `db.index.vector.queryNodes` di Neo4j sull'indice HNSW.

```
CALL db.index.vector.queryNodes($indexName, $topK, $embedding)
YIELD node, score
WHERE score > $scoreThreshold
RETURN node.id AS id, node.code AS code, score
ORDER BY score DESC
```

Listing 5.5: Query Cypher di ricerca vettoriale approssimata.

5.3.2 Generazione delle descrizioni di gamma

Il secondo script affronta un concetto meno scontato. I nodi gamma rappresentano le famiglie di prodotto, le stesse cinque descritte nel Capitolo 2 (Tractors, Combines and Headers, Balers, Telescopic Handlers, Front Loaders). Il modulo di Assisted Selling ha bisogno che il modello linguistico *conosca* ciascuna famiglia in modo approfondito per poter orientare le domande verso quella più adatta, come richiesto nella fase 1 del flusso conversazionale (Sezione 4.4.5). Per rendere più precisa l'identificazione della gamma, lo script recupera tutti i discendenti - range, subrange, modelli - con le relative schede tecniche, e chiede al modello di sintetizzare una descrizione coerente della famiglia. Quando il numero di discendenti è elevato (oltre venti), i dati vengono suddivisi in blocchi, ciascuno sintetizzato separatamente; una seconda invocazione unifica le sintesi parziali in un testo unico. Il risultato viene salvato come proprietà del nodo gamma in Neo4j insieme al suo embedding, come mostra il Listato 5.6: la descrizione testuale, il vettore e il timestamp di aggiornamento coesistono sullo stesso nodo, ciascuno con il prefisso del provider attivo.

```
const { content: description } = await synthesizeFamilyDescription
    ( gamma.name, gamma.descendants, ai);
const prefix = ai.getAiEmbeddingPrefix();
const { embedding } = await ai.getEmbedding(description);

await neo4jClient.executeQuery(`
    MATCH (gamma:Product_offering {_id: $gammaId})
    SET gamma.\`${prefix}_family_llm_description\` = $description,
        gamma.\`${prefix}_family_llm_description_embedding\` =
    $embedding,
        gamma.\`${prefix}_family_llm_description_last_updated\` =
    timestamp()`, { gammaId: gamma.id, description, embedding }
);
```

Listing 5.6: Generazione e salvataggio della descrizione sintetica di una gamma.

Queste descrizioni non servono solo come contesto statico: vengono iniettate dinamicamente nel *system prompt* ad ogni nuova conversazione, come si vedrà nella Sezione 5.4. Il modello opera così con informazioni sempre allineate al catalogo reale, senza necessità di riaddestramento o aggiornamento manuale. Nella pratica è un modo per ottimizzare il processo di identificazione della famiglia e renderlo più **accurato**, rispondendo al requisito in Tabella 3.3.

5.4 Ingegneria del system prompt

Se l'astrazione del provider e la pipeline di indicizzazione sono la struttura portante del sistema, il system prompt ne è il cervello. Il flusso conversazionale in tre fasi descritto nella Sezione 4.4.5 (identificazione della famiglia, riepilogo e conferma, matching e raccomandazione) si traduce in un prompt che istruisce l'LLM e la cui definizione ha richiesto numerose iterazioni. Ciascuna iterazione è stata motivata dall'osservazione di comportamenti indesiderati in alcuni test effettuati: ad esempio domande troppo lunghe e poco precise, l'individuazione di una famiglia a partire da informazioni discriminanti insufficienti, oppure modelli raccomandati con dati incompleti. Il risultato è un insieme di vincoli che non descrivono *cosa* l'assistente deve fare (che è già definito nel disegno del flusso) ma *come* deve farlo, correggendo le tendenze naturali del modello linguistico descritte successivamente e che si rivelano controproducenti nel contesto specifico della vendita assistita.

Il prompt stabilisce innanzitutto il ruolo - un esperto di vendita agricola amichevole e diretto - e impone tre vincoli strutturali, ciascuno nato da un problema concreto. Il primo è la regola della **domanda singola**, ovvero una sola domanda per messaggio. Senza questo vincolo, il modello poneva due o tre domande in un unico messaggio; l'utente rispondeva tipicamente solo all'ultima, e le informazioni delle precedenti andavano perse. Il secondo vincolo riguarda il **formato**: domande brevi, senza preamboli né spiegazioni del perché si stia chiedendo qualcosa. "*Slopes or flat terrain?*" e non "*Let me ask you about the terrain you work on. This will help me understand which product line fits best for your situation...*". Questa regola si è resa necessaria perché si è osservato che il modello, lasciato a sé stesso, tende a produrre domande prolisse con giustificazioni incorporate, un comportamento naturale per un assistente generico ma controproducente in un contesto di vendita dove essere concisi è un pregio. Il terzo vincolo vieta qualsiasi **riferimento economico**. Il sistema non dispone di informazioni sui prezzi, che compete esclusivamente al CPQ e non all'assistente.

La sezione relativa alle famiglie di prodotto disponibili non è statica: ad ogni nuova conversazione, le descrizioni sintetiche generate dallo script descritto nella Sezione 5.3.2 vengono iniettate nel prompt, racchiuse in un tag XML. Il modello opera così con informazioni sempre aggiornate sul catalogo senza che il prompt

debba essere riscritto manualmente. È la soluzione concreta al problema evidenziato nel Capitolo 2: le cinque famiglie hanno caratteristiche e scenari d'uso che il sistema deve conoscere per porre le domande giuste e per identificare la famiglia giusta. Il Listato 5.7 mostra l'estratto del prompt in cui le descrizioni vengono iniettate: l'array `gammas`, costruito a partire dalle proprietà Neo4j generate dallo script di Sezione 5.3.2, viene serializzato all'interno di un tag XML che il modello tratta come fonte autoritativa.

```
1 <available-hierarchies>
2     ${JSON.stringify(gammas)}
3 </available-hierarchies>
4
5 Each gamma has a NAME and DESCRIPTION that define its typical use
     cases, operating environments, and key features.
6 These descriptions are your ONLY source of truth for what each
     gamma is designed for.
```

Listing 5.7: Iniezione dinamica delle descrizioni di gamma nel system prompt.

Due workflow separati gestiscono poi il flusso conversazionale. Quando l'utente menziona un prodotto di riferimento - un caso frequente, come evidenziato nell'analisi del profilo utente (Sezione 3.2) - il prompt definisce una scala di confidenza a quattro livelli che regola quanto il modello deve affidarsi ai dati provenienti dalla ricerca web: con confidenza alta (0.8–1.0), i dati costituiscono un punto di partenza solido per domande mirate; con confidenza molto bassa (sotto 0.3), vengono ignorati del tutto perché troppo inaffidabili. La scelta di una scala graduata, anziché di un semplice valore binario, nasce dall'osservazione che dati mediamente affidabili possono suggerire direzioni da esplorare, e non per forza conclusioni da assumere. Quando invece non esiste un prodotto di riferimento, il modello deve generare domande *discriminanti* pensate per escludere una o più famiglie o confermarne altre, sulla base delle differenze tra le loro descrizioni. Il prompt vieta esplicitamente le domande generiche ("*Tell me about your farm*") e richiede che ogni domanda sia riconducibile a una caratteristica differenziante specifica delle famiglie. Questa regola è legata direttamente alla criticità identificata nel Capitolo 2: l'assenza di un meccanismo che mappi i bisogni operativi dell'utente sulla struttura del catalogo. Le domande discriminanti sono la traduzione conversazionale di quel meccanismo mancante.

Un passaggio cruciale, spesso trascurato, è il **riepilogo** delle esigenze raccolte prima di procedere alla raccomandazione. Nel prompt, questa fase è esplicitamente richiesta: una volta che il modello ha identificato una famiglia e raccolto informazioni sufficienti, deve presentare all'utente un riepilogo sintetico delle risposte chiave e chiedere conferma ("Is this recap correct?"). Solo dopo l'accettazione esplicita da parte dell'utente il sistema procede alla selezione dei modelli. Questa scelta nasce da due esigenze: da un lato, evitare che errori di interpretazione o risposte ambigue

portino a raccomandazioni non pertinenti; dall'altro, responsabilizzare l'utente, che può correggere eventuali fraintendimenti prima che il sistema prenda decisioni. Nei test effettuati, l'assenza di questa fase portava spesso a suggerimenti percepiti come arbitrari o "magici" dall'utente, mentre il recap esplicito aumenta la trasparenza e la fiducia nell'assistente. Il Listato 5.8 mostra l'estratto del prompt che governa questa fase.

```
1 Once you have clearly identified ONE gamma that fits the user
     needs:
2
3 Provide:
4   - A CLEAR recap of the user needs (briefly summarize the key
     discriminative answers)
5   - A CLEAR CITATION of the UNIQUE chosen gamma
6     (just the gamma name, NO explanation)
7
8 Then ask: "Is this recap correct?"
```

Listing 5.8: Estratto del prompt: istruzioni per il riepilogo e la conferma delle esigenze raccolte.

La fase finale dell'interazione è legata alla raccomandazione dei modelli. Il prompt istruisce il modello a combinare i risultati dei due canali di matching non attraverso un'operazione insiemistica deterministica, ma attraverso un giudizio qualitativo, la cui logica è stata descritta nella Sezione 4.4.4: prediligere i match espliciti quando sono forti, appoggiarsi di più ai match per profilo quando quelli espliciti scarseggiano. Far decidere al modello linguistico sfrutta una delle sue capacità più distintive: la comprensione del contesto.

Il prompt guida infine le transizioni di stato nell'interfaccia utente attraverso token convenzionali inseriti nel testo della risposta. La scelta dei token nasce da un vincolo architetturale: poiché l'output del modello è un flusso di testo libero, il frontend ha bisogno di segnali chiari per riconoscere a che punto del processo ci si trova ed eventualmente attivare le transizioni di stato, come per esempio il passaggio dalla visualizzazione della conversazione alla visualizzazione dei prodotti suggeriti. Il segnale **%%%** indica che la conversazione è conclusa e l'utente ha accettato il riepilogo; il token **###**, inviato dal frontend al modello, richiede le raccomandazioni finali. Il formato di output delle raccomandazioni è un array JSON racchiuso in tag XML. I tag fanno da delimitatori, necessari perché senza di essi il modello tende a mescolare testo con dati strutturati, rendendo il parsing inaffidabile.

I vincoli sulla struttura dei suggerimenti in output sono chiari ed esplicitati nel prompt, come mostra il Listato 5.9: ogni modello raccomandato deve contenere tutti i campi obbligatori, popolati esclusivamente con dati reali provenienti dal grafo, e i modelli con dati incompleti vengono scartati. Questo corrisponde alla traduzione procedurale del requisito di **accuratezza** (Tabella 3.3).

```
COMPLETENESS REQUIREMENTS - NON-NEGOTIABLE:
Each suggested model MUST contain ALL of the following
fields: id, code, name, type, rangeName, productImage.

VERIFICATION RULES:
  No field can be missing, null, empty,
    or contain placeholder values.
  You MUST NOT invent or fabricate any values.
  Every field MUST come from actual model data
    returned by the specs retrieval step.
  If the specs retrieval returns incomplete data
    for any model, REJECT that model entirely.
  Only return models that have 100% complete data
    for all required fields.
```

Listing 5.9: Estratto del prompt: vincoli di completezza sui modelli raccomandati.

5.5 I tool utilizzabili dal modello linguistico

I quattro strumenti (tools) descritti nella Sezione 4.4.3 sono il meccanismo concreto attraverso cui, tramite function calling, il modello linguistico interagisce con il grafo Neo4j e con i servizi esterni. Ciascun tool è definito all'interno del router con un formato generico e indipendente dal provider: un nome, una descrizione in linguaggio naturale, uno schema JSON dei parametri accettati e una funzione di implementazione che contiene la logica effettiva. Queste definizioni vengono passate al metodo di completamento della conversazione `getChatCompletion()` esposto dal provider attivo; al suo interno, ogni implementazione concreta le traduce nel formato richiesto dalla propria API - OpenAI le incapsula come `function` all'interno di un oggetto `tool`, Bedrock le converte in `toolSpec` con uno schema di input dedicato, Ollama adotta una convenzione ancora diversa. Il percorso inverso è altrettanto importante: quando il modello linguistico decide di invocare un tool, la risposta contiene una **tool call** nel formato nativo del provider. Il metodo `mapToolMessages`, implementato da ciascun provider, normalizza queste risposte in un formato comune, così che il ciclo conversazionale possa gestire le chiamate ai tool e i loro risultati senza sapere quale provider le ha originate. Una volta invocata la funzione di implementazione associata a ciascun tool richiesto, vengono raccolti i risultati e reinseriti nella conversazione come messaggi di tipo `tool`, messaggi che il modello linguistico utilizza come contesto per formulare la risposta successiva. Il modello decide autonomamente quando invocare ciascun tool, con quali parametri e come interpretarne i risultati. Il Listato 5.10 mostra la struttura di un tool a titolo esemplificativo: lo schema dei parametri segue la specifica JSON Schema ed è identico per tutti i provider.

```
{
    name: 'get_key_features_from_characteristics',
    description: 'Return key features matched to the user '
        + 'requirements, enriched with product models '
        + 'linked to each key feature.',
    parameters: {
        type: 'object',
        properties: {
            characteristics: {
                type: 'array',
                items: { type: 'string' },
                description: 'List of user requirements.',
            },
            gammaName: {
                type: 'string',
                description: 'Target gamma name.',
            },
        },
        required: ['characteristics', 'gammaName'],
    },
    implementation: async (args) => {
        return await getKeyFeaturesFromCharacteristics(
            neo4jClient, args.characteristics,
            args.gammaName, storage, user, ai,
        );
    },
}
```

Listing 5.10: Definizione del tool di matching per caratteristiche chiave.

Il ciclo di esecuzione che guida l'interazione fra LLM e strumenti è riportato nel Listato 5.11. La funzione invoca il modello in un ciclo: se la risposta contiene una o più **tool call**, il metodo `mapToolMessages` del provider attivo le traduce in messaggi nel formato comune, il sistema esegue le funzioni richieste e reinserisce i risultati nella conversazione; il ciclo riparte e il modello riceve il contesto aggiornato. Quando il modello non richiede più strumenti, il ciclo termina e la risposta testuale viene restituita all'endpoint. È importante notare come quest'ultimo scenario ricopre anche il caso, in realtà il più frequente, della conversazione ordinaria: quando il modello pone una domanda o presenta il riepilogo, non richiede strumenti, e la funzione compie una sola iterazione restituendo direttamente la risposta dell'assistente. Il ciclo multiplo si attiva solo nei momenti in cui il modello ha bisogno di interrogare il grafo o il web mediante l'utilizzo dei tools.

```
export async function runAIConversation(
    conversation: ChatMessage[],
    ai: AIProvider,
    tools?: ToolParam[],
```

```
): Promise<ChatResponse> {
    let totalTokens = 0;
    while (true) {
        const response = await ai.getChatCompletion({
            messages: conversation, tools,
        });
        totalTokens += response.totalTokens ?? 0;
        if (!response.toolCalls?.length) {
            return { content: response.content, totalTokens };
        }
        const { toolCallsMessage, toolResultsMessages } =
            await ai.mapToolMessages(response);
        conversation.push(toolCallsMessage);
        conversation.push(...toolResultsMessages);
    }
}
```

Listing 5.11: Ciclo di esecuzione tool.

La particolarità implementativa più significativa riguarda il primo tool, incaricato di effettuare la **web search**. Poiché l'operazione può richiedere diversi secondi, il tool pubblica eventi di stato (avvio, progresso, completamento, errore) su un bus di eventi interno di Node.js, che il canale SSE (Server-Sent Events) propaga in tempo reale al frontend (descritto nello specifico nella Sezione 5.8.4). L'utente vede apparire in chat un indicatore di ricerca in corso e il suo esito, anziché un'attesa muta – una soluzione dettata dal requisito di **prestazioni** (Tabella 3.3): quando la risposta non può essere istantanea, l'informazione sullo stato dell'operazione diventa essa stessa parte della risposta.

Il secondo tool, che realizza il **matching per key feature**, traduce in codice il primo dei due percorsi della strategia duale (Sezione 4.4.4). Ciascuna caratteristica ricevuta dal modello viene trasformata in un embedding e confrontata con gli indici dei nodi *key feature* in Neo4j tramite la ricerca vettoriale del Listato 5.5, con una soglia di cosine similarity fissata a 0.7. Questo è un valore calibrato empiricamente durante i test effettuati: soglie più basse (0.5-0.6) restituivano key feature poco pertinenti che inquinavano i risultati, soglie più alte (0.8-0.9) escludevano key feature lecite che magari erano espresse in termini diversi da quelle del catalogo. Per ogni key feature individuata, una query Cypher (Listato 5.12) risale le relazioni del grafo (gli stessi archi `HAS_LINK_TO` descritti nella Sezione 4.2) per trovare i modelli che la possiedono, filtrando per la famiglia identificata. Il risultato che torna al modello è un elenco di feature con punteggi di similarità e modelli collegati, ideali per capire quali prodotti soddisfano ciascun requisito specifico.

```
MATCH (kf:Key_feature)
WHERE kf.id IN $kfIds
OPTIONAL MATCH (po:Product_offering)-[:HAS_LINK_TO]->(kf)
    WHERE po.type = "MODEL"
```

```
5       AND po.gammaId = $gammaCode
6       AND po.disabled = false
7 RETURN kf.id AS kfId,
8        kf.title AS title,
9        kf.description AS description,
10       collect(DISTINCT po.id) AS modelIds
```

Listing 5.12: Query Cypher semplificata che risale dai nodi Key Feature ai modelli di prodotto collegati, filtrando per gamma.

Il terzo tool realizza il secondo percorso della strategia duale: il **matching per profilo d'uso**. Riceve una descrizione complessiva delle esigenze dell'utente e la confronta direttamente con gli embedding dei nodi modello. A differenza dello strumento precedente, che ragiona per feature atomiche, questo cattura corrispondenze più generali legate ad un quadro d'insieme delle esigenze: un modello può risultare rilevante per il contesto generale dell'utente anche senza corrispondere a nessuna key feature specifica. I due canali sono complementari per costruzione, e il modello linguistico li combina secondo le regole definite nel system prompt (Sezione 5.4).

L'ultimo tool è il più semplice: riceve una lista di identificativi di modello e restituisce le **specifiche complete** per ciascuno (nome, codice, range, immagine). Esiste perché i risultati dei due canali precedenti possono contenere informazioni parziali; prima di ritornare i suggerimenti finali, il modello è istruito a verificare la completezza di ogni campo e a richiedere le specifiche mancanti, e per questo motivo si rivela centrale per il requisito di **accuratezza** (Tabella 3.3). L'output finale in questo modo non è generato dal modello linguistico ma recuperato direttamente dal grafo, limitando definitivamente il problema di *allucinazione* discussa nella Sezione 4.4.2.

5.6 Persistenza e osservabilità

Ogni conversazione e ogni raccomandazione prodotta dal sistema vengono registrate su PostgreSQL. La scelta di un database relazionale per questo tipo di dati, intrinsecamente semi-strutturati, potrebbe sembrare controintuitiva, ma è motivata da ragioni pratiche: PostgreSQL era già parte dello stack infrastrutturale (utilizzato da Keycloak per la gestione dell'autenticazione), e il tipo JSONB offre la flessibilità necessaria per memorizzare messaggi di struttura variabile senza sacrificare la possibilità di query. Due tabelle, collegate da una relazione uno-a-molti, catturano rispettivamente la conversazione e i suoi risultati. La prima (Listato 5.13) memorizza l'intero storico dei messaggi (inclusi quelli di sistema e di tool), i metadati della sessione (provider AI, token consumati, flag di completamento) e i timestamp di creazione e aggiornamento. La seconda registra l'esito del processo di suggerimento

con un livello di dettaglio pensato per l'analisi a posteriori: i modelli suggeriti, le caratteristiche estratte, i parametri di ricerca utilizzati (soglia di similarità, topK, campi indicizzati) e il tempo di elaborazione.

```
@Entity('assisted_selling_conversations')
export class AssistedSellingConversation {
    @Column({ type: 'uuid', unique: true })
    declare conversationId: string;

    @Column({ type: 'jsonb' })
    declare messages: Array<{
        role: 'system' | 'user' | 'assistant'
            | 'result' | 'tool';
        content: string | unknown;
        timestamp?: Date;
    }>;
    ...
    @OneToMany(() => AssistedSellingResult,
               r => r.conversation)
    declare results: AssistedSellingResult[];
}

@Entity('assisted_selling_results')
export class AssistedSellingResult {
    @Column({ type: 'uuid' })
    declare conversationId: string;

    @Column({ type: 'jsonb' })
    declare suggestedModels: Array<{
        id: string; code: string; name: string;
        matchedKFCount: number; totalScore: number;
    }>;

    @Column({ type: 'jsonb', nullable: true })
    declare searchParameters: {
        productDescription: string;
        gammaCode: string | null;
        topK: number;
        scoreThreshold: number;
        fieldsKeyFeatures: string[];
        fieldsProduct: string[];
    };
    ...
    @ManyToOne(() => AssistedSellingConversation,
               c => c.results)
    declare conversation: AssistedSellingConversation;
}
```

Listing 5.13: Le due entità TypeORM per la persistenza delle sessioni.

Questa granularità non è fine a sé stessa. Il motivo dell'introduzione di un sistema di logging di questo tipo è dato dal fatto che a partire da ciascuna raccomandazione prodotta è possibile ricostruire a posteriori l'intero percorso decisionale: quali prodotti sono stati suggeriti, quali caratteristiche erano state estratte dalla conversazione, con quali parametri di ricerca (soglia di similarità, topK, campi indicizzati) e in quanto tempo. Questi dati sono la base per la calibrazione nel tempo: se i modelli suggeriti si rivelano inadeguati, i log consentono di risalire al motivo (la qualità delle caratteristica estratta, la soglia troppo permissiva o troppo restrittiva, un campo indicizzato irrilevante) e apportare delle modifiche per ottimizzare il processo.

5.7 Orchestrazione: l'endpoint principale

Il punto d'ingresso lato server è un router Express che espone quattro endpoint REST principali. L'endpoint principale segue il flusso integrato descritto nella Sezione 4.4.5: il modello linguistico dispone dei quattro strumenti e gestisce autonomamente l'intera sessione, dalla raccolta dei bisogni dell'utente alla raccomandazione finale. Questo endpoint viene interrogato **ad ogni nuovo messaggio dell'utente**: il server recupera lo storico della conversazione dal database, aggiunge il nuovo messaggio e lo invia al modello linguistico insieme al contesto completo. La scelta di ricostruire lo stato lato server ad ogni interazione, anziché mantenerlo nel browser, nasce dall'esigenza di garantire coerenza e recuperabilità della sessione anche in caso di interruzioni o ricariche della pagina. Un altro endpoint implementa un canale Server-Sent Events per notificare il frontend in tempo reale, di cui si parlerà nella Sezione 5.8.4.

Due dei restanti endpoint realizzano il flusso alternativo introdotto nella Sezione 4.4, in cui conversazione e matching sono disaccoppiati: il primo gestisce i singoli turni di dialogo, il secondo estrae le caratteristiche e lancia la strategia di matching duale (Sezione 4.4.4). Come anticipato, questo percorso garantisce compatibilità con provider che non supportano il function calling e durante lo sviluppo ha permesso di isolare e calibrare il matching indipendentemente dalla qualità della conversazione.

Il Listato 5.14 mostra la struttura completa dell'handler principale, in cui le parti non rilevanti vengono omesse. La struttura è lineare: ricostruzione della conversazione (o inizializzazione al primo turno), preparazione dei tool, invocazione del ciclo `runAIConversation`, logging della risposta, rilevamento dei token di stato e risposta al client.

```
1 router.post('/assistedSelling', async (req, res) => {
2     const startTime = Date.now();
3     const userMessage: ChatMessage = req.body.userMessage;
4     const conversationId: string = req.body.conversationId;
```

```
5       const ai = getAIProviderInstance();
6       const gammaDescriptions = await getGammaDescriptions(...);
7       const gammas = gammaDescriptions.map(g => ({
8           name: g.gamma, description: g.gammaDescription,
9       }));
10
11      // Ricostruzione o inizializzazione della conversazione
12      const conversationLogged = await logger.getConversation(
        conversationId);
13      if (!conversationLogged) {
14          conversation = [{ role: 'system', content: systemPrompt(
        gammas) }];
15          await logger.logConversation({ conversationId, messages:
        conversation, ... });
16      } else {
17          conversation = conversationLogged.messages;
18      }
19      conversation.push(userMessage);
20
21      // Definizione dei tool con le rispettive implementation
22      const tools: ToolParam[] = [
23          { name: 'web_search_reference_product',   implementation:
        ... },
24          { name: 'get_key_features_from_characteristics',
        implementation: ... },
25          { name: 'get_models_from_description',    implementation:
        ... },
26          { name: 'get_model_specs_from_model_ids', implementation:
        ... },
27      ];
28
29      // Ciclo AI: tool call o risposta testuale
30      const { content: assistantResponse, totalTokens } =
31          await runAIConversation(conversation, ai, tools);
32
33      conversation.push({ role: 'assistant', content:
        assistantResponse });
34
35      // Logging; %%% indica che l'utente ha accettato il riepilogo
36      const isCompleted = assistantResponse?.includes('%%%') ??
        false;
37      await logger.logConversation({
38          conversationId, messages: conversation,
39          ...(isCompleted && { isCompleted: true }), totalTokens,
40      });
41
42      // Estrazione e logging dei modelli finali
43      if (assistantResponse?.includes('<models>')) {
44          const modelsJson =
```

```
45              assistantResponse.match(/<models>([\s\S]*?)<\/models
    >/)?.[1] ?? '[]';
46          const qaPairs = extractQAPairsFinal(conversation);
47          await logger.logResult({ conversationId,
48              suggestedModels: JSON.parse(modelsJson), ... });
49          return res.json({ type: 'models', assistant: modelsJson,
    qaPairs });
50
51      }
52      res.json({ type: 'assistant-response', assistant:
    assistantResponse });
53  });
```

Listing 5.14: Struttura dell'handler `POST /assistedSelling`.

5.8 Interfaccia utente

Il modulo frontend è costruito con React 18 e Material-UI e si presenta come pannello laterale affiancato alla griglia di ricerca prodotti. L'utente può scorrere il catalogo mentre risponde alle domande dell'assistente, e quando le raccomandazioni arrivano, il confronto con ciò che stava già guardando è immediato. Spostarlo su una pagina separata avrebbe limitato la possibilità di confronto.

5.8.1 Gli stati dell'interfaccia

Il pannello attraversa tre stati principali che rispecchiano le fasi della sessione: la conversazione, l'attesa del calcolo e la presentazione dei risultati. Le transizioni non dipendono da una logica esplicita nel frontend, ma dai token che il modello linguistico restituisce all'interno della risposta, in particolare il token `%%%` che segnala l'accettazione del riepilogo e il tag `<models>` che porta alle raccomandazioni finali. Il Listato 5.15 mostra i due flag booleani che traducono questi segnali in schermate distinte.

```
1  const [calculateModels, setCalculateModels] = useState(false);
2  const [showModels, setShowModels] = useState(false);
3
4  // Nella risposta al token %%%
5  setCalculateModels(true);
6  // ... chiamata per i modelli ...
7  setCalculateModels(false);
8  setShowModels(true);
9
10 // Nel render
11 {showModels ? (
12     <AssistedSellingResults models={models} />
```

```
) : (
    <AssistedSellingChat
        calculateModels={calculateModels} /* flag nel pannello
    chat abilita lo skeleton che simula l'attesa dei suggerimenti
    */
        conversation={conversation}
        loading={loading}
    />
)}
```

Listing 5.15: I due flag che governano le transizioni di stato del pannello.

Prima fase: chat in corso. La conversazione non aspetta che l'utente scriva per primo: all'apertura del pannello il frontend invia automaticamente un messaggio di inizializzazione al server, che risponde con il saluto di benvenuto. Nei test effettuati, la versione con il pannello inizialmente vuoto produceva un momento di esitazione del tipo "cosa scrivo?", il che rallentava l'avvio. Il Listato 5.16 mostra la funzione di inizializzazione: il token `###INIT###` è il segnale convenzionale che il server riconosce per generare il saluto senza aspettarsi contenuto dall'utente. La guardia `hasInitializedRef` evita che il doppio render di React in StrictMode mandi due richieste alla stessa apertura del pannello.

```
const hasInitializedRef = useRef(false);
const [conversationId, setConverId] = useState(() => uuidv4());
const initChat = async () => {
    if (hasInitializedRef.current) return;
    hasInitializedRef.current = true;
    setLoading(true);

    const result = await AIConversationTools(
        { role: 'user', content: '###INIT###' },
        conversationId,
    );

    if (result.success) {
        setFullAssistantMessage(result.assistant ?? '');
    }
};

useEffect(() => {
    initChat();
}, [conversationId]);
```

Listing 5.16: Inizializzazione automatica della chat all'apertura del pannello.

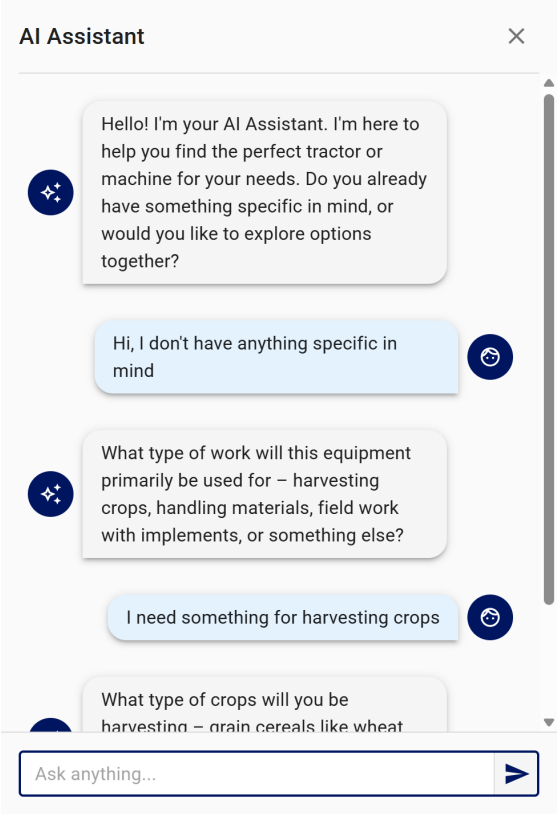

Figura 5.1: Il pannello di chat durante una sessione.

Ogni conversazione porta un UUID generato nel browser al momento dell'apertura del pannello. Il frontend non mantiene una copia locale dello storico: ad ogni messaggio invia al server solo il nuovo input, e il server recupera dal database l'intera cronologia da passare al modello. Il conversationId vive nel React state del componente e non viene salvato, quindi se l'utente chiude il pannello o ricarica la pagina, al successivo montaggio viene generato un nuovo UUID e la sessione riparte da zero. I messaggi rimangono nel database per finalità di log e analisi, ma non vengono mai riproposti all'utente in sessioni successive.

Seconda fase: l'attesa del calcolo. Quando il token `%%%` compare nella risposta del modello, il frontend lo riconosce come segnale che l'utente ha accettato il riepilogo e invia in background il token `###` che segnala l'attesa da parte del frontend dei modelli raccomandati dall'assistente. Il flag `calculateModels` diventa `true`, textbox di input viene disabilitata e il componente `AssistedSellingChat` renderizza uno `Skeleton`: l'utente viene notificato del fatto che qualcosa sta per accadere, e nello specifico che è in corso la generazione dei modelli.

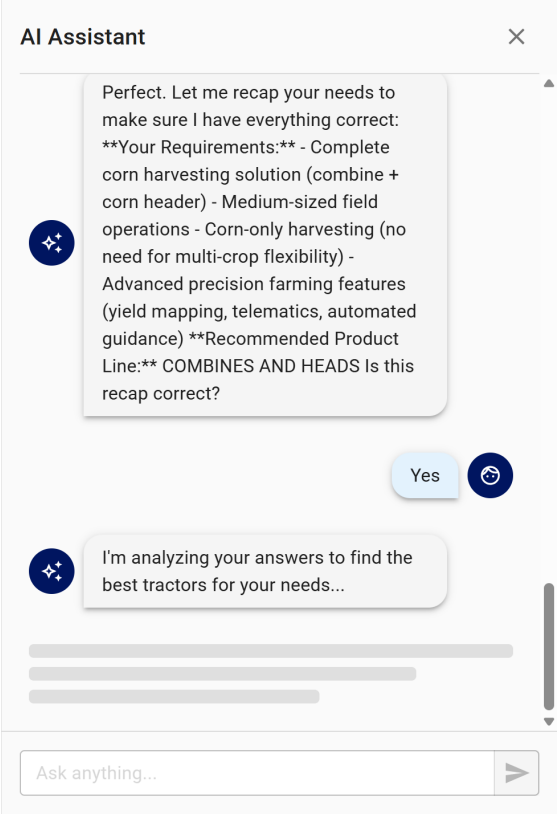

Figura 5.2: La schermata di attesa con skeleton animato.

```
if (
    result.assistant?.toLowerCase().includes('%%%')
) {
    setConversation([
        ...updatedConversation,
        {
            role: 'assistant',
            content: "I'm analyzing your answers...",
        },
    ]);
    setCalculateModels(true);
    setLoading(false);

    const modelsResult = await AIConversationTools(
        { role: 'user', content: '###' },
        conversationId,
    );
```

Listing 5.17: Rilevamento del token %%% e avvio del flusso di raccomandazione.

Terza fase: i modelli suggeriti. Alla ricezione della risposta del server con i modelli suggeriti, il flag `showModels` diventa `true` e il componente `AssistedSellingResults` sostituisce la chat. La schermata, la struttura delle schede modello e l'integrazione con il CPQ saranno descritte nella Sezione 5.8.2. Un pulsante "New Search" può essere cliccato e riporta allo stato iniziale generando un nuovo UUID di conversazione.

```
1  if (modelsResult.type === 'models') {
2      const modelsResultData = JSON.parse(modelsResult.assistant ||
       '[]');
3      setModels(
4          modelsResultData.map((model: any) =>
5                  buildModelWithProductUrl(model)
6          ),
7      );
8      setQaSummary(modelsResult.qaPairs || []);
9      setConversation([]);
10 }
11 setCalculateModels(false);
12 setShowModels(true);
```

Listing 5.18: Ricezione e parsing della risposta con i modelli suggeriti.

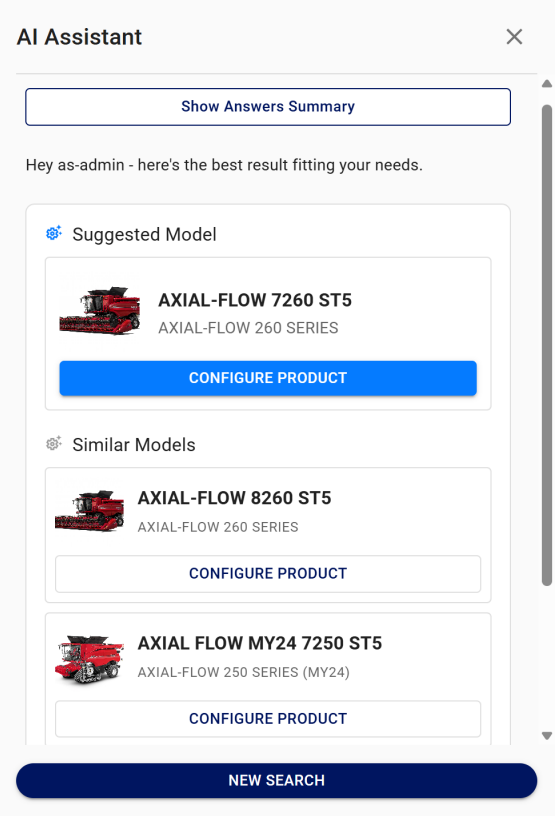

Figura 5.3: La schermata dei risultati.

Stato di assenza risultati. Esiste un quarto stato, secondario rispetto ai tre precedenti, implementato per rispondere al requisito **Gestione eccezioni** della Tabella 3.2 e gestito all'interno di `AssistedSellingResults` stesso: quando la lista dei modelli restituita dal server è vuota, il componente mostra un pannello di fallback con un'icona di avviso, il messaggio "Oops! No results found." e un invito a raffinare la ricerca. La logica è un semplice controllo sulla prop `models`: se `models.length === 0` l'intera sezione dei risultati viene sostituita dal fallback, mantenendo comunque visibile il riepilogo Q&A espandibile in cima. Il pulsante "New Search" rimane accessibile per avviare una nuova sessione.

```
1 const hasModelList = models.length > 0;
2 {hasModelList ? (
3     <>
4         <Typography variant="body2" mb={3}>
5             Hey {displayUserName} - here's the best result...
6         </Typography>
7         <ModelList models={models} />
8     </>
9 ) : (
```

```
10      <Box display="flex" flexDirection="column"
11           alignItems="center" textAlign="center">
12          <SearchOffIcon sx={{ fontSize: 64, mb: 2 }} />
13          <Typography variant="h6">
14              Oops! No results found.
15          </Typography>
16          <Typography>
17              Sorry, I couldn't find any models matching
18              your search criteria. Try refining your
19              search and start a new one.
20          </Typography>
21      </Box>
22  )}
```

Listing 5.19: Gestione dello stato di assenza risultati.

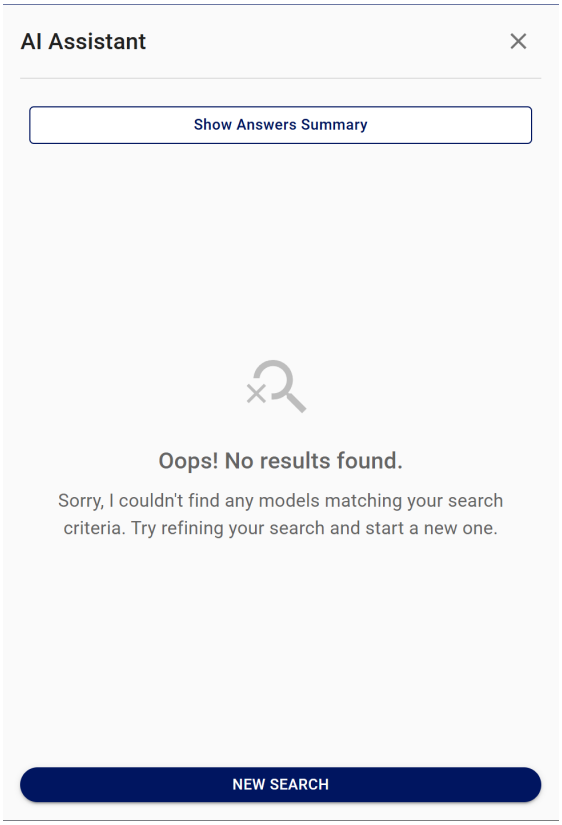

Figura 5.4: La schermata di assenza risultati.

5.8.2 Presentazione dei risultati e integrazione con il CPQ

La schermata dei risultati ospita due elementi. Il primo è un riepilogo espandibile della conversazione, identificato nell'interfaccia come `Show Answer Summary`, che presenta le coppie domanda–risposta in forma sintetica, così da consentire all'utente di rivedere lo storico del dialogo prima di procedere alla configurazione. L'estrazione avviene lato server tramite la funzione `extractQAPairsFinal`: ogni risposta dell'assistente viene accoppiata con la successiva risposta dell'utente, scartando il saluto iniziale e il riepilogo finale, e la lista restituita al client. Il secondo elemento è la lista dei modelli raccomandati, ordinati per affinità rispetto alle esigenze emerse. Il primo della lista viene evidenziato come "Suggested Model" con un'icona dedicata; i successivi sono raggruppati sotto "Similar Models", con la possibilità di espandere la lista completa. Ogni scheda mostra l'immagine del prodotto, il nome e il range di appartenenza. Il pulsante "Configure Product" realizza l'integrazione con il CPQ progettata nella Sezione 4.5. A partire da tre attributi del modello (gamma, range e nome) il frontend compone un URL diretto verso il portale del configuratore Case IH. Il Listato 5.20 mostra la funzione `buildModelWithProductUrl`: i tre valori vengono codificati con `encodeURIComponent` per gestire nomi con spazi e caratteri speciali, quindi composti nel path dell'URL del CPQ insieme a due parametri fissi che selezionano il mercato e la divisione di prodotto.

```
export function buildModelWithProductUrl(model: ProductMatch) {
    const urlencodedModelCode  = encodeURIComponent(model.name);
    const urlencodedGammaCode  = encodeURIComponent(model.
   gammaName);
    const urlencodedRangeName  = encodeURIComponent(model.
   rangeName);
    return {
        ...model,
        productUrl:
            URL_CONFIGURATORE_CNH
            + `${urlencodedGammaCode}/`
            + `${urlencodedRangeName}/`
            + `${urlencodedModelCode}`
            + `?vhsar=uk%20case&division=AG`,
    };
}
```

Listing 5.20: Costruzione dell'URL del CPQ.

Il click apre l'URL in una nuova scheda del browser, senza chiamate API intermedie né logica aggiuntiva. Le regole di compatibilità, il calcolo del prezzo e la generazione del preventivo restano interamente a carico del CPQ, che non viene toccato in alcun modo: il sistema di Assisted Selling si limita a indirizzare l'utente verso il punto d'ingresso corretto (Figura 5.3). È la traduzione più letterale

possibile del principio di **integrazione non invasiva** enunciato fra gli obiettivi del Capitolo 3: un singolo link è l'unico punto di contatto fra i due sistemi.

5.8.3 Effetto typewriter e percezione della latenza

Le risposte dell'assistente non compaiono tutte insieme. Un timer aggiunge un carattere alla volta ogni 50 millisecondi, simulando una digitazione in tempo reale. L'accorgimento serve due scopi: dà all'utente il tempo di leggere la risposta via via che appare, evitando il cosiddetto "muro di testo", e genera una percezione di naturalezza nell'interazione che un rendering istantaneo non avrebbe. Il Listato 5.21 mostra il `useEffect` responsabile dell'effetto: il messaggio completo è già disponibile in `fullAssistantMessage`, ma viene rivelato un carattere alla volta finché `index` non raggiunge la lunghezza totale; solo a quel punto l'intervallo viene liberato e il messaggio viene inserito nella cronologia della conversazione.

```
1  useEffect(() => {
2      if (!loading || !fullAssistantMessage) return;
3
4      let index = 0;
5      typingIntervalRef.current = setInterval(() => {
6          index += 1;
7          setTypingText(fullAssistantMessage.slice(0, index));
8          if (index === fullAssistantMessage.length) {
9              clearInterval(typingIntervalRef.current);
10             setLoading(false);
11             setConversation(prev => [
12                 ...prev,
13                 { role: 'assistant', content: fullAssistantMessage
       },
14             ]);
15             setFullAssistantMessage('');
16         }
17     }, 50);
18
19     return () => clearInterval(typingIntervalRef.current);
20 }, [fullAssistantMessage, loading]);
```

Listing 5.21: Effetto typewriter.

5.8.4 Aggiornamenti in tempo reale via SSE

La ricerca web è un'operazione molto dispendiosa in termini di tempo e avviene *all'interno* della stessa chiamata HTTP all'endpoint `/assistedSelling`: il tool di ricerca viene invocato dal modello LLM come parte del normale ciclo conversazionale e il server non restituisce la risposta finale finché tutta la catena non

è completata. Questo vincolo architetturale esclude soluzioni alternative come un endpoint separato su cui fare polling: non esiste un identificativo di *job* da interrogare, né un risultato intermedio che il client possa richiedere autonomamente. L'utente, senza feedback visivo, potrebbe pensare che qualcosa si sia bloccato, e l'unico modo per inviare aggiornamenti al frontend *durante* una chiamata ancora aperta è tenere viva una seconda connessione dedicata al solo flusso di eventi.

Per questo motivo è stato introdotto un canale Server-Sent Events parallelo. Il meccanismo si sviluppa su due livelli. Lato server, il tool di ricerca pubblica eventi su un bus `EventEmitter` interno di Node.js nel corso dell'operazione; un endpoint dedicato si abbona allo stesso bus e trascrive ogni evento sulla response HTTP con la formattazione SSE standard. Il Listato 5.22 mostra entrambe le parti: le chiamate a `publishAssistedSellingSearchEvent` nel tool e la logica di sottoscrizione e forwarding nell'endpoint.

```
1  // Nel tool di ricerca web
2  publishAssistedSellingSearchEvent({
3      conversationId, event: 'search_started',
4      message: `Web search started for "${query}"`,
5      operationId, timestamp: Date.now(),
6  });
7
8  const researchResult = await ai.runDeepResearch(query);
9
10 publishAssistedSellingSearchEvent({
11     conversationId, event: 'search_completed',
12     message: `Web search completed for "${query}"`,
13     operationId, timestamp: Date.now(),
14 });
15
16 // Nell'endpoint GET /assistedSellingStream
17 res.setHeader('Content-Type', 'text/event-stream');
18 res.setHeader('Cache-Control', 'no-cache');
19 res.setHeader('Connection', 'keep-alive');
20 res.flushHeaders();
21
22 const sendEvent = (event: AssistedSellingSearchEvent) => {
23     res.write(`event: ${event.event}\n`);
24     res.write(`data: ${JSON.stringify(event)}\n\n`);
25 };
26 const unsubscribe = subscribeAssistedSellingSearchEvents(
       conversationId, sendEvent);
27 const heartbeat = setInterval(() => {
28     res.write('event: ping\ndata: {}\n\n');
29 }, 25000);
30
31 req.on('close', () => {
32     clearInterval(heartbeat);
```

```
33      unsubscribe();
34      res.end();
35  });
```

Listing 5.22: Lato server: il tool pubblica eventi sul bus interno di Node.js.

Lato client, un `EventSource` apre la connessione SSE passando il `conversationId` come parametro e registra un handler per ciascuno dei quattro eventi. Ogni handler aggiorna in-place l'ultima voce di tipo `web-search` nella lista dei messaggi, senza aggiungere nuove voci: l'utente vede un'icona animata durante la ricerca (aggiornata a ogni evento di progresso), un segno di spunta verde al completamento, un'icona di errore in caso di fallimento. Il Listato 5.23 mostra la registrazione dei listener e la logica di aggiornamento.

```
1   const eventSource = openAssistedSellingStream(conversationId);
2
3   const handleStarted = (event) => {
4       const webSearchEvent = {
5           id: `ws-${Date.now()}`,
6           status: 'started', message: payload.message,
7       };
8       setConversation(prev => [
9           ...prev,
10          { role: 'web-search', content: '', event: webSearchEvent
        },
11      ]);
12  };
13
14  const handleProgress = (event) => {
15      setConversation(prev => {
16          const last = prev[prev.length - 1];
17          if (last?.role === 'web-search') {
18              return [...prev.slice(0, -1),
19                  { ...last, event: { ...last.event,
20                      status: 'progress', message: payload.message }
        }];
21          }
22          return prev;
23      });
24  };
25  // handleCompleted e handleErrored seguono lo stesso pattern
26  eventSource.addEventListener('search_started',   handleStarted);
27  eventSource.addEventListener('search_progress',  handleProgress);
28  eventSource.addEventListener('search_completed', handleCompleted);
29  eventSource.addEventListener('search_error',     handleErrored);
```

Listing 5.23: Lato client: `EventSource` ascolta il canale SSE.

La Figura 5.5 mostra l'indicatore visivo nella fase di ricerca in corso (spinner con messaggio di progresso).

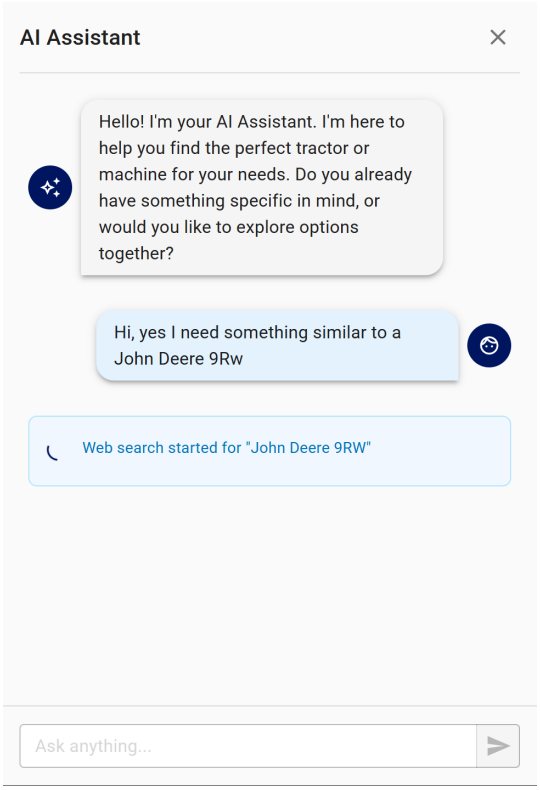

Figura 5.5: L'indicatore di ricerca web nel pannello chat.

Questa scelta implementativa è una risposta diretta al requisito di **prestazioni** (Tabella 3.3): quando il tempo di elaborazione non può essere ridotto ulteriormente, informare l'utente sullo stato dell'operazione trasforma un'attesa passiva in un'attesa consapevole, percepita come più breve.

Capitolo 6

Validazione delle soluzioni implementate

Questo capitolo valuta in che misura il modulo realizzato raggiunge gli obiettivi fissati e risponde alle criticità identificate nel Capitolo 2. Non si tratta di uno studio controllato con gli utenti finali, in quanto il contesto aziendale non lo rendeva praticabile. La valutazione si basa su **35 sessioni di test** condotte sul catalogo reale Case IH durante lo sviluppo, i cui dati sono stati raccolti dal sistema di logging descritto nella Sezione 5.6. Ogni sessione è registrata con il provider attivo, i token consumati, le caratteristiche estratte, i parametri di ricerca, i modelli suggeriti e i tempi di elaborazione.

Le sessioni sono state distribuite sui tre profili descritti nella Sezione 3.2 e sui tre provider disponibili, come riassunto nella Tabella 6.1.

Profilo utente	**OpenAI**	**Bedrock**	**Ollama**	**Totale**
Con prodotto di riferimento	5	4	–	9
Con esigenze operative chiare	5	4	5	14
Con bisogni vaghi	4	4	4	12
Totale	14	12	9	**35**

Tabella 6.1: Distribuzione delle sessioni di test per profilo utente e provider AI.

Ciascuno scenario è stato percorso più volte variando il provider attivo. Ollama non compare negli scenari con prodotto di riferimento perché un modello che gira in locale e quindi dipendente dalla macchina prevede dei tempi d'esecuzione elevati, e l'aggiunta della possibilità di fare web search avrebbe reso l'operazione troppo dispendiosa. Avendo previsto Ollama come provider di fallback e utilizzato

prevalentemente in fase di sviluppo iniziale, nella soluzione adottata non supporta la ricerca web (cfr. Sezione 5.2).

6.1 Flessibilità e indipendenza dal provider

Passare da un provider all'altro ha richiesto la sola modifica delle le variabili d'ambiente. Tutte e tre le implementazioni (OpenAI, AWS Bedrock, Ollama) sono state testate sull'intero flusso, dal saluto iniziale ai suggerimenti finali, senza toccare una riga di codice applicativo.

I due provider cloud generano embedding a 1536 dimensioni, Ollama a 768. In un'architettura con indici condivisi questa differenza sarebbe un problema; la separazione degli indici per provider descritta nella Sezione 5.2, è diventata invece oggetto di validazione della bontà della soluzione: lo stesso nodo Neo4j ospita più set di embedding coesistenti senza conflitti, e la stessa query inviata a provider diversi sui rispettivi indici ha permesso di confrontare i risultati senza dover rigenerare i dati.

Le conversazioni prodotte da OpenAI e Bedrock sono più mirate: domande più precise, identificazioni di famiglie più sicure. Ollama gira in locale su LLaMA 3.1, un modello molto più piccolo di GPT-4o o Claude Sonnet, e la differenza è evidente (domande più generiche, qualche incertezza in più sull'identificazione della famiglia). Non è un difetto dell'architettura: l'interfaccia è uniforme mentre la qualità del ragionamento dipende invece dal modello scelto. Ollama rimane il provider ideale per lo sviluppo locale in assenza di connessione o credenziali API, ma è stato progettato per operare come provider di fallback o in casi di "emergenza".

Un caso a parte ha riguardato la ricerca web. Come descritto nella Sezione 5.2, OpenAI e Bedrock la implementano in modo completamente diverso (tool nativo nel primo caso, Bedrock Agents nel secondo) ma il profilo d'uso che producono è nello stesso formato e viene iniettato nel contesto allo stesso modo, e ciò lo rende indistinguibile fra i due provider. Ollama non prevede ricerca web e il modello va avanti senza il profilo, il che è accettabile in fase di sviluppo ma esclude Ollama dagli scenari di deployment in produzione.

6.2 Prestazioni e latenza percepita

La valutazione delle prestazioni si basa sui tempi di risposta registrati dal sistema di logging per ciascun turno conversazionale. I tempi dipendono in modo significativo da quale percorso segue l'handler principale ad ogni turno, e dall'analisi dei log emergono quattro situazioni distinte. La Tabella 6.2 riporta i valori misurati sulle 35 sessioni, espressi come media e intervallo osservato per ciascuna tipologia di turno.

Tipo di turno	**Provider cloud (media)**	**Ollama locale (media)**
Conversazionale (nessun tool)	2.4 s [1.1–4.2 s]	14.8 s [8–22 s]
Con tool di matching vettoriale	5.6 s [3.2–8.4 s]	28.3 s [18–41 s]
Con ricerca web (OpenAI)	14.3 s [8–21 s]	n.d.
Con ricerca web (Bedrock Agents)	22.7 s [14–31 s]	n.d.

Tabella 6.2: Latenze misurate per tipo di turno conversazionale (media e intervallo osservato su 35 sessioni).

Nel caso base di conversazione ordinaria, in cui il modello elabora l'input dell'utente e formula una domanda o presenta il riepilogo senza invocare strumenti, il tempo impiegato è quasi tutto dovuto a latenza di rete e inferenza da parte del modello. Con i provider cloud (OpenAI e Bedrock) la media si attesta intorno a 2–3 secondi, del tutto accettabile per un'interazione domanda-risposta conversazionale.

Quando il modello invoca i tool di matching vettoriale, per key feature e per profilo d'uso, si aggiungono le operazioni di ricerca vettoriale in Neo4j. Una singola query HNSW su un indice di qualche centinaio di nodi non supera i 20 millisecondi: di fatto sposta poco o nulla rispetto ai tempi dell'LLM. Il tempo totale di un turno con invocazione di strumenti sale a circa 5–6 secondi in media, un valore che rimane nella soglia di attesa accettabile per un'operazione di raccomandazione.

L'invocazione della ricerca web è il caso più impattante. Il tool nativo di OpenAI richiede in media 14 secondi mentre con Bedrock Agents si arriva a circa 23 secondi, perché l'agente esterno ha una latenza di avvio non eliminabile. Nelle prime iterazioni, senza SSE, dopo 10–15 secondi di silenzio l'utente si chiedeva cosa stesse succedendo, il che generava un po' di confusione; il canale SSE descritto nella Sezione 5.8.4 ha risolto il problema percettivo: l'utente sa che il sistema sta lavorando e un indicatore mostra la ricerca in corso.

Una sessione completa va dal saluto ai modelli in media in 6 turni conversazionali. Senza ricerca web l'interazione attiva dura circa 45 secondi, con ricerca web si aggiunge la tempistica dell'operazione in questione mentre il resto della sessione non è influenzato.

Ollama in locale è significativamente più lento (15 secondi in media anche per un turno semplice) perché il modello gira sulla CPU della macchina di sviluppo. Il gap con i provider cloud è elevato ma, per lo scopo per cui è stato pensato, ovvero sviluppare offline senza costi di API, non rappresenta un problema. Lo diventerebbe se venisse usato in produzione, ma non è il caso.

6.3 Qualità delle raccomandazioni

Misurare la precisione dei suggerimenti in modo rigoroso richiederebbe una *ground truth*, ovvero un insieme di corrispondenze esigenze → modelli validate da un esperto di prodotto. Non avendo a disposizione uno strumento del genere, la valutazione è rimasta qualitativa e osservativa: per ogni sessione si è verificato se i modelli suggeriti risultassero sensati a chi conosce il catalogo, e se la conversazione avesse seguito un percorso coerente con le esigenze espresse. I risultati delle verifiche sono riassunti nella Tabella 6.3.

Esito	**OpenAI**	**Bedrock**	**Ollama**	**Totale**
Gamma corretta, modelli coerenti	11	9	5	25
Gamma corretta, modelli parz. coerenti	2	1	1	4
Gamma errata	1	1	2	4
Nessun risultato restituito	0	1	1	2
Totale	14	12	9	**35**

Tabella 6.3: Esito qualitativo delle sessioni di test per provider.

Su 35 sessioni complessive, in 25 casi (71%) la gamma identificata è risultata corretta e i modelli suggeriti coerenti con le esigenze espresse. In 4 casi (11%) la gamma era corretta ma i modelli solo parzialmente pertinenti. I casi di gamma errata si sono verificati 4 volte (11%), quasi tutti concentrati nelle sessioni senza descrizioni di gamma (cfr. Sezione 6.3.2).

6.3.1 Calibrazione della soglia di similarità

Il parametro più delicato da calibrare è stata la soglia di *cosine similarity* per la ricerca vettoriale, discussa nella Sezione 5.5 e fissata a 0.7. Il valore è stato determinato attraverso un processo iterativo, confrontando i risultati della stessa query con soglie diverse. La Tabella 6.4 mostra un esempio rappresentativo.

Soglia	**Key feature restituite per la query *"works on steep slopes"***	**Score**
0.5	Rotary Dust Screen Brush	0.54
0.7	POWERSHUTTLE	0.71
0.8	(nessun risultato)	–

Tabella 6.4: Effetto della soglia di cosine similarity sui risultati della ricerca vettoriale per una query rappresentativa.

Con una soglia più bassa (0.5–0.6), la ricerca diventava troppo permissiva e semanticamente distante dall'esigenza espressa. Un caso osservato è stato abbastanza emblematico: la query "works on steep slopes" restituiva una key feature legata ad una spazzola rotante che pulisce automaticamente il dust screen quando ci sono condizioni con molta polvere, pertanto non rispondeva all'esigenza specifica.

Con una soglia più alta (0.8–0.9), si verificava il problema opposto, ovvero key feature legittime espresse con terminologia differente venivano escluse. Nello specifico, la stessa query non trovava match con key feature esistenti che avevano lo stesso significato operativo.

La soglia di 0.7 ha prodotto ad esempio, per la stessa query, la feature di una leva capace di rendere più facili gli spostamenti e permettere manovre prevedibili e sicure, ed in generale è una soglia che garantisce il miglior equilibrio tra precisione e copertura, proponendo match pertinenti senza escludere sinonimi e varianti ragionevoli.

6.3.2 Identificazione della gamma

L'identificazione corretta della gamma è il prerequisito per la qualità delle raccomandazioni, dato che una gamma sbagliata invalida l'intero processo di matching. Sui dati delle sessioni di test, il tasso di identificazione corretta è riassunto nella Tabella 6.5.

Condizione	**Gamma corretta**	**Gamma errata**
Senza descrizioni di gamma	5 / 8 (63%)	3 / 8 (37%)
Con descrizioni di gamma	26 / 27 (96%)	1 / 27 (4%)

Tabella 6.5: Tasso di identificazione corretta della gamma con e senza descrizioni generate.

Nelle prime iterazioni e test svolti, le descrizioni di gamma non erano ancora state generate e il risultato era sistematicamente insufficiente. Senza descrizioni

il modello si aggrappava ai nomi comuni internazionali (ad esempio *tractor* o *harvester*) invece degli scenari d'uso, e sui casi ambigui andava fuori strada. Il caso più ricorrente: un utente che descriveva l'esigenza di *"raccogliere e legare balle di fieno"* finiva invariabilmente su Combines invece che su Balers, perché il modello associava "raccogliere" a mietitrebbia senza nessuna mediazione sul tipo di operazione.

Con le descrizioni generate dallo script (scenari tipici, ambienti operativi, caratteristiche differenzianti per ciascuna famiglia), lo stesso scenario veniva risolto correttamente: su 27 sessioni con descrizioni attive, la gamma è stata identificata correttamente in 26 casi (96%). Le descrizioni non sono quindi un'aggiunta opzionale ma una precondizione per il corretto funzionamento della fase di identificazione.

6.3.3 Completezza dei suggerimenti

Durante le sessioni di test è emerso che alcuni modelli nel grafo avevano dati incompleti: `productImage` assente, `rangeName` non valorizzato. Per quantificare l'effetto del vincolo di completezza nel prompt (Listato 5.9), si è confrontato il comportamento del sistema con e senza di esso su un campione di sessioni.

Condizione	**Modelli corretti**	**Modelli con campi inventati**
Senza vincolo di completezza	18	6
Con vincolo di completezza	42	0

Tabella 6.6: Effetto del vincolo di completezza nel prompt sulla qualità dei dati nei suggerimenti.

Senza il vincolo, il modello linguistico inventava i campi mancanti: nomi di range inesistenti, URL di immagini fabbricati. Con il vincolo attivo, i modelli con dati incompleti venivano scartati e a nessun campo inventato è stato permesso di raggiungere l'interfaccia. È la traduzione pratica del principio discusso nella Sezione 4.4.2.

6.4 Effetto del riepilogo

Nelle prime versioni del prompt, dopo la raccolta delle esigenze il modello passava direttamente ai suggerimenti. I risultati arrivavano, spesso pertinenti, ma l'utente non aveva modo di capire su quali premesse si basassero. Quando qualcosa andava storto, ad esempio gamma sbagliata, modelli fuori tema, non c'era modo di risalire al problema: era stata una risposta ambigua al terzo turno? Un'interpretazione errata nella fase di matching? Una scelta di famiglia sbagliata in partenza?

L'introduzione della fase di riepilogo ha cambiato questa dinamica. Per quantificarne l'effetto, si sono confrontate le sessioni condotte prima e dopo l'aggiunta del recap nel system prompt. I risultati sono riassunti nella Tabella 6.7.

Metrica	**Senza recap**	**Con recap**
Sessioni totali	12	23
Gamma errata nei suggerimenti finali	4 (33%)	0 (0%)
Fraintendimenti corretti dall'utente	–	3

Tabella 6.7: Effetto dell'introduzione del riepilogo sulla qualità delle raccomandazioni.

Senza recap, 4 sessioni su 12 (33%) hanno prodotto suggerimenti basati su premesse errate (gamma sbagliata o esigenze fraintese) senza che il problema fosse rilevabile prima dell'output finale. Con il recap attivo, i fraintendimenti sono diventati visibili e correggibili: in 3 casi l'utente ha potuto correggere un'interpretazione errata prima che il modello procedesse al matching, e i casi di raccomandazione basata su gamma sbagliata sono scesi a zero. Il recap funziona quindi come punto di controllo: non rende il modello più preciso nella raccolta delle informazioni, ma intercetta gli errori prima che si propaghino fino ai suggerimenti.

6.5 Valutazione dei requisiti

Le Tabelle 6.8 e 6.9 chiudono il capitolo incrociando ciascun requisito con lo stato risultante dalle sessioni di test e l'evidenza che lo supporta.

Requisito	**Stato**	**Evidenza**
Gestione interazione	Soddisfatto	Contesto mantenuto su tutti i turni in 35/35 sessioni
Richiesta chiarimenti	Soddisfatto	Domande di chiarimento generate in 8 sessioni con input ambigui
Estrazione informazioni	Soddisfatto	Soglia 0.7 calibrata su dati reali (Tab. 6.4)
Suggerimento modelli	Soddisfatto	83% sessioni con modelli coerenti (Tab. 6.3)
Generazione recap	Soddisfatto	Recap generato e confermato in 23/23 sessioni
Gestione eccezioni	Soddisfatto	Fallback attivato correttamente in 2 sessioni senza risultati
Integrazione CPQ	Soddisfatto	URL generati validi e funzionanti per tutti i modelli suggeriti
Ricerca prodotto di riferimento	Soddisfatto	Profilo recuperato in 9/9 sessioni con prodotto di riferimento

Tabella 6.8: Valutazione dei requisiti funzionali (cfr. Tabella 3.2).

Requisito	**Stato**	**Evidenza**
Accuratezza	Soddisfatto	Suggerimento modelli coerenti (Tab. 6.3) ed eliminazione della possibilità di generare campi inventati (Tab. 6.6)
Robustezza	Soddisfatto	Dialogo mantenuto in 8 sessioni con input incompleti o ambigui
Prestazioni	Parz. soddisfatto	Tra le altre, media turni conversazionali 2.4 s cloud e 14–23 s in media per web search (Tab. 6.2)
Modularità	Soddisfatto	Architettura indipendente dal catalogo, logica CPQ integrata correttamente

Tabella 6.9: Valutazione dei requisiti non funzionali (cfr. Tabella 3.3).

Il requisito di prestazioni è l'unico con uno stato parziale, e riguarda specificamente la ricerca web: una media di 14–23 secondi è oltre la soglia di attesa accettabile senza feedback visivo. Il canale SSE trasforma quell'attesa da silenzio a informazione risolvendo il problema percettivo, ma il tempo effettivo rimane. In

ambienti di produzione, la qualità dell'esperienza utente dipende dalla disponibilità del canale SSE e dalla qualità della connessione.

La robustezza è risultata soddisfatta nelle sessioni con input incompleti o ambigui: in tutti gli 8 casi il modello ha mantenuto il dialogo, formulando domande di chiarimento invece di interrompere il flusso o produrre suggerimenti non pertinenti.

Capitolo 7

Conclusioni

Il lavoro descritto in questa tesi ha portato alla realizzazione di un modulo di vendita assistita integrato nella piattaforma CoolPIM, progettato e sviluppato nel contesto del catalogo Case IH per il mercato UK. L'obiettivo era rispondere a una criticità concreta individuata nel Capitolo 2: il configuratore presuppone che l'utente sappia già cosa cercare prima ancora di iniziare. Il modulo inverte questa logica: è il sistema a condurre l'utente dalla descrizione delle proprie esigenze operative alle specifiche dei modelli, attraverso una conversazione guidata.

7.1 Sintesi del lavoro svolto

Il presente lavoro ha affrontato una criticità strutturale dei configuratori di prodotto, e in particolare quello di Case IH, brand di CHN Industrial: il processo di selezione preliminare del modello presuppone che l'utente conosca già la struttura del catalogo e le classificazioni all'interno, un'assunzione raramente valida nel contesto della meccanizzazione agricola. Il Capitolo 3 ha formalizzato questo disallineamento attraverso l'analisi di tre profili utente e la derivazione di un insieme di requisiti funzionali e non funzionali; il problema di fondo era già riconoscibile nell'osservazione iniziale del configuratore Case IH: la struttura tecnica dell'offerta e il modello mentale con cui l'utente ragiona non condividono lo stesso linguaggio, e il configuratore non offre alcun meccanismo per colmare quella distanza.

La soluzione descritta nel Capitolo 4 non è stata immediatamente ovvia. L'idea di usare un LLM come orchestratore del dialogo era naturale, ma utilizzarlo anche per generare i suggerimenti avrebbe introdotto il rischio di allucinazione: un modello che non conosce il catalogo Case IH può inventare prodotti che sembrano plausibili ma non esistono. Il paradigma GraphRAG ha risolto il problema separando i due compiti: il modello linguistico conduce la conversazione, raccoglie le esigenze, decide quando e come interrogare il catalogo; i suggerimenti vengono però estratti dal grafo

Neo4j attraverso quattro strumenti specializzati, non dall'inferenza del modello. Neo4j è stata la scelta naturale per rappresentare il modello dati perché la struttura gerarchica del catalogo si presta più a un grafo che ad una tabella: le relazioni tra gamma, range, modello e key feature sono esplicite e percorribili, mentre in una struttura piatta andrebbero ricostruite ogni volta. Inoltre, per risolvere il problema di matching semantico fra il linguaggio dell'utente e la terminologia tecnica del catalogo si è adottato il meccanismo degli embedding vettoriali, utile per rappresentare un informazione testuale in una struttura confrontabile, e alla ricerca vettoriale tramite indici HNSW, strumento principale per effettuare delle misurazioni tra vettori riducendo il tempo d'esecuzione. La scelta progettuale per quanto riguarda il flusso conversazionale è stata dividerlo in tre fasi quali identificazione della famiglia, riepilogo e conferma, suggerimento dei modelli, una scelta non solo per facilitare l'esperienza utente ma soprattutto perchè restringere prima la famiglia e poi cercare i modelli all'interno di quella famiglia ha prodotto risultati molto più precisi che cercare in tutto il catalogo. Il riepilogo, in particolare, è risultato più importante di quanto si prevedesse: senza di esso i fraintendimenti nelle prime fasi della conversazione si propagavano fino ai suggerimenti finali, spesso basati su premesse sbagliate.

La fase di implementazione descritta nel Capitolo 5 ha affrontato la coesistenza di tre provider con embedding di dimensionalità diversa (1536 per OpenAI e Bedrock, 768 per Ollama), il che ha richiesto un meccanismo di separazione degli indici per poter confrontare i risultati senza dover sovrascrivere gli altri. Il system prompt ha richiesto diverse iterazioni: ogni versione ha corretto un comportamento osservabile specifico, e non era sempre prevedibile quale modifica avrebbe risolto il problema senza crearne altri. La latenza della ricerca web non aveva una soluzione architetturale ovvia: il canale SSE ha risolto il problema percettivo dell'utente senza però ridurre il tempo effettivo. La validazione del Capitolo 6 ha confermato la buona riuscita delle soluzioni introdotte: i tre provider si intercambiano attraverso un cambio di variabili d'ambiente e la soglia di similarità a 0.7 è stato il giusto compromesso per restituire matching sensati. Ha anche messo in luce una dipendenza che non era evidente in fase di progettazione: le descrizioni di gamma generate dallo script di pre-elaborazione non sono solo un'ottimizzazione opzionale, ma una precondizione per il funzionamento della fase di identificazione della gamma. Senza di esse il modello si aggrappava ai nomi comuni delle famiglie e ciò portava spesso a non identificare la gamma corretta.

7.2 Limiti e criticità aperte

La valutazione condotta nel Capitolo 6 è osservativa. L'assenza di una *ground truth*, un insieme di coppie (esigenze, modelli corretti) validate da esperti di prodotto,

rende impossibile misurare la precisione dei suggerimenti in modo rigoroso. Si può affermare che i suggerimenti prodotti nelle sessioni di test apparivano sensati a chi conosce il catalogo; non si può affermare con quale frequenza il sistema selezioni il modello ottimale rispetto a quello semplicemente accettabile. Sul piano delle prestazioni, la criticità principale rimane la latenza della ricerca web descritta nella Sezione 6.2. Le tempistiche dell'operazione sono gestibili grazie al canale SSE, ma la dipendenza da questo canale introduce una fragilità: in assenza di connessione SSE stabile, quell'attesa torna a essere un blocco silenzioso. Non è un problema architetturale, ma un punto da monitorare in produzione.

7.3 Sviluppi futuri

Lo sviluppo più naturale è l'estensione ad altri cataloghi gestiti da CoolPIM. Il modulo è stato progettato senza dipendenze dal dominio specifico di Case IH, infatti il grafo, la struttura delle key feature e le famiglie di prodotto sono parametri di configurazione, non assunzioni architetturali. Estendere il modulo al catalogo di un altro brand richiederebbe la generazione delle descrizioni di gamma per quel catalogo e la reindicizzazione degli embedding, escludendo modifiche al codice applicativo.

Un secondo sviluppo riguarda la qualità della valutazione. Costruire una *ground truth* anche parziale, ovvero un set di scenari con risposta attesa validata da un esperto di prodotto, consentirebbe di misurare in modo quantitativo l'impatto delle scelte architetturali, tra cui il contributo della soglia di similarità, l'effetto delle descrizioni di gamma, il comportamento comparativo dei provider su casi noti. Sarebbe anche la base per rilevare regressioni al cambio di modello o di prompt.

Infine, i dati di log raccolti dal sistema di persistenza descritto nella Sezione 5.6 sono stati utilizzati per la fase di validazione illustrata nel Capitolo 6. Tuttavia, non sono ancora sfruttati in modo automatico per un feedback loop: in futuro, le sessioni di produzione potrebbero diventare la base per affinare dinamicamente i parametri, identificare famiglie di query problematiche o addestrare modello linguistico ed embedding specializzati.

Bibliografia

[1] Case IH. *About Case IH*. Consultato nel 2026. 2026. URL: https://www.caseih.com/en-gb/unitedkingdom/case-ih-world (cit. a p. 7).

[2] Case IH. *History*. Consultato nel 2026. 2026. URL: https://www.caseih.com/en-gb/unitedkingdom/case-ih-world/history (cit. a p. 7).

[3] CNH Industrial. *CNH Industrial — About Us*. Consultato nel 2026. 2026 (cit. a p. 7).

[4] Case IH. *Product Configurator - United Kingdom*. 2026. URL: https://www.caseih.com/en-gb/unitedkingdom/tools-resources/configurator/configurator (visitato il giorno 17/02/2026) (cit. alle pp. 7–9).

[5] Ian Robinson, Jim Webber e Emil Eifrem. *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.", 2015 (cit. alle pp. 23, 24, 27).

[6] Amazon Web Services. *Differenza tra database a grafo e database relazionali*. Accesso: 23 febbraio 2026. 2024. URL: https://aws.amazon.com/it/compare/the-difference-between-graph-and-relational-database/ (cit. a p. 24).

[7] Nadime Francis et al. «Cypher: An evolving query language for property graphs». In: *Proceedings of the 2018 international conference on management of data*. 2018, pp. 1433–1445 (cit. a p. 24).

[8] Coolshop Srl. «CoolPIM User Manual». Documentazione aziendale interna. 2025 (cit. a p. 25).

[9] Neo4j, Inc. *Graph Data Modeling Guidelines*. Consultato: marzo 2026. 2023. URL: https://neo4j.com/docs/getting-started/data-modeling/guide-data-modeling/ (cit. a p. 27).

[10] George A Miller. «WordNet: a lexical database for English». In: *Communications of the ACM* 38.11 (1995), pp. 39–41 (cit. a p. 29).

[11] Zellig S Harris. «Distributional structure». In: *Word* 10.2-3 (1954), pp. 146–162 (cit. a p. 30).

[12] Tomas Mikolov, Kai Chen, Greg Corrado e Jeffrey Dean. «Efficient estimation of word representations in vector space». In: *arXiv preprint arXiv:1301.3781* (2013) (cit. a p. 30).

[13] Jeffrey Pennington, Richard Socher e Christopher D Manning. «Glove: Global vectors for word representation». In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543 (cit. a p. 30).

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser e Illia Polosukhin. «Attention is all you need». In: *Advances in neural information processing systems* 30 (2017) (cit. alle pp. 30, 33).

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee e Kristina Toutanova. «Bert: Pre-training of deep bidirectional transformers for language understanding». In: *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 2019, pp. 4171–4186 (cit. a p. 30).

[16] Nils Reimers e Iryna Gurevych. «Sentence-bert: Sentence embeddings using siamese bert-networks». In: *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*. 2019, pp. 3982–3992 (cit. a p. 30).

[17] David Andrés e Josep Ferrer. *Issue 58: Embeddings in NLP*. Consultato il 20 Febbraio 2026. 2026. URL: `https://mlpills.substack.com/p/issue-58-embeddings-in-nlp` (cit. a p. 30).

[18] Yu A Malkov e Dmitry A Yashunin. «Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs». In: *IEEE transactions on pattern analysis and machine intelligence* 42.4 (2018), pp. 824–836 (cit. alle pp. 31, 48).

[19] Hinrich Schütze, Christopher D Manning e Prabhakar Raghavan. *Introduction to information retrieval*. Vol. 39. Cambridge University Press Cambridge, 2008 (cit. a p. 32).

[20] Tom Brown et al. «Language models are few-shot learners». In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901 (cit. a p. 33).

[21] Jason Wei et al. «Emergent abilities of large language models». In: *arXiv preprint arXiv:2206.07682* (2022) (cit. a p. 33).

[22] Ziwei Ji et al. «Survey of hallucination in natural language generation». In: *ACM computing surveys* 55.12 (2023), pp. 1–38 (cit. a p. 34).

[23] Patrick Lewis et al. «Retrieval-augmented generation for knowledge-intensive nlp tasks». In: *Advances in neural information processing systems* 33 (2020), pp. 9459–9474 (cit. a p. 35).

[24] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda e Thomas Scialom. «Toolformer: Language models can teach themselves to use tools». In: *Advances in neural information processing systems* 36 (2023), pp. 68539–68551 (cit. a p. 35).

[25] Darren Edge, T Krzyzanowski, F Basisty et al. «A Graph RAG Approach to Query-Focused Summarization». In: *arXiv preprint arXiv:2404.16130* (2024) (cit. a p. 35).

[26] Neo4j, Inc. *Intro to GraphRAG*. Ultimo accesso: 4 marzo 2026. Licenza CC BY 4.0. 2024. URL: `https://graphrag.com/concepts/intro-to-graphrag/` (cit. a p. 38).