ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica Scienza e Ingegneria

Corso di Laurea in Informatica

Quenya: un traduttore da linguaggio naturale a query SPARQL

Relatore:
Chiar.mo Prof.
Fabio Vitali

Presentata da:
Alberto Zuccari

Correlatore:
Chiar.mo Prof.
Paolo Bonora

I Sessione 2026
Anno Accademico 2025/2026

Non tocca a noi dominare tutte le maree del mondo
il nostro compito è di fare il possibile
per la salvezza degli anni nei quali viviamo

- J.R.R.Tolkien

Abstract

L'interrogazione dei database RDF richiede abitualmente competenze tecniche specialistiche, prima fra tutte la padronanza del linguaggio formale SPARQL. Per i ricercatori in ambito umanistico, questa barriera rappresenta spesso un ostacolo nell'esplorazione del patrimonio informativo.

Questa tesi documenta lo sviluppo di "Quenya", un'interfaccia intelligente costruita per il Datavault del centro di ricerca DH.ARC. Il repository ospita circa 19 milioni di triple RDF, un volume di dati difficilmente navigabile senza strumenti tecnici. Il sistema sviluppato traduce domande in linguaggio naturale direttamente in query SPARQL eseguibili, aggirando la rigidità strutturale del database a grafo.

A livello metodologico, il progetto implementa il fine-tuning di un modello linguistico compatto (attraverso l'algoritmo LoRA) combinandolo con un'architettura che vincola l'output allo schema ontologico del Datavault. Questo accorgimento previene in modo strutturale le allucinazioni sintattiche tipiche dei Large Language Models. Dal punto di vista dell'utente, l'interazione avviene tramite una singola barra di ricerca, mentre un backend difensivo gestisce in modo invisibile la traduzione e l'interrogazione dell'endpoint Blazegraph.

La validazione quantitativa, condotta misurando *precision* e *recall* su un test set *held-out*, ha registrato un'accuratezza sintattica ed esecutiva ottimale per le categorie di ricerca addestrate. I risultati confermano che un modello di piccole dimensioni, se opportunamente vincolato al proprio dominio, può abbattere la barriera tecnica d'accesso agli archivi semantici complessi.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei codici

Capitolo 1

Introduzione

Il claim, il problema e l'ambito

L'interrogazione di database RDF è un'operazione complessa che richiede competenze tecniche avanzate, in particolare la padronanza del linguaggio formale SPARQL. Nei contesti di ricerca umanistica, questa barriera può impedire ai ricercatori di esplorare e valorizzare patrimoni informativi estesi.

Il presente lavoro dimostra che un modello linguistico compatto, sottoposto a fine-tuning con la tecnica LoRA su un dataset sintetico costruito appositamente e guidato da un approccio ispirato alla Retrieval-Augmented Generation (RAG), è in grado di tradurre domande espresse in linguaggio naturale italiano in query SPARQL eseguibili su un repository RDF anche di grandi dimensioni. Il sistema permette ai ricercatori umanisti di interrogare un repository RDF senza dover scrivere SPARQL.

Il problema nasce da un'osservazione concreta: il Datavault del centro di ricerca DH.ARC (Digital Humanities Advanced Research Centre) dell'Università di Bologna ospita $\approx$ 19 milioni di risorse digitalizzate, esposte sotto forma di triple RDF[W3C14] per favorire l'interoperabilità semantica. Tuttavia, non esiste attualmente alcun meccanismo che consenta ai ricercatori del centro di interrogare questo patrimonio per contenuto o per concetto: l'unica alternativa è la navigazione manuale della struttura a container o la scrittura diretta di query SPARQL[W3C13], un linguaggio dichiarativo formale che impone vincoli rigorosi e richiede la conoscenza di ontologie e convenzioni di metadati specifiche del repository[VFQP25][Vaš24].

Questo ostacolo si articola su tre livelli distinti. Il primo è una **barriera cognitiva**: l'interazione con SPARQL esige la conoscenza di prefissi ontologici complessi (ad esempio Dublin Core, Exif ed EBUCore) anche solo per formulare una query di base, generando il fenomeno della "scatola vuota" che lascia l'utente senza orientamento. Il secondo livello è una **barriera strutturale**: l'assenza di una mappa visiva delle relazioni all'interno del grafo RDF fa sì che il ricercatore non sappia quali proprietà esistano realmente nel database né come i dati siano formalizzati nell'ontologia. Il terzo è una **barriera pratica**: la forte rigidità sintattica di SPARQL determina che un singolo errore di battitura non produca risultati, senza messaggi di errore informativi che consentano un debugging autonomo.

L'ambito del lavoro si colloca all'intersezione tra le Digital Humanities, l'accesso a Knowledge Graphs nel dominio del patrimonio culturale e la traduzione automatica Text-to-SPARQL. Il contributo si concretizza in "Quenya" (**Qu**ery **E**ngine for **N**avigating **Y**our **A**rchives), un sistema integrato nell'infrastruttura del Datavault e progettato per chi conosce il dominio umanistico ma non SPARQL.

Contesto scientifico e tecnologico

Il problema affrontato in questa tesi si colloca in un contesto scientifico e tecnologico che il capitolo 2 analizza in modo esteso.

Sul piano dello **stato dell'arte**, la letteratura documenta un divario strutturale tra i bisogni esplorativi degli studiosi umanisti e la rigidità sintattica dei sistemi di interrogazione tradizionali. I ricercatori nel dominio delle Digital Humanities non procedono per query esatte: adottano strategie di ricerca non lineari, teorizzate da Marcia Bates con il modello *berrypicking*[Bat89], in cui l'obiettivo informativo si raffina progressivamente durante l'esplorazione stessa. In questo contesto, la ricerca favorisce la *serendipity*, la scoperta inattesa di pattern rilevanti durante la navigazione dei dati, che gli strumenti tradizionali basati su query formali ostacolano.

Sul piano dei **fondamenti teorici**, l'architettura dell'informazione giustifica il ricorso a un'interfaccia conversazionale mediata da LLM. Morville e Rosenfeld[MR08] hanno dimostrato che trattare il recupero di informazioni come un problema puramente algoritmico è inadeguato quando si opera su contenuti semistrutturati: i sistemi tradizionali sono progettati per recuperare *dati esatti*, mentre gli studiosi ricercano *concetti*, *idee* e *relazioni*. Rosati[Ros07] completa il quadro analizzando il carico cognitivo attraverso due principi: la *legge del minimo sforzo*, per cui gli utenti accettano risultati di qualità inferiore piuttosto che affrontare strategie di ricerca più impegnative, e la *Legge di Hick*[Ros07], per cui esporre l'utente a un numero elevato di opzioni non organizzate genera paralisi decisionale. Un'interfaccia basata su un singolo campo di testo azzera entrambi questi effetti.

Sul piano **tecnologico**, due innovazioni rendono oggi possibile una soluzione che bilanci flessibilità e affidabilità. La prima è il paradigma Retrieval-Augmented Generation (RAG)[LPP+20], originariamente concepito per compiti di question-answering su testi non strutturati: nel nostro caso, anziché recuperare documenti testuali, il sistema recupera frammenti di schema RDF, esempi di predicati e descrizioni ontologiche, vincolando la generazione del modello allo schema reale del Datavault e riducendo il rischio di allucinazioni su classi o proprietà inesistenti. La seconda è la tecnica LoRA (*Low-Rank Adaptation*)[HSW+22], che consente il fine-tuning di un modello linguistico compatto su un dataset di dominio specifico senza caricare in memoria l'intero modello.

La soluzione proposta

La soluzione proposta, descritta in dettaglio nei capitoli 3 and 4, è il Quenya Query Builder: un sistema che traduce interrogazioni espresse in italiano naturale in query SPARQL eseguibili direttamente sull'endpoint Blazegraph del Datavault, mostrando poi i risultati filtrati all'interno di un'interfaccia grafica navigabile.

Dal punto di vista architetturale, il sistema si articola in tre componenti. Il primo è il **frontend** (Data Vault Explorer), un'applicazione WEB basata su React che raccoglie l'input testuale dell'utente e renderizza i risultati in formato navigabile. Il punto d'ingresso è la *Quenya Search Bar*, un singolo campo di testo in linguaggio naturale che sostituisce la complessità di form a filtri multipli o di endpoint SPARQL esposti, riducendo il carico cognitivo descritto nella sezione precedente. L'interfaccia è simile a un motore di ricerca, ma il backend interpreta le sfumature del linguaggio naturale tramite il modello. Il secondo componente è il **backend** (Quenya Query Builder + RAG Engine), implementato in FastAPI: riceve la domanda, recupera lo schema RDF, genera la query SPARQL tramite il modello e gestisce gli errori semantici in modo difensivo. Il terzo è il **data layer** (Blazegraph + Schema RDF), che immagazzina le triple ed esegue l'interrogazione generata.

Il flusso dell'interazione segue quattro fasi invisibili all'utente: l'input in linguaggio naturale, l'elaborazione semantica da parte del modello Quenya (che, conoscendo l'ontologia del Datavault, produce autonomamente la query SPARQL corrispondente), l'esecuzione della query su Blazegraph, e la restituzione dei risultati nell'interfaccia grafica. L'utente non vede il codice SPARQL: la complessità sintattica e ontologica resta nel backend.

Il cuore del sistema è il modello Quenya, ottenuto tramite fine-tuning del modello Phi-3-mini con un adapter LoRA. L'addestramento è stato condotto su un dataset costruito appositamente attraverso un generatore a template parametrici, capace di produrre coppie (domanda in italiano, query SPARQL corretta) distribuite su 13 categorie di interrogazione. Ho definito personalmente queste categorie per mappare le ricerche più comuni di uno studioso umanistico all'interno del Datavault. A queste si aggiunge la gestione delle richieste fuori dominio (Out-of-Domain). La coerenza tra fase di addestramento e fase di inferenza è garantita da un system prompt identico, byte per byte, in entrambi i contesti: 8 regole assolute che vincolano la struttura sintattica della query generata (un'unica struttura `SELECT DISTINCT`, predicati fissi per ciascun intento, gestione esplicita dei casi fuori dominio), riducendo lo spazio delle risposte plausibili e minimizzando il rischio di allucinazioni strutturali.

La valutazione

Il prototipo è stato valutato su due piani, descritti nel capitolo 5: l'**efficienza**, relativa alle prestazioni quantitative misurabili, e l'**efficacia**, relativa al miglioramento qualitativo apportato dal sistema.

La valutazione quantitativa è stata condotta secondo una pipeline a due fasi. Nella prima fase è stato generato un test set su 500 domande, e son stati passati al backend eseguite su GPU NVIDIA T4 per generare la query SPARQL corrispondente a ciascuna delle domande del test set. Nella seconda fase, eseguita localmente su CPU, sia la query generata sia quella di riferimento vengono eseguite contro l'endpoint Blazegraph reale del Datavault, e i rispettivi insiemi di risultati vengono confrontati per calcolare precision, recall e F1.

Il **100% delle query generate è stato eseguito con successo** contro Blazegraph, senza errori di sintassi o di esecuzione, il che indica che le regole del system prompt

vincolano il modello a generare query sintatticamente valide. Le metriche di retrieval (precision, recall, F1) raggiungono il valore di 1.000 su tutte le 7 categorie testate (che ho selezionato tra le 13 per mettere alla prova i casi più complessi), con un riconoscimento del 100% anche per le richieste Out-of-Domain. Il tempo medio di generazione per query è di 7.45 secondi su GPU T4, con la maggioranza delle query generata in 7–9 secondi.

Questi risultati vanno però qualificati. Un'ispezione diretta delle 500 coppie rivela che tutte le query generate sono identiche, carattere per carattere, alla rispettiva query di riferimento. Questo fenomeno è spiegabile alla luce di come il test set è stato costruito: le domande e le query di riferimento sono prodotte dallo stesso sistema di template parametrici usato per il training. Inoltre, le 8 regole del system prompt (che ho scritto appositamente per guidare il modello) vincolano fortemente la forma sintattica della query attesa, riducendo lo spazio delle risposte plausibili a un insieme ristretto e, per costruzione, quasi deterministico. Di conseguenza, la valutazione qui presentata misura principalmente la capacità del modello di rispettare in modo affidabile la grammatica e le regole imposte, più che la sua capacità di decodificare formulazioni libere e impreviste di un ricercatore umano. La prova con domande poste spontaneamente da utenti reali resta un obiettivo degli sviluppi futuri.

Sviluppi futuri

Il capitolo 6 dettaglia gli sviluppi futuri del sistema, organizzati per priorità decrescente.

Gli interventi più urgenti riguardano l'estensione della valutazione alle categorie di query non rappresentate nel test set attuale (negazione, OR logico, paginazione custom), la trasformazione del namespace EXIF hardcoded in un parametro di configurazione centralizzato, e la misurazione dei tempi di inferenza reali sul backend di produzione in CPU, attualmente non disponibile.

Manca poi un test di usabilità con utenti reali del dominio DH.ARC: non esiste ancora una misura empirica del vantaggio percepito da un ricercatore umanista, sia in termini di tempo risparmiato sia di soddisfazione d'uso. Ulteriori estensioni riguardano il supporto multilingue, l'aggiunta di un'interfaccia di correzione manuale della query generata per gli utenti più esperti, e l'ampliamento della copertura semantica dello schema a predicati non ancora sfruttati.

Guardando più avanti, le estensioni includono il passaggio da un prompt statico a un vero meccanismo RAG dinamico che recuperi solo i frammenti di schema rilevanti per ciascuna domanda; l'integrazione di modelli di visione (ad esempio embedding CLIP-like) per una ricerca per contenuto visivo, non limitata ai metadati; e l'evoluzione verso un sistema federato capace di interrogare più triplestore appartenenti a istituzioni diverse del patrimonio culturale.

Struttura della dissertazione

La dissertazione è organizzata come segue:

- Il capitolo 2 inquadra il problema dal punto di vista teorico, analizzando lo stato dell'arte della ricerca semantica nel patrimonio culturale, i principi

dell'architettura dell'informazione che governano il carico cognitivo dell'utente, e le tecnologie abilitanti (modelli linguistici compatti, tecnica RAG, fine-tuning con LoRA).

- Il capitolo 3 presenta il sistema dal punto di vista dell'utente finale: il dominio di applicazione (DH.ARC e il Datavault), il profilo dei ricercatori target, l'architettura funzionale, l'interfaccia conversazionale e tre casi d'uso concreti con le relative schermate.
- Il capitolo 4 dettaglia l'architettura tecnica e le scelte implementative: lo stack tecnologico, la pipeline di fine-tuning del modello, l'architettura del backend FastAPI e la struttura del dataset di addestramento.
- Il capitolo 5 sottopone il sistema a una valutazione quantitativa (tasso di esecuzione, metriche di retrieval, tempi di generazione) e qualitativa (miglioramento rispetto alle soluzioni precedenti), con una discussione critica dei limiti metodologici.
- Il capitolo 6 chiude il lavoro con un bilancio dei punti di forza e dei limiti del sistema, e delinea gli sviluppi futuri ordinati per priorità.

Capitolo 2

Dall'interrogazione formale alla ricerca semantica

Il capitolo esplora il background scientifico e tecnologico alla base del lavoro proposto. Partiamo dal *problema di fondo*: come fornire accesso content-aware a repository RDF di dimensioni massive per utenti privi di competenze tecniche formali?

La risposta richiede una comprensione su tre livelli di complessità:

1. **Lo stato dell'arte**: come gli approcci precedenti (form rigidi, endpoint SPARQL nudi) falliscono nel dominio umanistico;
2. **I fondamenti teorici**: i principi dell'architettura dell'informazione che guidano un'alternativa;
3. **Le tecnologie emergenti**: come Retrieval-Augmented Generation (RAG) e Large Language Models (LLM) abilitano una soluzione più flessibile.

Procederemo così: nel paragrafo 2.1 analizziamo il divario tra i bisogni esplorativi degli studiosi umanisti e la rigidità sintattica dei sistemi tradizionali. Successivamente, nel paragrafo 2.2, esaminiamo i principi fondativi dell'architettura dell'informazione che giustificano il ricorso agli LLM. Infine, nel paragrafo 2.3, descriviamo il paradigma RAG come soluzione tecnologica che bilancia flessibilità e affidabilità.

2.1 Lo stato dell'arte

2.1.1 Il contesto: Digital Humanities e la barriera della ricerca

Nel dominio delle Digital Humanities (DH) e della gestione del patrimonio culturale, la ricerca all'interno di repository RDF è un ostacolo per chi manca di competenze tecniche avanzate. Come evidenziato in letteratura, lavorare con strumenti come SPARQL impone vincoli rigorosi[VFQP25] e richiede una familiarità profonda con ontologie complesse e convenzioni di metadati[Vaš24].

Tali approcci si basano su query create manualmente, che risultano fragili e difficili da adattare a esigenze mutevoli[Mar25]. Questo pone il ricercatore di fronte all'ostacolo

cognitivo della "scatola vuota": chi possiede un bisogno informativo vago o in via di definizione viene paralizzato dalla necessità di dichiarare tutto a priori.

2.1.2 Il divario tra ricerca strutturata e ricerca esplorativa

Alla base dei tradizionali sistemi di Information Retrieval (IR) vi è un assunto fondamentale: che l'utente inizi l'interazione con un bisogno informativo perfettamente strutturato. Nel dominio umanistico, tuttavia, l'utente procede per esplorazione.

Questo comportamento è stato teorizzato da Marcia Bates con il modello *berry-picking*[Bat89]: una strategia di ricerca non lineare in cui gli studiosi si muovono iterativamente, aggiustando la rotta e raffinando gli obiettivi man mano che acquisiscono nuove informazioni durante l'esplorazione. In questo scenario, la ricerca esplorativa favorisce la *serendipity*, la scoperta inattesa di pattern rilevanti durante la navigazione dei dati, che gli strumenti tradizionali ostacolano.

Considerando il netto contrasto tra la rigidità sintattica di SPARQL e la natura fluida ed evolutiva della ricerca umanistica, serve un modo diverso di interagire con i dati, che consenta ai ricercatori di esprimere bisogni emergenti senza dover ricorrere a linguaggi formali.

2.1.3 La "scatola vuota" e le tre barriere di SPARQL

Costringere ricercatori umanisti a esprimere bisogni complessi tramite endpoint SPARQL genera un ostacolo su tre livelli:

1. **Barriera cognitiva**: L'interazione esige la scrittura in un linguaggio dichiarativo formale e l'obbligo di conoscere complessi prefissi ontologici (es. `dc:`, `exif:`, `ebucore:`) solo per abbozzare una query di base.
2. **Barriera strutturale**: L'assenza di una mappa visiva delle relazioni all'interno del grafo RDF fa sì che l'utente non sappia quali proprietà esistano realmente nel database né come i dati siano formalizzati nell'ontologia.
3. **Barriera pratica**: La forte rigidità sintattica determina che un singolo errore di battitura produca un set di risultati vuoto. L'assenza di messaggi di errore parlanti impedisce qualsiasi tentativo di debugging autonomo.

Studi sull'usabilità delle interfacce per Knowledge Graphs confermano che linguaggi come SPARQL impongono vincoli rigidi e richiedono una conoscenza esplicita degli schemi, competenze estranee ai professionisti del patrimonio culturale[VFQP25].

2.2 Fondamenti di architettura dell'informazione

La progettazione di un sistema di accesso per un database RDF non può limitarsi alla pura efficienza computazionale del recupero dati. Deve invece rispondere ai principi dell'*architettura dell'informazione*, ovvero la disciplina che studia come rendere le informazioni realmente trovabili e usabili.

In un contesto umanistico, costringere l'utente a interfacciarsi con la complessità dello schema sottostante attraverso filtri rigidi o linguaggi formali genera un ostacolo cognitivo. Analizziamo i principi che giustificano il ricorso a un'interfaccia conversazionale mediata da LLM.

2.2.1 L'illusione della ricerca algoritmica

I sistemi di reperimento basati su query formali (SQL, SPARQL) assumono erroneamente che l'utente sappia sempre cosa cercare e come tradurre il proprio bisogno in un costrutto logico. Peter Morville e Louis Rosenfeld hanno smontato questo assunto:

> "Consideriamo questo modello pericoloso perché basato su un equivoco: che trovare informazioni sia un problema diretto, che può essere risolto da un semplice approccio algoritmico. Dopo tutto, abbiamo risolto la sfida del recupero dati — che, ovviamente sono fatti e cifre — con tecnologie database come SQL. Così procede la teoria, si trattano idee e concetti incorporati nei documenti testuali semistrutturati nello stesso modo" [MR08]

I sistemi algoritmici tradizionali falliscono nel dominio umanistico perché sono progettati per recuperare *dati esatti*, mentre gli studiosi ricercano *concetti*, *idee* e *relazioni*. Tali concetti sono per loro natura ambigui e sfumati. Un LLM colma questo divario: accetta l'ambiguità del linguaggio naturale e la converte nella sintassi rigida richiesta dalla macchina.

2.2.2 Recall, Precision e il compromesso ineluttabile

Un problema critico delle interfacce tradizionali a filtri è l'impossibilità di calibrare dinamicamente il raggio della ricerca. Morville e Rosenfeld spiegano che ogni ricerca è un compromesso matematico tra due metriche inversamente correlate:

> $$\texttt{recall} = \frac{|\texttt{doc rilevanti recuperati}|}{|\texttt{doc nella raccolta}|}$$
>
> $$\texttt{precision} = \frac{|\texttt{doc rilevanti recuperati}|}{|\texttt{doc rilevanti della raccolta}|}$$
>
> "[...] Non sarebbe bello poter ottenere contemporaneamente elevati valori di recall e precision? Purtroppo non si può avere botte piena e moglie ubriaca: recall e precision sono inversamente correlati" [MR08]

In un'interfaccia a maschera con decine di campi e filtri ontologici, l'utente è costretto a dichiarare a priori il livello di precisione desiderato. Se sbaglia un filtro, ottiene zero risultati (eccesso di Precision); se ne omette uno, viene sommerso da migliaia di risultati irrilevanti (eccesso di Recall).

In una conversazione in linguaggio naturale con un LLM, invece, l'aggiustamento avviene in modo iterativo e fluido. L'utente può iniziare con una richiesta esplorativa e ampia (es. "Mostrami tutti i manoscritti del XIV secolo", privilegiando il Recall) e, osservando i primi risultati, raffinare il tiro conversando ("Filtra solo quelli con miniature e digitalizzati ad alta risoluzione", aumentando la Precision). L'LLM permette di variare queste metriche senza costringere l'utente a riscrivere manualmente la query.

2.2.3 Carico cognitivo: la Legge del minimo sforzo e la Legge di Hick

Dal punto di vista del carico cognitivo, esporre all'utente l'immensa topologia di un Knowledge Graph attraverso decine di menu a tendina o l'obbligo di conoscere specifiche ontologie porta alla frustrazione e può sfociare nell'abbandono del sistema.

Luca Rosati analizza questo fenomeno attraverso due principi cognitivi fondamentali. Il primo è la *legge del minimo sforzo*:

> "La legge del minimo sforzo (least effort) regola quindi il nostro sistema di information seeking, al punto che spesso accettiamo anche contenuti di bassa qualità o affidabilità (se più facili da ottenere e usare) pur di non ricorrere a strategie di ricerca attive, le quali comportano necessariamente sforzi, tempi e competenze maggiori" [Ros07]

Se l'interfaccia richiede la fatica di apprendere prefissi RDF o di comprendere uno schema dati complesso, è plausibile che l'utente rinuncerà alla ricerca.

Il secondo principio è la *Legge di Hick*, la quale afferma che, in presenza di n alternative equiprobabili, il tempo necessario per selezionarne una è proporzionale (a meno di una costante additiva) al logaritmo in base 2 del numero di scelte più 1:

$$\text{tempo} = a + b \log_2(n + 1)$$

I coefficienti a e b dipendono da specifiche condizioni al contorno, tra cui le modalità di presentazione delle opzioni e il grado di familiarità dell'utente [Ros07].

Tuttavia, Rosati sottolinea che, qualora le alternative "siano presentate in modo confuso" o costituiscano "una lista disordinata" priva di un criterio logico riconoscibile, il tempo di decisione scala in modo lineare, aumentando il carico cognitivo fino a divenire insostenibile e causando paralisi decisionale. Esporre l'utente a un form SPARQL nudo o a una maschera di ricerca avanzata con decine di menu a tendina equivale a questo: innescare un "paradosso della scelta" paralizzante.

Un'interfaccia conversazionale basata su LLM nasconde questa complessità strutturale nel backend. L'utente si trova di fronte a un singolo "spazio" (un box di chat),

azzerando l'effetto della Legge di Hick e assecondando la Legge del minimo sforzo. Non dovendo scegliere tra decine di opzioni incomprensibili, il ricercatore delega la traduzione ontologica all'LLM e si concentra sull'analisi delle risorse.

2.2.4 Paradigma classico: navigazione e filtri rigidi

L'architettura dell'informazione si definisce come il design strutturale di ambienti informativi finalizzato a plasmare l'usabilità e la trovabilità[MR08]. Poggia su quattro sistemi interconnessi:

(i) sistemi di organizzazione, che categorizzano le informazioni;

(ii) sistemi di etichettatura, che ne determinano la rappresentazione linguistica;

(iii) sistemi di navigazione, che consentono l'orientamento nello spazio;

(iv) sistemi di ricerca, che permettono interrogazioni dirette.

Applicato al Datavault, l'interrogazione tradizionale mediante filtri rigidi o SPARQL presenta tre limiti strutturali:

1. Pretende conoscenza a priori di stringhe esatte, incompatibile con la ricerca umanistica [Vaš24];
2. Non gestisce sinonimia, sfumature e vaghezza del linguaggio naturale;
3. Obbliga la formulazione di query multi-criterio in sintassi rigide.

Il risultato è che l'informazione è presente nel database ma l'utente non riesce a trovarla.

2.2.5 Il nuovo paradigma: ricerca semantica con LLM

Un LLM cambia il modo di esplorare i dati semantici: converte il linguaggio naturale in query SPARQL strutturate, agendo come intermediario tra il ricercatore e il triplestore.

Il ricercatore non deve più imparare il linguaggio della macchina: è il sistema che si adatta all'utente. La complessità cognitiva e sintattica passa dall'utente al sistema.

Questo si lega al concetto di "findability" (trovabilità)[Ros07]. Lo scopo del sistema è rendere le risorse trovabili, non solo accessibili, nel momento del bisogno.

La ricerca semantica mediata dall'LLM aumenta la trovabilità per tre ragioni:

- abbassa drasticamente la soglia di accesso all'informazione, permettendo a chiunque di fare una domanda diretta al database;
- il modello gestisce in modo nativo la vaghezza, l'ambiguità e i sinonimi intrinseci nel linguaggio umano;
- consente di esprimere bisogni informativi multi-criterio complessi, impossibili da supportare attraverso form a tendina.

Il paradigma basato su LLM presenta però sfide note: il rischio di allucinazioni (invenzione di predicati inesistenti), la dipendenza dall'accuratezza del prompt di sistema, e l'elevato costo computazionale delle chiamate ai modelli generativi. Queste metriche verranno analizzate in dettaglio nel capitolo 5.

2.3 Retrieval-Augmented Generation (RAG) come soluzione tecnologica

2.3.1 Il paradigma RAG tradizionale

Retrieval-Augmented Generation (RAG) è un paradigma che combina la potenza dei modelli generativi con l'affidabilità del recupero di informazioni da fonti strutturate. La struttura di base è semplice ma efficace:

1. **Ingresso**: l'utente pone una domanda in linguaggio naturale;
2. **Retrieval**: un modulo di ricerca estrae documenti o frammenti rilevanti da una base di conoscenza strutturata;
3. **Augmentation**: il contesto recuperato viene fornito al modello generativo come "prompt esteso";
4. **Generation**: il modello generativo produce una risposta grounded nei documenti recuperati, riducendo le allucinazioni.

Il modello è vincolato a operare all'interno di uno spazio di conoscenza ben definito, il che riduce il rischio di allucinazioni.

2.3.2 RAG applicato a Knowledge Graphs

Nel nostro caso, il "corpus" da cui recuperare informazioni è uno **schema RDF strutturato**. L'applicazione di RAG a Knowledge Graphs richiede una variante: anziché recuperare documenti di testo, recuperiamo *frammenti di schema*, *esempi di predicati*, e *descrizioni di ontologie*.

Quindi il flusso diventa:

1. **Ingresso**: domanda dell'utente (es. "Mostrami tutte le immagini digitalizzate con una Nikon nel 2022");
2. **Retrieval**: il sistema accede allo schema RDF del Datavault e recupera i predicati rilevanti (es. `exif:make`, `exif:dateTime`);
3. **Augmentation**: lo schema e gli esempi di predicati vengono forniti al modello Quenya come parte del prompt di sistema;
4. **Generation**: il modello Quenya produce una query SPARQL coerente con lo schema;
5. **Esecuzione**: la query viene eseguita su Blazegraph e i risultati vengono restituiti all'utente.

Il risultato combina la flessibilità del linguaggio naturale con l'affidabilità di una fonte strutturata.

2.3.3 Fine-tuning e LoRA per Domain Adaptation

Un miglioramento ulteriore consiste nel fine-tuning del modello LLM su un dataset specifico del dominio. Anziché usare un LLM generico, possiamo adattare un modello più piccolo (es. `Phi-3-mini`) a generare query SPARQL corrette per il Datavault specifico.

Tecniche come LoRA (*Low-Rank Adaptation*) permettono un fine-tuning efficiente senza memorizzare l'intero modello in memoria. Il modello apprende:

1. La sintassi SPARQL corretta per il Datavault;
2. L'ontologia specifica (predicati, tipi di dato, convenzioni di naming);
3. Strategie di generazione robuste (evitare predicati inesistenti, gestire date, etc.);
4. Quando e come segnalare query out-of-domain.

Questo riduce il rischio di allucinazioni e migliora l'accuratezza della generazione.

2.4 Conclusioni del capitolo

In questo capitolo abbiamo analizzato il contesto scientifico e tecnologico alla base del lavoro:

1. Gli approcci tradizionali (SPARQL nudo, filtri rigidi) falliscono nel dominio umanistico perché impongono barriere cognitive, strutturali e pratiche;
2. I modelli teorici dell'architettura dell'informazione (berrypicking, precision e recall, carico cognitivo) giustificano il ricorso a un'interfaccia conversazionale mediata da LLM;
3. Le tecnologie emergenti (RAG, fine-tuning con LoRA) rendono possibile un'implementazione affidabile e accurata di tale interfaccia.

Il capitolo 3 presenta nel concreto il Quenya Query Builder: il dominio di applicazione (DH.ARC e il Datavault), il profilo degli utenti target, l'architettura funzionale del sistema, e i casi d'uso che ne dimostrano il valore nel contesto umanistico.

Capitolo 3

Quenya

Dopo aver esaminato il contesto nel capitolo 2, presentiamo ora il prototipo di Quenya (**Qu**ery **E**ngine for **N**avigating **Y**our **A**rchives), un query builder sviluppato per sopperire alle problematiche riscontrate.

Questo capitolo è organizzato secondo una prospettiva *user-centered*: partiamo dal dominio di applicazione (DH.ARC e il Datavault) e dal profilo degli utenti target (paragrafo 3.1), passiamo all'architettura funzionale e all'interfaccia del sistema (paragrafo 3.2), e concludiamo con tre scenari d'uso concreti (paragrafo 3.3) arricchiti dalle relative schermate, per dimostrare il valore aggiunto della nostra proposta.

3.1 Il dominio e gli utenti target

3.1.1 Il contesto operativo: DH.ARC e il Datavault

Il Digital Humanities Advanced Research Centre (DH.ARC) è un polo di ricerca dell'Università di Bologna afferente al Dipartimento di Filologia Classica e Italianistica (FICLIT), nato con l'obiettivo di "connettere studenti, ricercatori, personale IT e professori del FICLIT e del Dipartimento di Informatica - Scienza e Ingegneria (DISI)"[ALM26]. Il centro ha lo scopo di fungere da hub per gli studiosi e le istituzioni, supportandoli "nella progettazione, sviluppo e manutenzione di progetti di ricerca DH" e promuovendo iniziative innovative nel campo umanistico.

All'interno di questo contesto, una delle infrastrutture tecnologiche di conservazione è il *Datavault*, un repository digitale basato sull'architettura Fedora, progettato per supportare l'archiviazione a lungo termine, la conservazione e la gestione strutturata delle risorse digitalizzate. Dal punto di vista tecnico, il repository espone le proprie risorse sotto forma di triple RDF per favorire l'interoperabilità semantica e l'integrazione con strumenti di analisi avanzata.[FD25] Tuttavia, il Datavault ospita attualmente grande mole di dati, dell'ordine di milioni di triple RDF fortemente interconnesse e strutturalmente molto simili tra loro. Questa vastità e omogeneità sintattica rende la ricerca e il recupero di specifiche risorse un'operazione complessa.

3.1.2 Profilo dell'utente finale: i ricercatori del centro

Profilo disciplinare e competenze tecniche

Gli utenti di riferimento del centro DH.ARC sono docenti, ricercatori, assegnisti e studenti di discipline prettamente umanistiche, quali filologia, storia, storia dell'arte e archeologia.[PT22] Questi profili hanno forti competenze nel dominio delle fonti culturali e nell'analisi dei testi critici, ma tendono ad essere privi di un background tecnico/informatico.

La loro alfabetizzazione digitale nel campo dell'Information Retrieval si limita generalmente all'impiego di strumenti "pronti all'uso": cataloghi OPAC, database relazionali con maschere pre-configurate o motori di ricerca generici; raramente hanno familiarità con triplestore RDF, SPARQL, o Knowledge Graphs.

Caratterizzazione dei bisogni informativi

In questo ecosistema, gli utenti non ricercano identificativi esatti o risposte chiuse; adottano comportamenti di ricerca esplorativa, collegando concetti e scoprendo associazioni non immediate tra i documenti. Interrogando il Datavault, gli umanisti del centro necessitano di porre interrogazioni cross-dominio articolate, formulando mentalmente domande come:

> *"Quali immagini di codici medievali del XIV secolo sono state digitalizzate nel corso del progetto X?"*

o ancora:

> *"Mostrami tutti i manoscritti attribuiti a un determinato autore e digitalizzati nell'ultimo anno".*

3.2 Architettura Funzionale e Interfaccia Utente

3.2.1 L'interfaccia: come si usa il Data Vault Explorer

Per superare le barriere legate all'alto carico cognitivo e alla rigidità formale descritte nel capitolo precedente, l'applicativo è stato esteso introducendo un modulo di interazione minimalista e conversazionale.

Invece di dover navigare attraverso complessi moduli a filtri multipli o di dover studiare a priori l'ontologia sottostante per formulare interrogazioni dirette, l'utente ha ora a disposizione un singolo e intuitivo punto di ingresso: la *Quenya Search Bar*. L'interfaccia è simile ai motori di ricerca, ma il backend interpreta le sfumature del linguaggio naturale traducendo l'intento del ricercatore in una query SPARQL eseguibile.

Figura 3.1: L'interfaccia principale del Data Vault Explorer con la Quenya Search Bar.

Come visibile in figura 3.1, l'utilizzo del sistema si riduce a un'azione elementare: digitare il proprio bisogno informativo in italiano naturale. Il sistema restituisce i risultati strutturandoli in una lista di risorse, permettendo un'esplorazione fluida e intuitiva dei dati.

3.2.2 Il flusso dell'interazione

Dal punto di vista dell'utente, l'interazione con il sistema si articola secondo un flusso strutturato, progettato per astrarre la complessità tecnica sottostante. Questo processo è schematizzato nello schema a fianco.

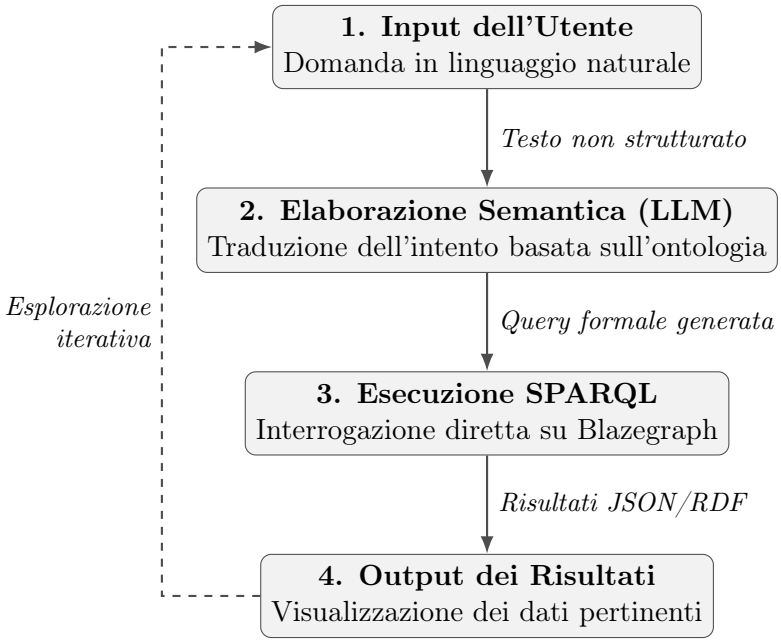


Figura 3.2: Schema del flusso di interazione tra utente, Quenya Query Builder e Datavault.

Come illustrato nello schema, le fasi del processo sono le seguenti:

1. **Input**: l'utente digita il proprio bisogno informativo in italiano in forma libera. In questa fase non vi sono campi strutturati da compilare né regole di sintassi da rispettare.
2. **Elaborazione**: il testo naturale viene inviato al modello Quenya fine-tuned che, essendo a conoscenza dello schema RDF e dell'ontologia del Datavault, produce autonomamente la query SPARQL corrispondente.
3. **Esecuzione e Output**: la query SPARQL viene inoltrata a Blazegraph. L'utente non è esposto al codice formale (salvo esplicita richiesta), ma visualizza esclusivamente i risultati pertinenti nell'interfaccia grafica.

3.2.3 I componenti del sistema e le loro responsabilità

Il sistema si articola in tre componenti principali:

1. **Frontend (Data Vault Explorer)**: raccoglie l'input testuale e renderizza i risultati in formato navigabile.
2. **Backend (Quenya Query Builder + RAG Engine)**: il componente centrale del sistema. Riceve la domanda, recupera lo schema RDF, genera la query SPARQL tramite il modello e gestisce gli errori semantici (evitando le allucinazioni).
3. **Data Layer (Blazegraph + Schema RDF)**: immagazzina i milioni di triple ed esegue l'interrogazione generata dal backend.

3.3 Casi d'uso principali ed esempi

Descriviamo tre scenari d'uso concreti che dimostrano l'efficacia del sistema presso DH.ARC.

3.3.1 Caso d'uso 1: ricerca per caratteristiche di acquisizione

Attore: un ricercatore che vuole individuare immagini acquisite con specifiche caratteristiche tecniche.
Input testuale:

`‘‘Filtra i risultati mostrandomi le risorse acquisite con il dispositivo NextScan’’`

```
PREFIX exif: <http://wwww3org/2003/12/exif/ns#>
PREFIX fedora: <http://fedora.info/definitions/v4/repository#>
PREFIX ebucore: <http://www.ebu.ch/metadata/ontologies/ebucore/ebucore#>
PREFIX ldp: <http://www.w3.org/ns/ldp#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT DISTINCT ?image WHERE {
    ?image exif:make ?make .
    FILTER(CONTAINS(LCASE(?make), "nextscan"))
}
LIMIT 20
```

Code 3.1: Caso d'uso 1: Query SPARQL generata e sanitizzata

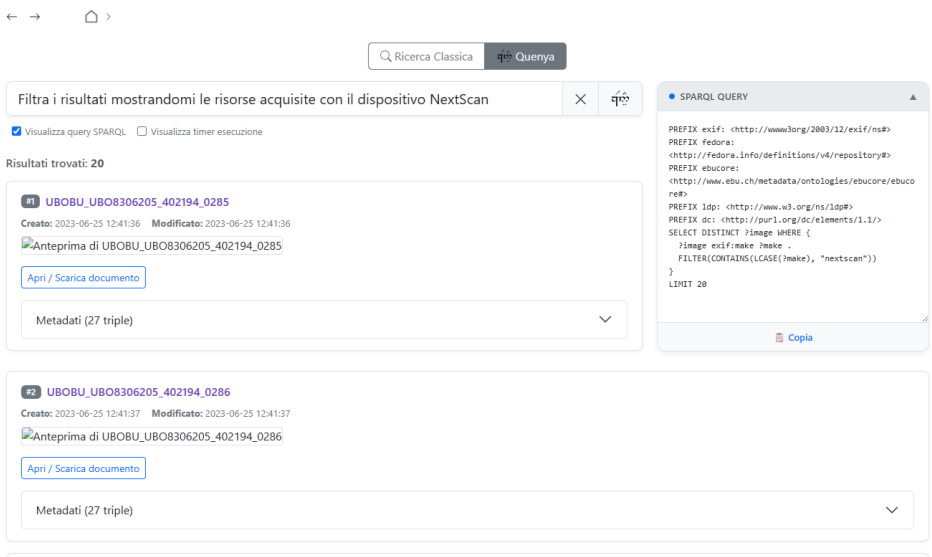


Figura 3.3: Caso d'uso 1: Estrazione delle risorse corrispondenti al dispositivo NextScan.

Risoluzione del problema: Oltre a delegare all'LLM l'onere della traduzione del linguaggio naturale nel predicato corretto (`exif:make`), questo scenario dimostra la robustezza dell'architettura.

3.3.2 Caso d'uso 2: disambiguazione semantica dell'autore

Attore: un responsabile di un progetto di digitalizzazione o ricercatore interessato alla provenienza materiale dello scatto.
Input testuale:

"Filtra le risorse mostrandomi i file in cui l'autore dello scatto è University"

```
PREFIX exif: <http://wwww3org/2003/12/exif/ns#>
PREFIX fedora: <http://fedora.info/definitions/v4/repository#>
PREFIX ebucore: <http://www.ebu.ch/metadata/ontologies/ebucore/ebucore#>
PREFIX ldp: <http://www.w3.org/ns/ldp#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT DISTINCT ?image WHERE {
    ?image exif:artist ?artist .
    FILTER(CONTAINS(LCASE(?artist), "university"))
}
LIMIT 20
```

Code 3.2: Caso d'uso 2: Query SPARQL per disambiguazione autore

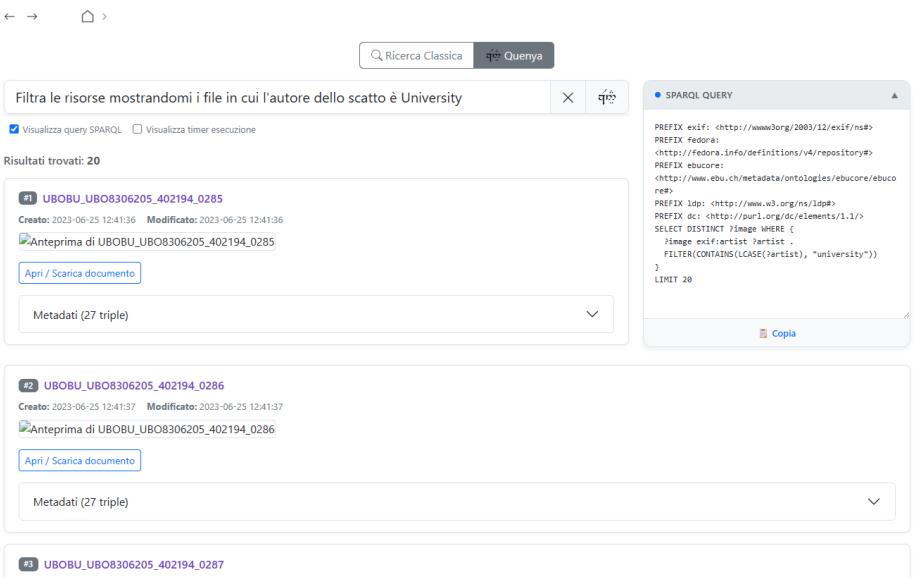


Figura 3.4: Caso d'uso 2: Disambiguazione semantica dell'autore e visualizzazione dei risultati.

Risoluzione del problema: Questo scenario evidenzia la capacità dell'LLM di operare una corretta disambiguazione semantica. Il modello comprende l'intento dell'utente distinguendo l'autore materiale dello scatto (`exif:artist`) dall'utente responsabile del caricamento nel Data Vault (`fedora:createdBy`).

3.3.3 Caso d'uso 3: ricerca multi-criterio con filtro temporale

Attore: un ricercatore che vuole analizzare la produzione digitale filtrandola all'interno di un preciso arco temporale.
Input testuale:

`"Mostrami tutte le risorse comprese tra l'anno 2023 e il 2026"`

```
PREFIX exif: <http://wwww3org/2003/12/exif/ns#>
PREFIX ebucore: <http://www.ebu.ch/metadata/ontologies/ebucore/ebucore#>
PREFIX fedora: <http://fedora.info/definitions/v4/repository#>
PREFIX ldp: <http://www.w3.org/ns/ldp#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT DISTINCT ?image WHERE {
    ?image exif:dateTime ?date .
    FILTER(STR(?date) >= "2023" && STR(?date) <= "2026")
}
LIMIT 20
```

Code 3.3: Caso d'uso 3: Query SPARQL per ricerca con filtro temporale

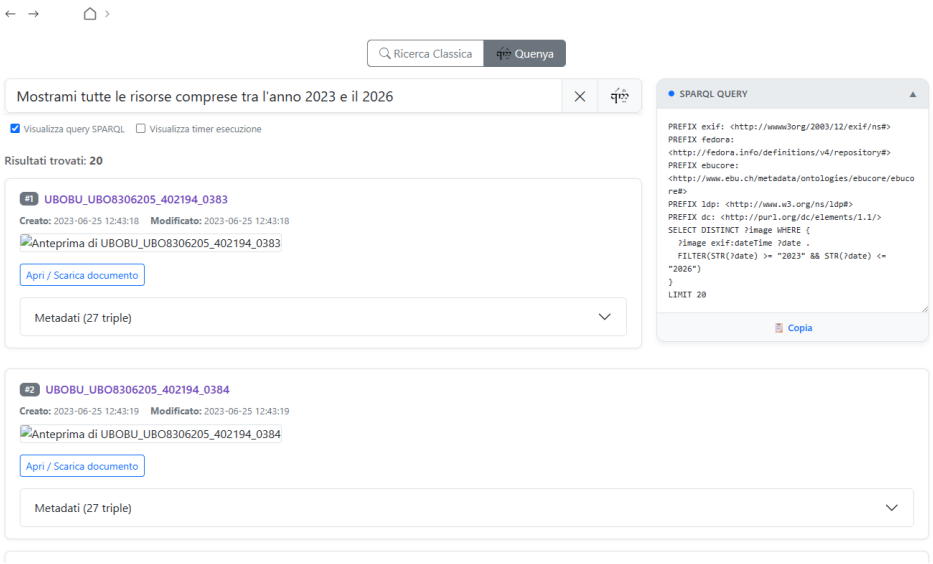


Figura 3.5: Caso d'uso 3: Applicazione logica del range temporale ed estrazione delle immagini.

Risoluzione del problema: L'estrazione basata su range temporali richiede una sintassi SPARQL rigorosa e insidiosa per un utente non tecnico. Il sistema individua automaticamente il predicato `exif:dateTime` e costruisce i costrutti logici di comparazione testuale (`>=` e `<=`).

3.4 Conclusioni del capitolo

In questo capitolo abbiamo presentato il prototipo e la sua interfaccia, rispondendo attivamente alle criticità sollevate nel capitolo 2:

1. Il problema della **"scatola vuota"** e la paralisi decisionale (Legge di Hick) sono stati superati adottando un'interfaccia basata unicamente sull'input testuale naturale.
2. L'obbligo di conoscere schemi complessi (es. prefissi EXIF o Dublin Core) è stato demandato al backend, azzerando il carico cognitivo per l'utente finale.
3. La rigidità formale di SPARQL è stata assorbita dall'LLM, che agisce come mediatore invisibile.

Il capitolo 4 approfondirà il dietro le quinte tecnologico: la struttura del backend, l'ottimizzazione del proxy Vite e il delicato processo di fine-tuning del modello linguistico.

Capitolo 4

Architettura e dettagli implementativi

Questo capitolo illustra l'architettura tecnica e le scelte implementative del motore di Quenya. Mentre il capitolo 3 descriveva il sistema dal punto di vista dell'utente finale, qui analizziamo i componenti software, lo stack tecnologico e le metodologie di addestramento alla base dell'applicativo.

4.1 Stack tecnologico: Fedora e Blazegraph

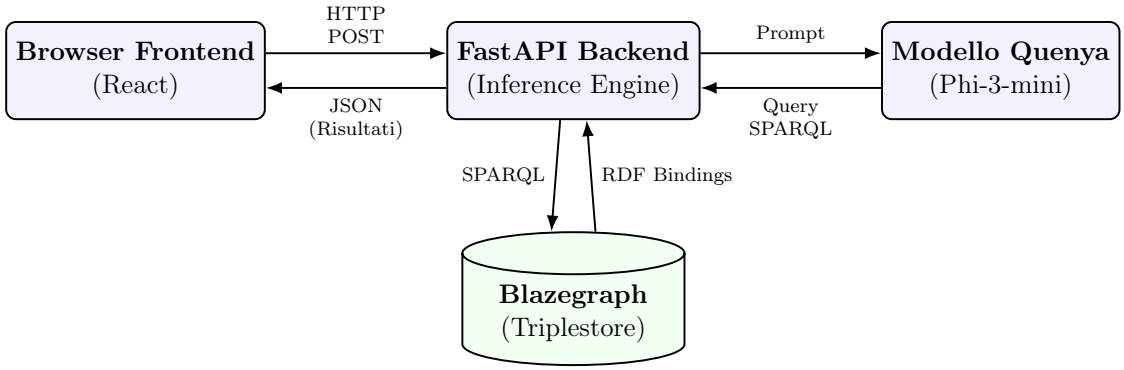


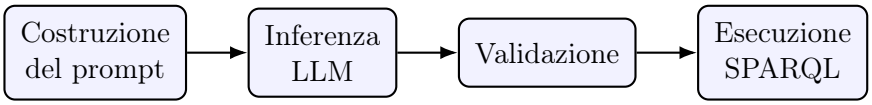
Figura 4.1: Architettura logica e flusso di comunicazione del sistema.

L'infrastruttura dati su cui si appoggia Quenya è costituita da Fedora e Blazegraph [SYS24]. Blazegraph opera come triplestore per il Datavault e fornisce l'endpoint SPARQL interrogato dal sistema. La modellazione architetturale del Datavault e le relative pipeline semantiche (Semantic ETL) per l'estrazione e l'allineamento dei metadati sono state ampiamente documentate e realizzate nel lavoro precedente di Farokhi Darani[FD25]. Il nostro lavoro è consistito nel creare un layer intermedio per dialogare con Blazegraph, astraendo la sintassi SPARQL dall'interfaccia utente.

4.1.1 I tre layer del sistema

A livello logico, il sistema suddivide il flusso di lavoro in tre layer:

- **Presentazione (Datavault Browser)**: applicazione React con due barre di ricerca testuale, una per una ricerca google-like `<SearchField/>` e una per una ricerca avanzata con il tool `<QuenyaSearchBar/>` selezionabili da un bottone centrato sopra di esse. In modalità "Assistente IA", è possibile la visualizzazione della query SPARQL generata per utenti più esperti. Raccoglie l'input testuale dell'utente e lo invia al backend.
- **Applicativo (Quenya Query Builder backend, FastAPI)**: il componente centrale che riceve la domanda naturale, dirige il flusso:

  e gestisce gli errori (query malformate, allucinazioni, timeout, richieste out-of-domain) e restituisce risultati strutturati al frontend.
- **Data Layer (Datavault / Blazegraph)**: immagazzina i milioni di triple RDF del Datavault, esegue la query SPARQL ricevuta dal backend e ne restituisce i risultati grezzi.

4.1.2 Il flusso end-to-end di una richiesta

Una richiesta tipica attraversa il sistema secondo i seguenti passi:

1. L'utente digita una domanda in linguaggio naturale (es. "Mostrami le immagini scansionate con una Nikon nel 2022") nella barra di ricerca del frontend.
2. Il frontend invia un `HTTP POST` al backend con il testo della domanda.
3. Il backend costruisce il prompt completo (system prompt con le regole di generazione + domanda dell'utente, si veda paragrafo 4.4.1) e lo sottopone a Quenya.
4. Il modello genera una query SPARQL (o un JSON di errore, se la richiesta è fuori dominio).
5. Il backend ripulisce e valida l'output (si veda il "sanitizzatore", paragrafo 4.3.1).
6. Se valida, la query viene eseguita su Blazegraph.
7. I risultati vengono trasformati in struttura ad albero e restituiti al frontend, che li visualizza in forma navigabile.

4.1.3 Scelte tecnologiche principali

Il sistema combina le seguenti tecnologie, ciascuna scelta per ragioni specifiche legate ai vincoli del progetto:

- **React**[Met13] (frontend): per la parte di presentazione, già descritta funzionalmente nel capitolo 3.
- **FastAPI**[Ram18] (backend): scelto per la type safety offerta da Pydantic, la documentazione automatica, le performance asincrone e la facilità di integrazione diretta con PyTorch/Transformers per servire il modello.
- **Phi-3-mini**[Mic24] (modello base): dimensione contenuta (3.8 miliardi di parametri) che rende fattibile il fine-tuning mantenendo buone performance di base su compiti di generazione strutturata.
- **Unsloth + LoRA** (fine-tuning): Unsloth[Uns24] ottimizza il fine-tuning e riduce l'impronta in memoria; LoRA (Low-Rank Adaptation) permette di adattare il modello aggiornando solo un sottoinsieme di parametri, abbattendo i costi computazionali.
- **Blazegraph**[SYS24] (triplestore): tecnologia già in uso presso DH.ARC per il Datavault, con supporto nativo a SPARQL 1.1[W3C13] e scalabilità fino a centinaia di milioni di triple.

4.2 Pipeline di fine-tuning dell'LLM

L'utilizzo di un LLM generico si è rivelato insufficiente per ottenere traduzioni affidabili dal linguaggio naturale a SPARQL. Un modello addestrato su testo web conosce la sintassi SPARQL, ma non possiede la mappa mentale dello specifico schema del Datavault, rischiando di allucinare predicati inesistenti o ignorare le particolarità dell'ontologia. Abbiamo perciò introdotto una fase di *domain adaptation* tramite fine-tuning.

4.2.1 Dataset generativo e schema SPARQL

Il dataset di addestramento è stato prodotto in modo sintetico tramite lo script `genera_dataset.py`, usando un approccio a **template parametrici**. Per ciascuna categoria di query è stata definita una funzione Python che, ricevuti parametri come l'anno o il formato file, restituisce la query SPARQL popolata. La corrispondente domanda in linguaggio naturale viene generata variando lessico e struttura sintattica, esponendo così il modello a una pluralità di formulazioni per il medesimo intento.

Il ciclo di generazione

Il cuore del generatore è il ciclo di costruzione del dataset, riportato (in forma semplificata) di seguito:

```
def crea_dataset_massivo(target: int = 10000):
  # ... inizializzazione contatori ...
  tipi_sparql = list(range(1, 14))

  while conteggio < target:
    # Iniezione controllata di esempi Fuori Dominio
    forza_ood = (ood_count < ood_target) and (
      sparql_count >= sparql_target or random.random() < 0.15
    )
    if forza_ood:
      frase = random.choice(out_of_domain)
      query = '{"error": "OUT_OF_DOMAIN"}'
      ood_count += 1
    else:
      # Estrazione di una categoria e popolamento del template
      tipo = random.choice(tipi_sparql)
      if tipo == 1:
        ...  # popolamento template categoria 1
      elif tipo == 2:
        ...  # popolamento template categoria 2
      # [...]
      else:  # tipo == 13
        ...  # popolamento template categoria 13

    # Salvataggio in formato JSONL pronto per il fine-tuning,
    # con lo stesso system prompt usato in inferenza
    riga_dataset = {
      "messages": [
        {"role": "system",  "content": SYSTEM_PROMPT},
        {"role": "user",    "content": frase},
        {"role": "assistant", "content": query},
      ]
    }
```

Code 4.1: Struttura del ciclo di generazione del dataset

Questa architettura garantisce una distribuzione controllata tra le categorie di query, prevenendo sbilanciamenti verso quelle più semplici. Assicura inoltre un'iniezione costante di esempi out-of-domain (circa il 10% del totale), essenziale affinché il modello impari a riconoscere richieste estranee al Datavault. L'approccio generativo ha anche permesso di iterare rapidamente sugli esempi, correggendo a monte gli errori di formattazione o di mapping ontologico emersi nelle prime fasi di test.

Le 13 categorie di query SPARQL (+1 Out-of-Domain)

La versione finale del generatore comprende 13 funzioni template, ciascuna corrispondente a un intento di ricerca distinto. Le funzioni sono state progettate per coprire lo spettro delle interrogazioni tipiche del dominio umanistico, più la gestione separata delle query fuori dominio:

#	**Funzione**	**Categoria**
1	`query_autore_anno`	Ricerca combinata: autore dello scatto + anno
2	`query_dispositivo`	Ricerca per dispositivo di acquisizione (`exif:make`)
3	`query_anno`	Ricerca generica per anno
4	`GOLDEN_QUERY_LIBERA`	Richiesta dell'intero archivio ("mostrami tutto")
5	`query_container`	Ricerca in una cartella/container (`ldp:contains+`)
6	`query_testo`	Ricerca full-text su nome file (`ebucore:filename`)
7	`query_formato`	Filtro per formato/MIME type (es. `image/jpeg`)
8	`query_data_esatta`	Ricerca per data precisa (anno, mese, giorno)
9	`query_intervallo_anni`	Ricerca per intervallo temporale (range di anni)
10	`query_not_anno`	Negazione: esclusione di un anno specifico
11	`query_or_formati`	OR logico tra due formati file
12	`query_limite_custom`	Paginazione: restituzione degli ultimi N file
13	`query_uploader`	Ricerca per autore del caricamento (`fedora:createdBy`)
—	*(separato)*	Out-of-Domain: richieste estranee al dominio (∼10% del dataset)

Tabella 4.1: Le 13 categorie di query SPARQL generate da `genera_dataset.py`, più la categoria Out-of-Domain gestita a parte nel ciclo di generazione.

Si noti come le categorie 1 e 13 condividano lo stesso intento superficiale ("ricerca per autore") ma si distinguano per il predicato RDF target (`exif:artist` per l'autore dello scatto, `fedora:createdBy` per chi ha effettuato il caricamento): questa distinzione, esplicitata anche nella Regola 2 del system prompt (paragrafo 4.4.1), rappresenta uno degli edge case linguistici più rilevanti affrontati nella progettazione del dataset.

Edge case e bug risolti durante la generazione

Nelle prime fasi di validazione avevamo registrato un tasso di errore significativo dovuto a difetti di sintassi SPARQL, in primis un'errata gestione dei prefissi ontologici. Anziché implementare controlli a valle, abbiamo raffinato la sorgente dei dati correggendo i template del training set. Nello specifico, abbiamo risolto i seguenti casi limite:

- **Type mismatch su date**: il formato del filtro temporale era inizialmente incoerente con `xsd:dateTime` — risolto normalizzando il formato nei template e, come descritto nella Regola 6 del system prompt, imponendo il confronto tramite `STR ()` anziché tramite tipi di dato tipizzati.
- **Case-sensitivity nei filtri di negazione** — risolto normalizzando i predicati nel template generatore.
- **Generazione di date calendarialmente non valide** (es. 30 febbraio) — risolto validando le date generate con la libreria Python `datetime`.
- **Ambiguità del separatore nelle date EXIF** (`:` vs `-`) — risolto unificando il formato nel dataset.

Schema RDF reale e un bug di produzione nel namespace EXIF

Predicato	Frequenza	Significato
`rdf:type`	2.707.278	Tipizzazione della risorsa
`ldp:contains`	676.186	Rel. di contenimento container → risorsa
`ebucore:filename`	674.985	Nome del file
`ebucore:hasMimeType`	674.985	Tipo MIME della risorsa
`premis:hasSize`	674.985	Dimensione del file
`exif:dateTime`	674.959	Data/ora di acquisizione
`exif:make / exif:model`	674.959	Dispositivo di acquisizione
`exif:imageWidth`	672.607	Dimensioni in pixel
`dc:title`	1.196	Titolo (solo sui container)

Tabella 4.2: Predicati principali dello schema RDF del Datavault, con frequenza osservata

Durante l'analisi della distribuzione dei predicati nel triplestore di produzione è emerso che il namespace EXIF effettivamente presente in Blazegraph è malformato: `http://wwww3org/...`, anziché lo standard `http://www.w3.org/...` (mancano i due punti e il prefisso `www.` corretto). Il bug è stato confermato eseguendo una query SPARQL di verifica diretta sull'endpoint:

```
SELECT DISTINCT ?p WHERE {
  ?s ?p ?o .
  FILTER(CONTAINS(STR(?p), "exif"))
} LIMIT 5
```

Code 4.2: Query di verifica del namespace EXIF reale

Impatto del bug. Questo mismatch invalida a cascata le interrogazioni: qualunque query SPARQL basata sul namespace EXIF *canonico* restituisce zero risultati per i predicati EXIF, a prescindere dalla sua validità logica. Poiché tali predicati coprono gran parte delle risorse del Datavault, il difetto comprometteva la maggior parte dei casi d'uso del sistema.

Soluzione adottata: allineamento al dato reale, non correzione. A differenza di un approccio "correttivo" (normalizzare il namespace malformato a quello standard prima dell'esecuzione della query), la soluzione adottata è stata l'**allineamento sistematico al namespace reale**, poiché il typo è presente fisicamente nei dati e non è modificabile lato Datavault. Qualunque normalizzazione verso il namespace standard avrebbe prodotto query sintatticamente corrette ma semanticamente vuote. La soluzione è stata applicata in due punti della pipeline:

1. **Generazione del dataset di training** (`genera_dataset.py`): il prefisso malformato è stato hardcoded esplicitamente tra i prefissi fissi usati per generare gli esempi di training, con un commento esplicito a documentazione della scelta:

```
# Il prefisso exif MANTIENE il typo "wwww3org" per
   matchare il tuo DB reale
PREFISSI = (
  "PREFIX exif: <http://wwww3org/2003/12/exif/ns#>\n"
  # ...
)
```

Code 4.3: Prefissi hardcoded nel generatore del dataset

   In questo modo il modello, durante il fine-tuning, apprende direttamente a generare query con il namespace realmente presente in produzione.

2. **Backend di inferenza** (`main.py`): per garantire robustezza anche nel caso in cui il modello generi (o allucini) un prefisso diverso da quello atteso, il backend rimuove qualunque blocco di prefissi generato dall'LLM e re-inietta forzatamente un blocco di prefissi hardcoded, contenente anch'esso il namespace malformato (si veda paragrafo 4.3.1 per il dettaglio del sanitizzatore).

Considerazioni. Questa soluzione, per quanto pragmatica, espone una fragilità intrinseca dell'approccio: il sistema è ora *accoppiato* a un errore presente nei dati di produzione. Se in futuro il Datavault venisse corretto o migrato verso un namespace EXIF standard, sia il dataset di training sia il backend richiederebbero un aggiornamento sincronizzato per evitare di reintrodurre lo stesso problema in forma invertita. Questo aspetto viene ripreso come punto di attenzione nel capitolo 6 (sviluppi futuri) e il suo impatto sulle metriche di valutazione è discusso nel capitolo 5.

4.2.2 Configurazione dell'addestramento con Unsloth

La pipeline di addestramento è stata implementata tramite la libreria Unsloth, per contenere i tempi di esecuzione e il consumo di VRAM, avvalendosi dell'approccio LoRA (Low-Rank Adaptation).

L'algoritmo LoRA

LoRA congela i pesi originali del modello e affianca alle matrici bersaglio W un aggiornamento di basso rango: $W' = W + \frac{\alpha}{r}BA$ (dove $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$, e $r \ll \min(d, k)$). Questo approccio rende il training eseguibile anche su hardware consumer, limitando l'aggiornamento alle sole matrici A e B.
Nel nostro setup abbiamo impostato $r = 16 = \alpha$ e applicato l'aggiornamento sia ai proiettori di attention (`q/k/v/o_proj`) sia a quelli feed-forward (`gate/up/down_proj`). Rispetto all'applicazione standard (limitata all'attention), questa copertura più ampia facilita l'apprendimento del vocabolario di dominio, pur mantenendo contenuto il numero di parametri addestrabili.

Iperparametri utilizzati

Parametro	**Valore**
Modello base	`unsloth/Phi-3-mini-4k-instruct`
Configurazione LoRA	
LoRA rank (r)	16
LoRA alpha	16
LoRA dropout	0
Target modules	`q_proj, k_proj, v_proj, o_proj, gate_proj, up_proj, down_proj`
Bias	`none`
Gradient checkpointing	`True`
Random state	3407

Tabella 4.3: Iperparametri di fine-tuning LoRA (1/2): modello base e configurazione dell'adapter LoRA, configurato tramite Unsloth.

Parametro	Valore
Batch size (per device)	1
Gradient accumulation steps	8
Batch size effettivo	8 (1 × 8)
Warmup steps	10
Epochs	1
Learning rate	`2e-4`
Optimizer	`adamw_8bit`
Weight decay	0.01
LR scheduler	`linear`
Seed	3407
fp16 / bf16	`False` / `False`
Packing	`False`
Dataset num proc	2
Max sequence length	1024 token
Step totali eseguiti	1188
Loss finale	0.010

Tabella 4.4: Iperparametri di fine-tuning LoRA (2/2): configurazione del training tramite `SFTTrainer` (libreria TRL).

Il valore di `max_seq_length` è stato fissato a 1024 token: sufficiente a contenere il system prompt (regole di generazione), la domanda dell'utente in linguaggio naturale, e la query SPARQL attesa, mantenendo comunque un footprint di memoria contenuto durante il training su GPU con VRAM limitata.

Il codice di configurazione effettivamente utilizzato è riportato di seguito. Il primo passo è il caricamento del modello base tramite Unsloth:

```
import torch
from unsloth import FastLanguageModel

max_seq_length = 1024
dtype = torch.float16
load_in_4bit = False

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "unsloth/Phi-3-mini-4k-instruct",
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit,
)
```

Code 4.4: Caricamento del modello base

Da notare che il modello viene caricato in precisione `float16` e senza quantizzazione a 4 bit: questo è distinto dai flag `fp16` / `bf16` del `SFTTrainer` (entrambi `False`), che controllano invece l'uso di *mixed precision training* (autocast durante il calcolo del gradiente) e non la precisione nativa dei pesi del modello.

Segue la configurazione dell'adapter LoRA e del training vero e proprio:

```
model = FastLanguageModel.get_peft_model(
    model,
    r = 16,
    target_modules = ["q_proj", "k_proj", "v_proj", "o_proj",
                      "gate_proj", "up_proj", "down_proj"],
    lora_alpha = 16, lora_dropout = 0,
    bias = "none",   use_gradient_checkpointing = True,
    random_state = 3407,
)
```

Code 4.5: Configurazione dell'adapter LoRA tramite Unsloth

```
from trl import SFTTrainer, SFTConfig
trainer = SFTTrainer(
    model = model, tokenizer = tokenizer,
    train_dataset = train_dataset,
    eval_dataset = eval_dataset,
    args = SFTConfig(
        per_device_train_batch_size = 1,
        gradient_accumulation_steps = 8,
        warmup_steps = 10,
        num_train_epochs = 1,
        learning_rate = 2e-4,
        fp16 = False, bf16 = False,
        logging_steps = 10,
        eval_strategy = "steps",
        eval_steps = 50,
        save_strategy = "steps",
        save_steps = 100,
        save_total_limit = 2,
        report_to = "none",
        optim = "adamw_8bit",
        weight_decay = 0.01,
        lr_scheduler_type = "linear",
        seed = 3407,
        output_dir = "outputs",
        dataset_text_field = "text",
        max_seq_length = max_seq_length,
        dataset_num_proc = 2,
        packing = False,
    ),
)
```

Code 4.6: Configurazione del training tramite SFTTrainer (libreria TRL)

4.3 Architettura del backend e integrazione frontend

4.3.1 Inference engine (FastAPI)

Il coordinamento delle richieste è affidato a un backend in Python basato su FastAPI. L'engine inietta in modo invisibile il *system prompt* in ogni chiamata, costringendo il modello al ruolo di traduttore tecnico. Abbiamo dedicato particolare cura alla gestione del token EOS (*End Of Sentence*), essenziale per troncare l'output e prevenire accodamenti di testo generato dopo la query SPARQL.

Caricamento del modello e fusione dell'adapter LoRA

Nel backend di inferenza, il modello viene istanziato tramite le librerie standard Hugging Face Transformers e PEFT. Il caricamento avviene esclusivamente all'avvio del server, ammortizzando così il costo computazionale della lettura in memoria e della fusione dei pesi:

```
@app.on_event("startup")
async def load_model():
  global model, tokenizer
  print("Inizializzazione del motore Quenya...")

  tokenizer = AutoTokenizer.from_pretrained(ADAPTER_PATH)

  # Usiamo la versione base per evitare conflitti di
   vocabolario
  base_model = AutoModelForCausalLM.from_pretrained(
    "unsloth/Phi-3-mini-4k-instruct",
    torch_dtype=torch.float32,
    device_map="cpu"
  )

  # Fondiamo i pesi LoRA addestrati nel modello base
  peft_model = PeftModel.from_pretrained(base_model,
   ADAPTER_PATH)
  model = peft_model.merge_and_unload()
  print("Fusione completata! Modello Quenya pronto...")
```

Code 4.7: Caricamento e fusione dell'adapter LoRA all'avvio del server FastAPI

Commentiamo due scelte architetturali degne di nota: `merge_and_unload()` fonde *fisicamente* i pesi dell'adapter LoRA nel modello base, anziché mantenerli separati durante l'inferenza, eliminando l'overhead computazionale di applicare l'aggiornamento di basso rango a ogni forward pass; e `device_map=‘‘cpu’’` con `torch_dtype=torch.float32` rispecchiano un deployment senza GPU dedicate, dove la piena precisione a 32 bit evita instabilità numeriche nella fusione dei pesi (lo stesso tipo di instabilità osservato con precisioni ridotte durante il training).

La route di generazione

Ad ogni richiesta dell'utente, il backend espone l'endpoint `POST /generate`, che riceve la domanda in linguaggio naturale e restituisce la query SPARQL (o l'errore Out-of-Domain) generata dal modello:

```python
@app.post("/generate")
async def generate_sparql(request: ChatRequest):
  if not model or not tokenizer:
    raise HTTPException(status_code=503, detail="Modello non ancora caricato")
  try:
    # Costruzione dei messaggi basata sul System Prompt identico al training
    messages = [
      {"role": "system", "content": SYSTEM_PROMPT_TEXT},
      {"role": "user", "content": request.message}
    ]

    # Uso del chat template nativo del modello
    prompt = tokenizer.apply_chat_template(messages, tokenize=False, add_generation_prompt=True)
    inputs = tokenizer(prompt, return_tensors="pt")

    model.eval()
    end_token_id = tokenizer.convert_tokens_to_ids("<|end|>")
    terminators = [tokenizer.eos_token_id, end_token_id]

    # Inferenza pura senza gradienti
    with torch.no_grad():
      outputs = model.generate(
        **inputs,
        max_new_tokens=250,
        do_sample=False,
        pad_token_id=tokenizer.eos_token_id,
        eos_token_id=terminators
      )

    # Estrazione della risposta rimuovendo i token iniziali
    input_length = inputs['input_ids'].shape[1]
    generated_tokens = outputs[0][input_length:]
    assistant_reply = tokenizer.decode(generated_tokens, skip_special_tokens=True).strip()

    # Segue il sanitizzatore
    return {"response": sanitized_query}
```

Code 4.8: Route di generazione della query SPARQL

Si notino tre scelte implementative in questo blocco:

- **Doppio token di stop**: Phi-3 impiega il token `<|end|>` per chiudere il turno di conversazione, oltre all'EOS standard. Omettere `<|end|>` nei `terminators` causava cicli di generazione inarrestabili.
- **Generazione deterministica**: `do_sample=False` disattiva il campionamento stocastico. Nel contesto della traduzione verso SPARQL, dove si insegue l'esattezza sintattica piuttosto che la varietà lessicale, l'output deterministico è strettamente necessario.
- **Limite ai token generati**: `max_new_tokens=250` fornisce sufficiente capienza per query vincolate al template, fungendo anche da rete di salvataggio in caso di fallimento dei token di stop.

Il sanitizzatore: parsing e validazione dell'output

Il testo grezzo generato dal modello (`assistant_reply`) non viene restituito direttamente al frontend, ma passa attraverso una fase di post-processing che nel codice del backend è internamente chiamata "sanitizzatore". Le responsabilità di questo passaggio includono:

- **Pulizia dei prefissi**: rimozione di eventuali blocchi `PREFIX` generati (o allucinati) dal modello, sostituiti dal blocco di prefissi hardcoded e correttamente allineati al namespace reale del Datavault.
- **Fallback Out-of-Domain**: riconoscimento e gestione del caso in cui il modello restituisca il JSON `{''error'': ''OUT_OF_DOMAIN''}` , da intercettare prima di tentare un'esecuzione SPARQL che fallirebbe.
- **Fix del formato data**: normalizzazione del formato e del cast `STR()` nei filtri sulle date, per garantire coerenza con la Regola 6 del system prompt anche in caso di lievi variazioni nell'output del modello.

Il codice qui di seguito illustra l'implementazione in Python di questo modulo, evidenziando l'uso di espressioni regolari (regex) per estrarre in sicurezza il corpo della query.

```python
import re
import json

# Prefissi hardcoded
HARDCODED_PREFIXES = """
  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  PREFIX ldp: <http://www.w3.org/ns/ldp#>
  PREFIX dc: <http://purl.org/dc/elements/1.1/>
  PREFIX fedora: <http://fedora.info/definitions/v4/repository#>
  PREFIX ebucore: <http://www.ebu.ch/metadata/ontologies/ebucore/ebucore#>
  PREFIX exif: <http://wwww3org/2003/12/exif/ns#>
"""

def sanitize_sparql_query(raw_output: str) -> str:
  # 1. Gestione rapida delle richieste Fuori Dominio
  if "OUT_OF_DOMAIN" in raw_output:
    return '{"error": "OUT_OF_DOMAIN"}'
  # 2. Isolamento del corpo della query
  # Cerca la clausola SELECT e cattura tutto il testo
  match_select = re.search(r"(SELECT\s+DISTINCT.*)", raw_output,
                           flags=re.IGNORECASE | re.DOTALL)

  if not match_select:
    # Fallback di sicurezza: se non trova la SELECT, ritorna l'output grezzo
    return raw_output

  core_query = match_select.group(1).strip()
  # 3. Fix difensivo sul formato date (Regola 6)
  # Assicura che la variabile della data sia castata a stringa dentro STRSTARTS
  core_query = re.sub(
    r'STRSTARTS\s*\(\s*\?([a-zA-Z0-9_]+)\s*,',
    r'STRSTARTS(STR(?\1),',
    core_query
  )
  # 4. Assemblaggio finale
  sanitized_query = f"{HARDCODED_PREFIXES}\n{core_query}"

  return sanitized_query
```

Code 4.9: Implementazione del sanitizzatore nel backend FastAPI.

Il sanitizzatore applica tre filtri di sicurezza a cascata:

- **Short-circuit evaluation (Out-of-Domain):**
  Se il testo contiene `OUT_OF_DOMAIN`, la funzione si arresta e restituisce il JSON di errore. Ciò impedisce che il backend tenti di parsare l'errore come query SPARQL.
- **Troncamento (`re.search`):**
  Anche se addestrato a rispondere solo con la `SELECT`, occasionalmente il modello anticipava la query con blocchi `PREFIX`. La regex estrae solo dal pattern `SELECT DISTINCT` in poi, demandando al backend l'iniezione dei namespace corretti.
- **Normalizzazione semantica (`re.sub`):**
  In rari casi il modello ometteva la conversione esplicita `STR(?date)` richiesta dalla Regola 6. Avendo Blazegraph bisogno di confronti testuali puri, l'omissione produceva query valide ma senza risultati. La regex rileva la dimenticanza e ripristina il cast `STR()` dove necessario.

4.4 Formato dati: dataset_datavault.jsonl

Il formato dati alla base del fine-tuning struttura la conoscenza del dominio in formato JSONL: ogni riga è un oggetto JSON contenente una tripla di messaggi (`system`, `user`, `assistant`), pronta per l'addestramento supervisionato tramite `SFTTrainer`.

4.4.1 Le 8 regole di generazione SPARQL

Per istruire l'LLM sono state definite 8 regole di generazione SPARQL, sotto forma di system prompt. Una caratteristica progettuale importante, nonché garanzia di coerenza tra training e inferenza, è che questo system prompt è identico sia in `genera_dataset.py` (training) sia in `main.py` (inferenza), eliminando una possibile fonte di *distribution shift* tra le condizioni di addestramento e quelle di utilizzo reale del modello:

```
Sei un traduttore da linguaggio naturale a SPARQL per un Data
   Vault Fedora. REGOLE ASSOLUTE:
1. Restituisci SEMPRE e SOLO: SELECT DISTINCT ?image WHERE {
   ... }. Nessun altra variabile nella SELECT.
2. Autore scatto: usa exif:artist. Autore caricamento: usa
   fedora:createdBy.
3. Dispositivo: usa exif:make.
4. Formato file: usa ebucore:hasMimeType.
5. Ricerca testuale: cerca SOLO dentro ebucore:filename.
   Attenzione: se l'utente cerca parole come "data", "del", "
   vault", trattale come testo, NON come date.
6. Date: usa exif:dateTime. Usa SEMPRE STR() nel filtro, es:
   FILTER(STRSTARTS(STR(?date), "2023:05:10")).
7. Ricerca in cartelle/container: usa ?container ldp:contains+
    ?image e dc:title sul container.
8. FUORI DOMINIO: Se la richiesta e' estranea all'archivio,
   rispondi SOLO con il JSON: {"error": "OUT_OF_DOMAIN"}.
```

Code 4.10: System prompt utilizzato in training e inference (identico in entrambi i file)

Queste regole impongono restrizioni al modello su quali relazioni attraversare, come filtrare le date, e come gestire la paginazione e i limiti dei risultati.

Se le regole devono essere modificate, è necessario aggiornarle in entrambi i punti della codebase: `main.py` per un impatto immediato in inferenza, e `genera_dataset.py` nel caso si voglia generare un nuovo dataset e ri-addestrare il modello, al fine di evitare la reintroduzione di un disallineamento tra training e inferenza.

4.4.2 Gestione degli errori: OUT_OF_DOMAIN

Per garantire l'affidabilità è fondamentale che il sistema sia in grado di ammettere i propri limiti. Per questo motivo, nel dataset è stata implementata un'architettura di *error handling* JSON-based: quando l'utente pone una domanda irrilevante

rispetto allo schema del Datavault, il modello è addestrato (Regola 8, paragrafo 4.4.1) a non tentare di forzare una query SPARQL, ma a restituire un esplicito stato `OUT_OF_DOMAIN`. Questo fallback previene l'esecuzione di query malformate e migliora la trasparenza dell'interfaccia utente.

Per riassumere, la tabella seguente elenca il comportamento del sistema nei principali casi di errore gestiti:

Caso di errore	Comportamento del sistema e strategia di mitigazione
Predicato allucinato dal modello	Prefissi hardcoded re-iniettati dal sanitizzatore (paragrafo 4.3.1); la validazione a valle blocca le query contenenti predicati non riconosciuti.
Query SPARQL malformata	La validazione sintattica blocca l'esecuzione prima dell'invio al triplestore; un errore strutturato viene restituito al frontend.
Richiesta Out-of-Domain	Il modello risponde esclusivamente con il JSON di fallback `{"error": "OUT_OF_DOMAIN"}` (Regola 8).
Generazione ininterrotta (assenza di token EOS)	Adozione di un doppio token di stop (`<|end|>` + EOS standard) abbinato a un limite di sicurezza hardware (`max_new_tokens=250`).
Mismatch del namespace EXIF	Allineamento sistematico al namespace reale presente nei dati di produzione, rinunciando alla correzione canonica a monte.

Tabella 4.5: Riepilogo dei principali casi di errore gestiti dal sistema e relative strategie di mitigazione adottate.

4.5 Conclusioni del capitolo

Lo sviluppo discusso in questo capitolo mostra come la conversione di un'idea teorica in uno strumento utilizzabile dipenda dall'ingegnerizzazione dei componenti. La cura del dataset, la gestione dei namespace e l'architettura dei layer applicativi sono requisiti essenziali, non dettagli.

Il prossimo capitolo metterà alla prova l'intero sistema, valutandone l'efficacia sul campo.

Capitolo 5

Valutazione

Dopo aver descritto la costruzione del sistema nel capitolo 4, in questo capitolo ne valutiamo il comportamento. L'analisi si articola su due piani: l'**efficienza** (paragrafo 5.2), misurata tramite metriche quantitative (tempi di generazione, tassi di esecuzione, convergenza), e l'**efficacia** (paragrafo 5.3), che riflette il salto qualitativo rispetto alle pratiche precedenti (funzionalità, usabilità, integrazione).

5.1 Metodologia generale

La valutazione quantitativa si sviluppa in due fasi, eseguite su macchine distinte per ottimizzare le risorse hardware disponibili:

1. **Fase 1 — Generazione delle predizioni**
   (`generate_predictions.py`): per ciascuna domanda di un test set di 500 esempi, il modello fine-tuned genera la query SPARQL corrispondente, registrando anche il tempo di generazione per esempio.
2. **Fase 2 — Valutazione contro Blazegraph**
   (`evaluate_against_blazegraph.py`): sia la query generata sia quella di riferimento vengono eseguite contro l'endpoint Blazegraph reale del Datavault, e i rispettivi insiemi di risultati vengono confrontati per calcolare precision, recall e F1.

Abbiamo dovuto dividere la pipeline in due fasi perché la fase di generazione richiede l'accelerazione GPU (non disponibile localmente), mentre la fase di valutazione richiede solo chiamate I/O verso l'endpoint Blazegraph e può essere eseguita comodamente su CPU.

Nota metodologica: confronto sull'insieme completo dei risultati

Una scelta metodologica rilevante riguarda il trattamento delle clausole `ORDER BY` e `LIMIT`. SPARQL non garantisce un ordine deterministico dei risultati quando è presente un `LIMIT` senza un corrispondente `ORDER BY`: eseguire due volte la stessa query può restituire campioni diversi delle righe disponibili, producendo falsi negativi anche quando la query generata è testualmente identica a quella di riferimento.

Per questo motivo, precision e recall vengono calcolate sull'insieme *completo* dei risultati, dopo aver rimosso `ORDER BY` e `LIMIT` da entrambe le query prima dell'esecuzione. L'obiettivo è isolare la correttezza della *logica di filtro* (la clausola `WHERE`), cioè se il modello ha capito *quali* elementi cercare, da un controllo distinto sulla clausola `LIMIT`: registrato nella colonna `limit_match` del dataset di valutazione, verifica se il modello ha capito anche *quanti* elementi erano richiesti.

5.2 Efficienza: valutazione quantitativa

5.2.1 Struttura dei test

I parametri di esecuzione del test sono i seguenti:

- **Macchina (generazione)**: GPU NVIDIA T4 (su Kaggle), modello Phi-3-mini fine-tuned con adapter LoRA fuso (`merge_and_unload`).
- **Macchina (valutazione)**: CPU locale. Nessun modello caricato; esegue unicamente chiamate HTTP verso l'endpoint Blazegraph via proxy Vite.
- **Task**: Traduzione di una domanda in italiano in una query SPARQL, confrontata contro il riferimento generato dal sistema a template (paragrafo 4.2.1).
- **Dati (Test Set)**: 500 esempi. Costituisce una reale partizione *held-out* (estratta al 5% con seed deterministico `3407`), priva di sovrapposizioni col training.

5.2.2 Distribuzione del test set per categoria

Il test set estratto copre 7 delle 14 categorie di query definite nel generatore (paragrafo 4.2.1). Come visibile in tabella 5.1 e figura 5.1, la distribuzione non è uniforme: la categoria `data_esatta` da sola costituisce quasi la metà del campione (233/500, 46.6%), mentre alcune categorie di nicchia (es. `query_not_anno`) non risultano rappresentate.

Questa distribuzione irregolare influenza l'interpretazione dei risultati: per questo motivo, nel paragrafo 5.3.1 le metriche verranno analizzate per singola categoria piuttosto che come media aggregata.

Categoria	N. esempi
`data_esatta`	233
`formato_file`	66
`fuori_dominio`	57
`dispositivo`	40
`container_cartella`	30
`ricerca_testuale`	28
`autore`	26
`data_intervallo`	20
Totale	500

Tabella 5.1: Distribuzione del test set per categoria euristica.

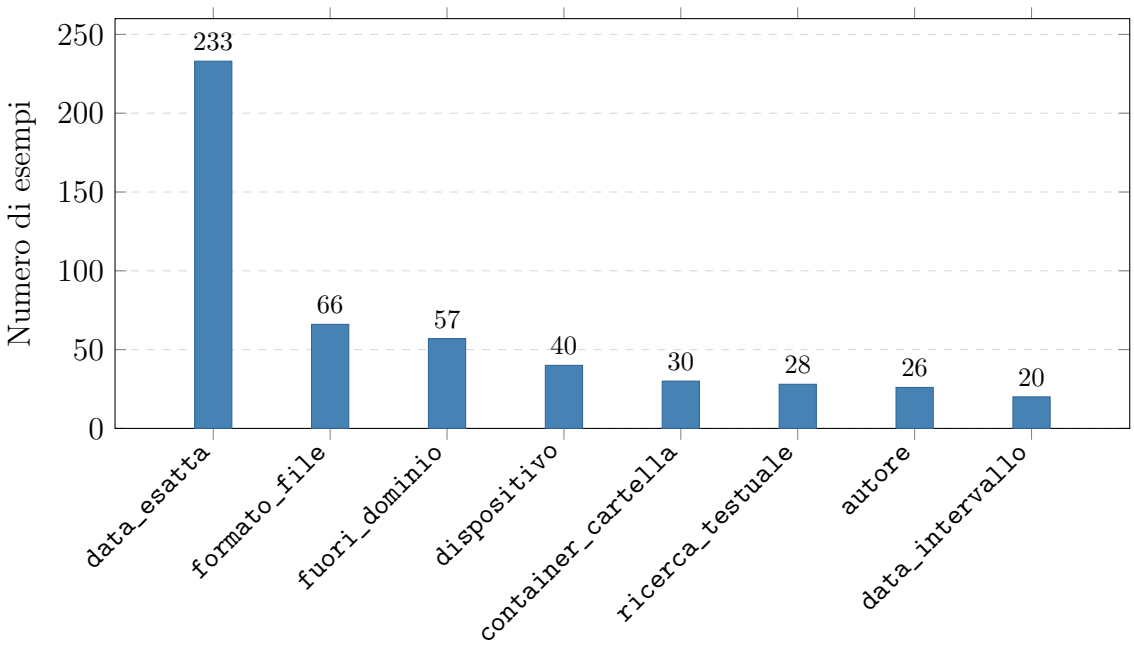


Figura 5.1: Distribuzione del test set di valutazione ($n = 500$).

5.2.3 Tempo di generazione

Il tempo di generazione per query, misurato sulla GPU T4 durante la Fase 1, presenta le seguenti statistiche calcolate sull'intero set di 500 esempi:

Statistica	Valore (s)
Media	7.45
Deviazione standard	2.51
Minimo	0.65
25° percentile	7.64
Mediana	8.19
75° percentile	8.59
Massimo	10.45

Tabella 5.2: Statistiche del tempo di generazione per query (GPU T4).

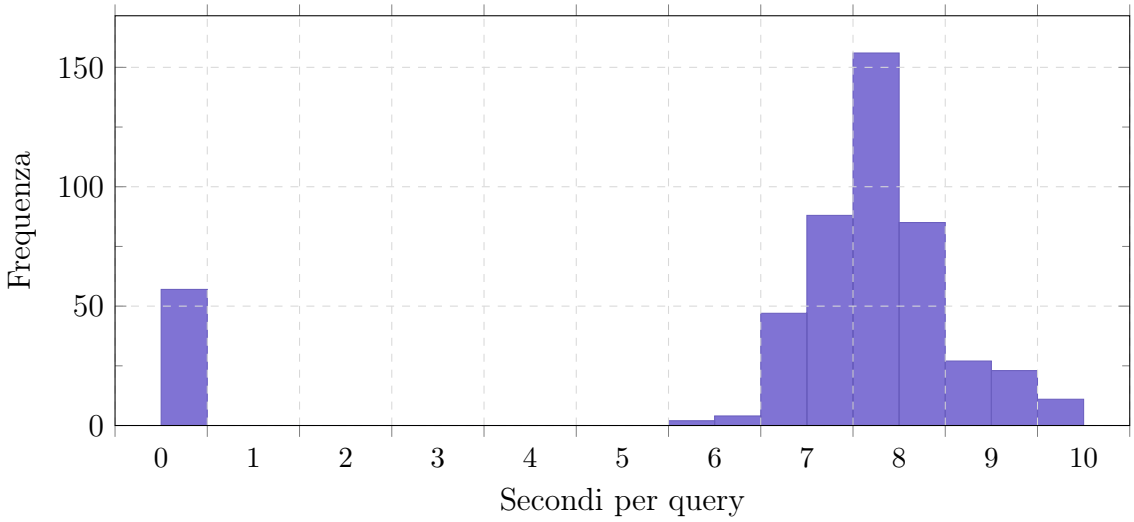


Figura 5.2: Distribuzione dei tempi di generazione su GPU T4.

La distribuzione è concentrata tra i 7 e i 9 secondi per la maggioranza delle query, con una coda di valori bassi (minimo 0.65s) verosimilmente legata a generazioni più brevi, come le risposte *Out-of-Domain*, strutturalmente più corte di una query SPARQL completa.

5.2.4 Tasso di esecuzione corretta

Su tutti i 443 esempi non Out-of-Domain del test set, il **100% delle query generate è stato eseguito con successo** contro l'endpoint Blazegraph reale, senza errori di sintassi o di esecuzione (`execution_ok = True` per tutte le righe, nessun `exec_error` registrato). Questo dato conferma la validità del system prompt (paragrafo 4.4.1): il modello rispetta rigidamente le regole sintattiche, generando esclusivamente query ben formate.

Nessun impatto misurato del bug del namespace EXIF

Il capitolo 4 aveva anticipato che l'impatto del namespace EXIF malformato (paragrafo 4.2.1) sulle metriche di valutazione sarebbe stato verificato in questo capitolo. Il dato è chiaro: **nessun impatto negativo è stato osservato**. Su 293 dei 443 esempi non Out-of-Domain (categorie `data_esatta`, `data_intervallo` e `dispositivo`), la query di riferimento utilizza effettivamente un predicato `exif:` nel corpo del `WHERE`, non solo nella dichiarazione dei prefissi; tutti e 293 sono stati eseguiti correttamente, con precision e recall pari a 1.000 (tabella 5.3).

L'esito attesta il successo della strategia di mitigazione introdotta nel capitolo 4. Avendo allineato il dataset e il backend al namespace malformato di Blazegraph, il sistema non cade nell'errore di usare il prefisso standard (`www.w3.org`), che farebbe fallire la query. Se ignorato, il bug avrebbe annullato *precision* e *recall* su quasi due terzi del test set; l'assenza di un calo prestazionale dimostra l'efficacia del workaround, ma il problema originario persiste nei dati di produzione (una criticità architetturale ripresa nel capitolo 6).

5.2.5 Convergenza del training

Il log di training (paragrafo 4.2.2) mostra una rapida convergenza della loss: da un valore di 0.044 allo step 50 fino a stabilizzarsi attorno a 0.010–0.011 a partire circa dallo step 400, mantenendosi sostanzialmente costante fino al termine dell'addestramento (step 1188, loss finale 0.010).

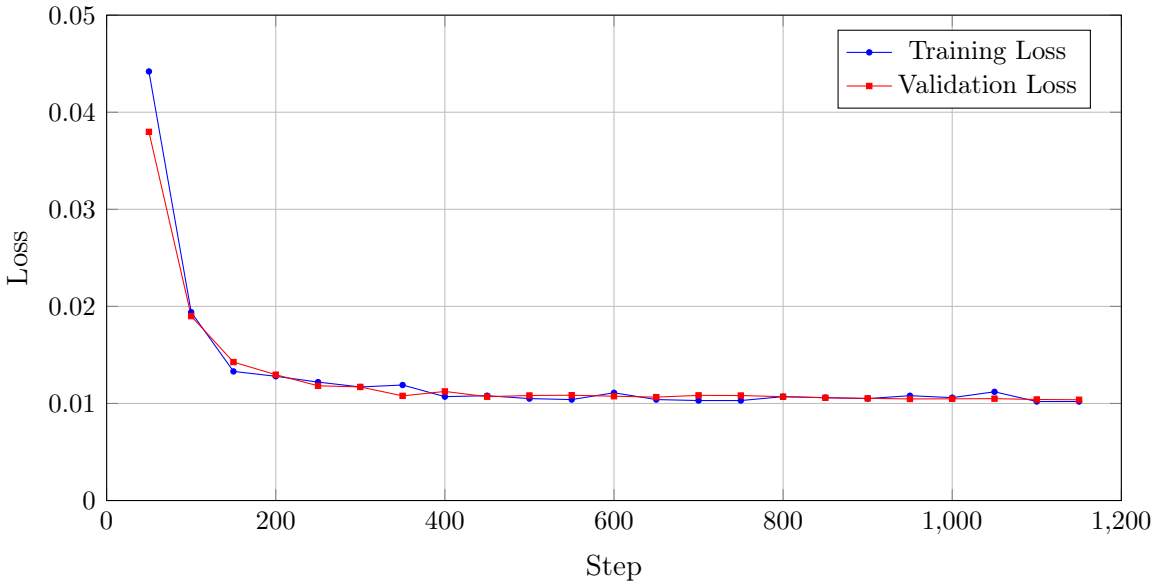


Figura 5.3: Andamento della training loss e della validation loss durante il fine-tuning

La curva di *validation loss* aderisce a quella di training, escludendo fenomeni di overfitting. Entrambe si stabilizzano intorno allo step 400 su 1188, indicando che il modello avrebbe raggiunto la convergenza anche con un addestramento più breve.

5.3 Efficacia: valutazione qualitativa

5.3.1 Metriche di retrieval per categoria

Applicando la metodologia descritta in paragrafo 5.1, ovvero il confronto sull'insieme completo dei risultati, al netto di `ORDER BY` / `LIMIT`, le metriche di precision, recall e F1 per categoria sono le seguenti:

Categoria	N	Precision	Recall	F1
data_esatta	233	1.000	1.000	1.000
formato_file	66	1.000	1.000	1.000
dispositivo	40	1.000	1.000	1.000
container_cartella	30	1.000	1.000	1.000
ricerca_testuale	28	1.000	1.000	1.000
autore	26	1.000	1.000	1.000
data_intervallo	20	1.000	1.000	1.000
Media complessiva	**443**	**1.000**	**1.000**	**1.000**

Tabella 5.3: Precision, recall e F1 medi per categoria, calcolati sull'insieme completo dei risultati (al netto di `ORDER BY` / `LIMIT`)

Anche il riconoscimento delle richieste Out-of-Domain (paragrafo 4.4.2) raggiunge il 100% di accuratezza: tutti i 57 esempi fuori dominio nel test set sono stati correttamente classificati come tali dal modello. Allo stesso modo, il `limit_match` (corrispondenza esatta del valore richiesto nella clausola `LIMIT`) è verificato nel 100% dei 443 casi non Out-of-Domain in cui almeno una delle due query include un `LIMIT`.

5.3.2 Nota critica: interpretazione dei risultati perfetti

I risultati perfetti riportati in tabella 5.3 (punteggio 1.000 su ogni fronte) richiedono un'interpretazione analitica.

Ispezionando il file `predictions.json`, emerge che **le 500 query generate coincidono, carattere per carattere, con le query di riferimento**. Non assistiamo semplicemente a un'equivalenza semantica dei risultati, ma all'esatta riproduzione della stringa SPARQL attesa.

Questo comportamento deriva dalla genesi del test set: essendo prodotto dagli stessi template parametrici (paragrafo 4.2.1) usati per il training, ne eredita la rigidità strutturale. Le regole del system prompt (paragrafo 4.4.1) restringono drasticamente lo spazio delle query valide, imponendo predicati e costrutti fissi. Di conseguenza, a parità di intento informazionale, la traduzione diventa sostanzialmente deterministica.

In altre parole, una corrispondenza esatta al 100% è coerente con due ipotesi non mutuamente esclusive:

1. Il modello ha effettivamente appreso con grande efficacia la mappatura tra intenti linguistici e struttura SPARQL imposta dalle regole, generalizzando correttamente su parametri (anni, dispositivi, nomi di file) anche se non identici a quelli visti in training;
2. Il test set, essendo generato dallo stesso processo del training set, condivide con esso una distribuzione di superficie sufficientemente vincolata (per via delle 8 regole assolute) da rendere il compito di generazione quasi deterministico, riducendo il valore di questa valutazione come misura di *generalizzazione* a domande realmente nuove e formulate liberamente da un utente umano.

Il codice di inferenza (`generate_predictions.ipynb`) conferma che il test set costituisce una reale partizione *held-out*: l'estrazione è avvenuta in modo deterministico con un rapporto 95%/5% e lo stesso seed (`3407`) usato per la configurazione del training. Questo esclude *data leakage* o sovrapposizioni (il modello ha elaborato istanze mai viste in fase di addestramento) e avvalora l'ipotesi della generalizzazione strutturale (punto 1), pur confermando che tale generalizzazione avviene strettamente *in-distribution* rispetto ai template.

Ciò non toglie valore al traguardo raggiunto: l'accuratezza del 100% garantisce un'operatività robusta per l'applicativo. Tuttavia, ne ridimensiona la portata: questa valutazione certifica la disciplina del modello nell'applicare la grammatica del system prompt, ma non la sua capacità di interpretare formulazioni del tutto slegate dai pattern di addestramento. Una prova empirica con ricercatori umani, liberi di porre le proprie domande senza vincoli a priori, rimane un tassello mancante (capitolo 6).

5.3.3 Miglioramento rispetto alle soluzioni precedenti

Oltre alle metriche quantitative, l'efficacia si valuta sull'impatto pratico per i ricercatori, risolvendo le barriere esaminate nel capitolo 2:

- **Funzionalità**: colma l'assenza di un motore di ricerca semantico sul Datavault, affiancando la ricerca per contenuto alla preesistente navigazione per cartelle (paragrafo 3.1.1).
- **Usabilità**: la barra di ricerca testuale astrae completamente i concetti di triplestore, prefissi RDF e sintassi SPARQL, rendendo accessibile il patrimonio informativo a utenti non tecnici (paragrafo 2.1.3).
- **Integrazione**: opera come layer aggiuntivo, inserendosi nello stack Fedora/Blazegraph senza richiedere alterazioni dello schema o migrazioni di dati.
- **Manutenzione**: l'architettura modulare (dataset sintetico, prompt, backend isolato) permette di espandere le categorie di query o aggiornare la logica senza impattare il database sottostante.

Nessun confronto diretto con l'alternativa manuale

L'analisi appena presentata è prettamente architetturale: manca una validazione empirica. Non abbiamo cronometrato la scrittura manuale delle query SPARQL per i casi d'uso del capitolo 3, né abbiamo sottoposto il sistema a un campione di veri utenti DH.ARC.

L'onere sintattico dei casi d'uso (che richiedono l'uso combinato di predicati come `exif:make`, `exif:dateTime` ed `ebucore:hasMimeType` in una singola clausola `WHERE`) suggerisce un netto risparmio di tempo e di frustrazione rispetto all'alternativa manuale, in particolare per un'utenza umanistica. Tale beneficio, per ora presunto e supportato dalla teoria del carico cognitivo, dovrà essere quantificato empiricamente. Questo limite metodologico troverà spazio tra gli sviluppi futuri (capitolo 6).

5.4 Discussione dei risultati e limiti della valutazione

I dati raccolti offrono un quadro chiaro sulle prestazioni:

1. **Solidità sintattica**: il vincolo del system prompt (paragrafo 4.4.1) garantisce un tasso di esecuzione privo di errori contro Blazegraph.
2. **Tempi di risposta adeguati**: l'elaborazione media di 7.45 secondi su GPU T4 risulta compatibile con un uso esplorativo. Le misurazioni dovranno tuttavia essere estese all'effettivo deployment su CPU (paragrafo 5.2.3).
3. **Metriche apparentemente perfette**: precisione, recall e F1 a quota 1.000 misurano l'efficacia del modello nel replicare costrutti generati dai template (paragrafo 5.3.2), senza per questo attestare un'analoga robustezza nell'interpretazione di input utente destrutturati.

La natura sintetica del test set rappresenta dunque la principale vulnerabilità dell'analisi. Per saggiare la reale elasticità dell'assistente occorrerà raccogliere una base dati organica, derivante dall'interazione spontanea dei ricercatori del centro (capitolo 6).

Capitolo 6

Conclusioni e sviluppi futuri

Questo capitolo tira le fila del progetto, tracciando un bilancio: cosa il sistema realizza in modo affidabile, dove mostra il fianco, e perché rappresenti comunque un passo avanti rispetto all'assenza di un motore di ricerca semantico per il Datavault. La sezione finale organizza gli sviluppi futuri per priorità: dai correttivi immediati fino alle esplorazioni più ambiziose.

Analisi critica del sistema (**Risultati ottenuti e punti di forza**):

Affidabilità sintattica

Il dato più solido è il tasso di esecuzione corretta del 100% sul test set held-out (paragrafo 5.2.4): ogni query generata è risultata eseguibile senza errori su Blazegraph. Per un sistema basato su intelligenza artificiale generativa, in cui l'esattezza sintattica non è scontata, l'assenza di errori testimonia che l'abbinamento tra system prompt (paragrafo 4.4.1) e dataset di fine-tuning produce un comportamento prevedibile.

Uno split di valutazione genuinamente held-out

Abbiamo verificato esplicitamente (paragrafo 5.3.2) che il test set non si sovrapponesse ai dati di training: i 500 esempi usati per la validazione non sono mai stati osservati dal modello in fase di addestramento. Pur consapevoli della natura *in-distribution* dei dati valutati, la trasparenza sulla partizione dei dati distingue l'analisi da approcci meno rigorosi.

Economicità delle risorse impiegate

L'intera pipeline di addestramento è stata condotta impiegando hardware accessibile (GPU T4 di Kaggle) e un modello di base leggero (Phi-3-mini, 3.8 miliardi di parametri). Il fine-tuning si è stabilizzato in meno di 1200 step (paragrafo 5.2.5). Il risultato dimostra come, delimitando il perimetro a una sintassi vincolata, un task di traduzione testo-a-codice sia affrontabile senza ricorrere ad architetture dispendiose.

Coerenza architetturale tra training e inferenza

Il system prompt usato per generare il dataset è lo stesso, riga per riga, inserito nel backend di inferenza (paragrafo 4.4.1). Questa identità scongiura una criticità diffusa: il *train-serve skew*, ovvero il disallineamento tra i prompt visti dal modello in addestramento e quelli inviati in produzione.

Limiti del sistema e criticità: Limiti della valutazione rispetto agli scenari d'uso reale

Come discusso nel paragrafo 5.3.2, la valutazione quantitativa fotografa l'accuratezza del modello su formati già visti in training. Non sappiamo se l'assistente si dimostrerà altrettanto abile nell'interpretare domande spontanee da parte di ricercatori estranei alle 13 categorie (paragrafo 4.2.1). Manca all'appello un vero stress test: l'interazione sul campo con utenti DH.ARC.

Copertura parziale delle categorie nel test set

Il generatore produce 13 categorie di query più la gestione Out-of-Domain, ma il test set effettivamente valutato ne copre solo 7 (paragrafo 5.2.2). Categorie come la negazione (`query_not_anno`) o l'OR logico tra formati (`query_or_formati`) non sono rappresentate nel campione analizzato: non sappiamo con la stessa sicurezza se il sistema le gestisca altrettanto bene.

Fragilità del sistema rispetto al bug del namespace EXIF

La soluzione descritta in paragrafo 4.2.1 funziona, ma accoppia il sistema a un errore presente nei dati di produzione. Se il Datavault venisse in futuro corretto o migrato, sia il dataset di training sia il backend richiederebbero un aggiornamento sincronizzato: una dipendenza fragile che un sistema più maturo dovrebbe gestire in modo più robusto, ad esempio con un meccanismo di rilevamento automatico del namespace realmente in uso.

Inferenza su CPU in produzione

Il backend di produzione esegue l'inferenza su CPU (paragrafo 4.3.1), mentre i tempi di generazione misurati (paragrafo 5.2.3) si riferiscono all'esecuzione su GPU T4. Non è stato misurato il tempo di risposta reale che un utente sperimenterebbe nell'uso quotidiano del sistema in produzione, un dato che sarebbe stato preferibile avere.

Vincolo di lunghezza della sequenza

Il limite di 1024 token in `max_seq_length` (paragrafo 4.2.2), scelto per contenere l'uso di memoria durante il training, potrebbe risultare insufficiente per domande particolarmente articolate o per un futuro ampliamento del system prompt (ad esempio se si aggiungessero più predicati o esempi few-shot nel prompt stesso).

Impatto e prospettive nel dominio umanistico

Rispetto allo scenario precedente, in cui l'esplorazione del Datavault richiedeva obbligatoriamente la scrittura manuale di SPARQL (paragrafo 2.1.3), l'interfaccia realizzata trasforma un'operazione preclusa agli umanisti in un'attività fluida. Pur coi suoi limiti, il tool offre una soluzione percorribile a un problema finora irrisolto.

6.1 Sviluppi futuri

Gli sviluppi futuri sono organizzati in ordine di priorità decrescente: dai bug fix più urgenti alle estensioni più ambiziose.

Bug fix

1. **Estendere la valutazione alle categorie mancanti**: eseguire la Fase 1/Fase 2 della pipeline di valutazione (paragrafo 5.1) su un campione che includa esplicitamente le categorie non rappresentate nel test set attuale (negazione, OR tra formati, paginazione custom, ricerca per uploader), per verificare se il tasso di esecuzione corretta del 100% si mantiene anche lì.
2. **Rendere il namespace EXIF configurabile**: estrarre il namespace hardcoded (paragrafo 4.2.1) in un parametro di configurazione centralizzato, così che un'eventuale correzione futura del Datavault richieda una modifica in un solo punto anziché in due file separati.
3. **Misurare i tempi di inferenza reali su CPU**: eseguire un benchmark diretto sul backend di produzione, sostituendo la stima qualitativa attuale (paragrafo 5.2.3) con un dato misurato.

Feature mancanti

1. **Un vero test di usabilità con utenti reali**: come discusso in paragrafo 5.3.3, manca ancora una misura empirica del vantaggio percepito da un ricercatore umanista reale, sia in termini di tempo risparmiato sia di soddisfazione d'uso rispetto all'assenza di alternative.
2. **Supporto multilingue**: il sistema opera attualmente solo in italiano. Un'estensione naturale, vista la composizione internazionale della comunità di ricerca in Digital Humanities, è l'aggiunta di supporto per l'inglese e altre lingue.
3. **Un'interfaccia di correzione della query generata**: per gli utenti più esperti, permettere una modifica diretta della query SPARQL proposta dal sistema (già visualizzabile, si veda paragrafo 4.1.1) prima dell'esecuzione, chiudendo il cerchio tra automazione e controllo manuale.

Estensioni fondamentali

1. **Ampliare la copertura semantica dello schema**: le 13 categorie attuali (paragrafo 4.2.1) coprono i pattern di ricerca più comuni rispetto all'attuale popolazione del Datavault, ma lo schema RDF contiene predicati non ancora sfruttati (ad esempio quelli PREMIS relativi alla dimensione dei file, o le

relazioni RDFS di sottoclasse osservate nello schema). Un'estensione del dataset generativo potrebbe abilitare interrogazioni oggi non supportate.

2. **Passare da un prompt statico a un vero meccanismo RAG dinamico**: il sistema attuale inietta lo schema RDF nel system prompt in forma fissa (paragrafo 4.4.1). Un'evoluzione naturale, discussa a livello teorico nel paragrafo 2.3, è recuperare dinamicamente solo i frammenti di schema rilevanti per ciascuna domanda, permettendo al sistema di scalare a schemi più ampi o mutevoli senza dover ri-addestrare il modello a ogni modifica dell'ontologia.
3. **Inferenza su GPU in produzione**: sostituire l'attuale deployment su CPU (paragrafo 4.3.1) con un'infrastruttura GPU dedicata, per ridurre i tempi di risposta percepiti dall'utente finale a un livello compatibile con un'interazione conversazionale fluida.

Possibili sviluppi futuri

1. **Ricerca per contenuto visivo, non solo per metadati**: l'intero sistema attuale opera sui metadati EXIF, Dublin Core ed EBUCore associati alle immagini, non sul loro contenuto visivo. Un'estensione di grande ambizione sarebbe l'integrazione di modelli di visione (ad esempio embedding CLIP-like) che permettano domande come "mostrami le immagini che raffigurano una miniatura con un drago", andando oltre la ricerca per attributi tecnici o descrittivi verso una vera ricerca per concetto visivo, in piena coerenza con la visione di "findability" discussa nel capitolo 2.
2. **Un sistema federato su più Knowledge Graph**: estendere l'approccio oltre il singolo Datavault di DH.ARC, verso un'interrogazione federata di più triplestore appartenenti a istituzioni diverse del patrimonio culturale, con il sistema in grado di determinare autonomamente quali fonti interrogare per una data domanda.
3. **Apprendimento continuo dal feedback degli utenti**: un meccanismo di active learning che raccolga le correzioni manuali degli utenti esperti (si veda il punto sull'interfaccia di correzione, paragrafo 6.1) per arricchire iterativamente il dataset di training e migliorare il modello nel tempo, senza richiedere una nuova generazione sintetica da zero.

In sintesi

Il Datavault di DH.ARC contiene milioni di triple RDF, ma finora non offriva un'esplorazione agnostica a SPARQL. Quenya traduce domande in linguaggio naturale in interrogazioni valide, operando tramite un modello linguistico compatto affinato appositamente per questo archivio.

La valutazione ha restituito parametri eccellenti sul piano della sintassi, dell'efficienza temporale e della stabilità. Tuttavia, la consapevolezza dei limiti empirici impone di leggere l'applicativo per quello che è: un prototipo funzionale che risolve una carenza strutturale preesistente.

Gli sviluppi teorizzati, dalla calibrazione del sistema tramite feedback sul campo all'integrazione di motori di intelligenza artificiale per l'elaborazione delle immagini, tracciano un cammino evolutivo chiaro. Seguendolo, l'interfaccia potrà stabilizzarsi come strumento indispensabile per l'esplorazione autonoma del patrimonio documentale.

Riferimenti bibliografici

[Bat89] Marcia J. Bates. "the design of browsing and berrypicking techniques for the online search interface". *Online Review*, 13(5):407–424, 1989.

[FD25] Mohammad Javad Farokhi Darani. *Metadata management with semantic ETLs: From production repositories to dissemination environments.* Master's thesis, Alma Mater Studiorum - Università di Bologna, 2025.

[HSW+22] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, et al. "lora: Low-rank adaptation of large language models". In *International Conference on Learning Representations*, 2022.

[LPP+20] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, et al. "retrieval-augmented generation for knowledge-intensive nlp tasks". In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474, 2020.

[Mar25] Mohammed Maree. "quantifying relational exploration in cultural heritage knowledge graphs with llms: A neuro-symbolic approach for enhanced knowledge discovery". *Data*, 10(4):52, 2025.

[MR08] Peter Morville and Louis Rosenfeld. Information Architecture for the World Wide Web. O'Reilly Media, Sebastopol, CA, 3 edition, 2008.

[PT22] Silvio Peroni and Francesca Tomasi. "approaching digital humanities at university: a cultural challenge". *Journal of Art Historiography*, 27, December 2022.

[Ros07] Luca Rosati. Architettura dell'informazione: Guida alla trovabilità, dagli oggetti quotidiani al Web. Apogeo, 2007.

[Tol70] J. R. R. Tolkien. Il Signore degli Anelli. Bompiani, 1970.

[Vaš24] Linas Vaštakas. *Cultural Heritage Search with Large Language Models: Enhancing the Discoverability of Cultural Heritage Artifacts through Large Language Model-Based Search Systems.* Master's thesis, Linnaeus University, Sweden, 2024.

[VFQP25] Iva Vasic, Hans-Georg Fill, Ramona Quattrini, and Roberto Pierdicca. "knowledge graphs vs. large language models: Competitors or partners in supporting virtual museums". *ACM Journal on Computing and Cultural Heritage*, 18(4), dec 2025.

Sitografia

[ALM26] ALMA MATER STUDIORUM - Università di Bologna. */DH.ARC - Digital Humanities Advanced Research Centre*. `https://centri.unibo.it/dharc/en`, 2026. Ultima visita: 28/06/2026.

[Lyr24] Lyrasis. *Fedora Repository*. `https://lyrasis.org/programs/fedora/`, 2024. Ultima visita: 28/06/2026.

[Met13] Meta Platforms, Inc. *React: The library for web and native user interfaces*. `https://react.dev/`, 2013. Ultima visita: 28/06/2026.

[Mic24] Microsoft. *Phi-3-mini-4k-instruct*. `https://huggingface.co/microsoft/Phi-3-mini-4k-instruct`, 2024. Ultima visita: 28/06/2026.

[Ram18] Sebastián Ramírez. *FastAPI*. `https://fastapi.tiangolo.com/`, 2018. Ultima visita: 28/06/2026.

[SYS24] SYSTAP, LLC. *Blazegraph DB*. `https://blazegraph.com/`, 2024. Ultima visita: 28/06/2026.

[Uns24] Unsloth AI. *Unsloth: Make LLM Finetuning 2x faster and use 70% less VRAM*. `https://unsloth.ai/`, 2024. Ultima visita: 28/06/2026.

[W3C13] W3C SPARQL Working Group. *SPARQL 1.1 Overview*. `https://www.w3.org/TR/sparql11-overview/`, 2013. Ultima visita: 28/06/2026.

[W3C14] W3C RDF Working Group. *RDF 1.1 Concepts and Abstract Syntax*. `https://www.w3.org/TR/rdf11-concepts/`, 2014. Ultima visita: 28/06/2026.