**REPORT**

# GiAs-LLM: Architettura LLM + Agents

IPOTESI DI SOLUZIONE PER L'INTEGRAZIONE DI SERVIZI DI AI IN GISA

## Introduzione

Sulla base del documento "Final Report Bridge Campania", è possibile delineare un'architettura software ad agenti per il sistema **"Hey GISA"**, progettata per supportare gli ispettori sul campo attraverso l'integrazione di Intelligenza Artificiale applicata ai dati del sistema GISA Controlli Ufficiali. Il progetto descritto nel documento propone l'integrazione di un modulo di Intelligenza Artificiale all'interno della piattaforma GISA per ottimizzare i controlli ufficiali attraverso un approccio basato sul rischio. L'obiettivo principale è trasformare i dati storici degli ultimi dieci anni in modelli predittivi capaci di stimare le probabilità di non conformità delle imprese. Gli ispettori sul campo saranno supportati da un assistente virtuale in grado di fornire informazioni in tempo reale e suggerire priorità geografiche e operative. In sintesi, la strategia mira a migliorare l'efficienza amministrativa, garantire la trasparenza e tutelare la salute pubblica attraverso una pianificazione dei controlli più mirata e oggettiva.

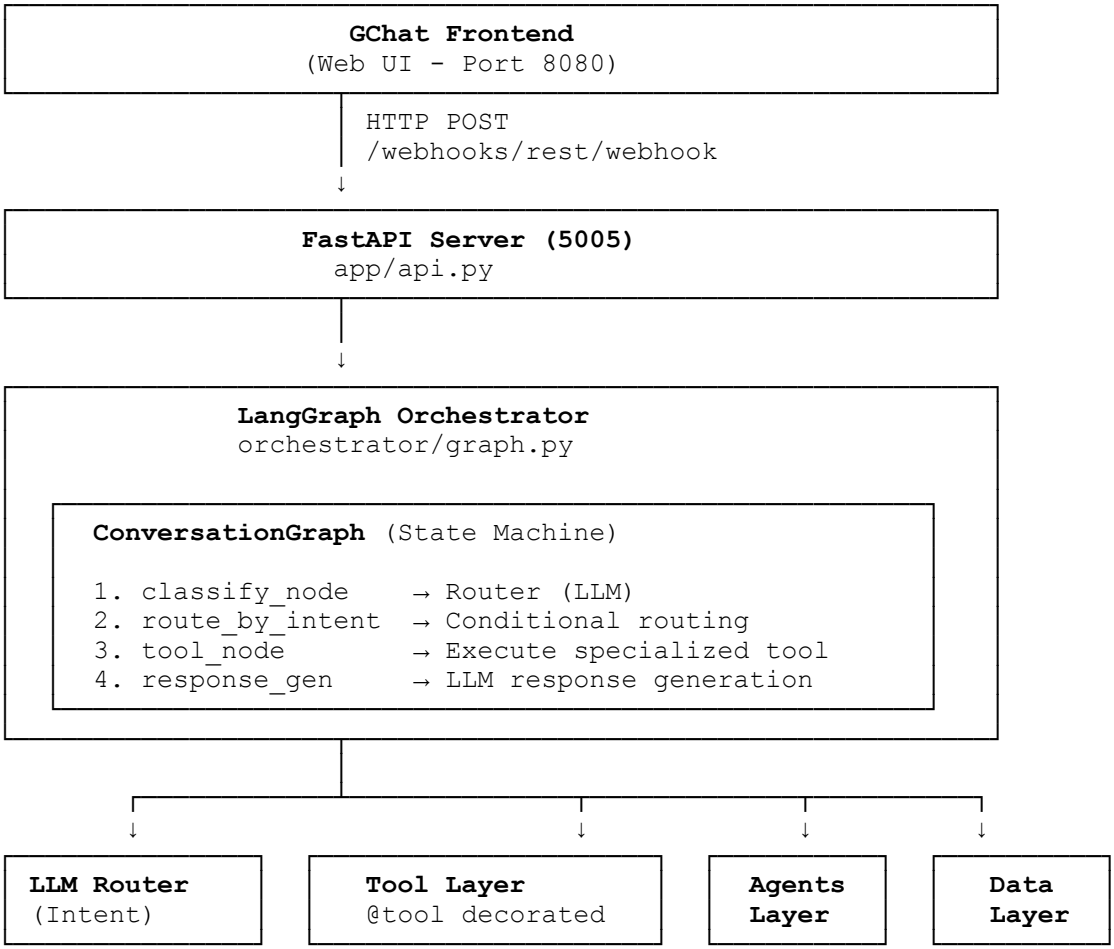
## Overview del modulo di chat bot (GChat)

**GiAs-LLM** è un sistema basato su backend LangGraph + tools, agenti specializzati e LLM per interagire in modalità conversazionale con i dati dell'ecosistema GISA della Regione Campania. Il front-end è rappresentato da una web app che fa da interfaccia utente (GUI) al chatbot. L'interazione col backend avviene attraverso l'uso di FASTAPI.

Nello specifico, l'uso di LangGraph permette di definire flussi di lavoro complessi attraverso l'uso di agenti. A differenza delle catene lineari, introduce il concetto di loop e gestione del persisted state, garantendo un controllo granulare su come l'informazione evolve durante l'esecuzione. In un'architettura ad agenti, il suo vantaggio principale è la capacità di gestire processi iterativi e decisioni ramificate con estrema precisione, superando i limiti dei sistemi "black box". Grazie alla sua struttura a nodi e archi, facilita il coordinamento di più agenti specializzati rendendo il sistema più robusto, prevedibile e facilmente monitorabile rispetto alle implementazioni tradizionali.

Per la parte di backend invece l'uso di FastAPI, framework Python ad alte prestazioni, garantisce la capacità di gestire migliaia di richieste simultanee in modo non bloccante. FASTAPI riduce difatti drasticamente i tempi di sviluppo e gli errori a runtime, garantendo velocità paragonabili a linguaggi come Node.js o Go.

Di seguito è mostrata l'architettura a blocchi della componente di Chatbot ipotizzata




L'immagine illustra l'architettura software a più livelli progettata per un sistema di chat intelligente, dove il flusso dell'informazione parte dal frontend web GChat che comunica tramite protocollo HTTP POST verso il server FastAPI. Quest'ultimo funge da interfaccia API principale ed espone un endpoint specifico per i webhook, agendo da ponte per inoltrare le richieste dell'utente verso il cuore logico del sistema rappresentato dall'orchestratore LangGraph. All'interno dell'orchestratore risiede una macchina a stati chiamata ConversationGraph, la quale gestisce la logica conversazionale attraverso una sequenza di passaggi che includono la classificazione dell'intento tramite un router LLM e il successivo instradamento condizionale.

Il processo prosegue poi con l'esecuzione di strumenti specializzati nel nodo dedicato ai tool, per terminare infine nella generazione della risposta testuale da parte del modello di linguaggio. Per supportare queste operazioni, l'orchestratore interagisce con diversi moduli sottostanti che forniscono le capacità necessarie: un layer per la gestione dei tool decorati, un livello dedicato agli agenti per compiti più complessi e un data layer per l'accesso alle informazioni. L'intera struttura evidenzia una netta separazione tra l'interfaccia utente, la gestione delle chiamate asincrone e la logica di ragionamento del grafo, permettendo un controllo granulare su ogni fase del dialogo.

A seguire saranno descritti i singoli layer che compongono l’architettura proposta.

## Layer 1: LLM Router - Intent Classification

In questa specifica architettura, l'**LLM Router** agisce come il "cervello decisionale" iniziale che analizza l'input testuale proveniente dall'utente per determinarne l'intenzione sottostante. Il suo compito primario è la classificazione dell'intento (Intent Classification), trasformando una richiesta in linguaggio naturale in una categoria specifica che il sistema può processare.

Una volta identificata la natura della richiesta, il router comunica con la logica di **route_by_intent** all'interno del ConversationGraph per decidere quale percorso debba intraprendere l'esecuzione. Ad esempio, può stabilire se la domanda richieda l'attivazione di uno strumento specifico nel **Tool Layer**, se debba essere delegata a un agente specializzato nell'**Agents Layer**, o se sia sufficiente una risposta diretta basata sui dati presenti nel **Data Layer**.

In sintesi, l'LLM Router funge da smistatore che garantisce che ogni query venga gestita dal modulo più competente, ottimizzando le risorse del sistema ed evitando passaggi computazionali non necessari.

## Esempio di Prompt Engineering per la classificazione

```
CLASSIFICATION_PROMPT = """
**TASK**: Classifica il messaggio utente in uno degli intent disponibili.

**MESSAGGIO UTENTE**: "{message}"

**METADATA**:
- ASL: {asl}
- UOC: {uoc}
- User ID: {user_id}

**INTENT DISPONIBILI**:
1. greet: Saluti (es. "ciao", "buongiorno")
2. goodbye: Saluti finali (es. "arrivederci")
3. ask_help: Richieste aiuto (es. "cosa puoi fare?")
4. ask_piano_description: Descrizione piano (es. "di cosa tratta A1?")
5. ask_piano_stabilimenti: Stabilimenti per piano
6. ask_piano_attivita: Attività per piano
7. ask_piano_generic: Query generica su piano
8. search_piani_by_topic: Ricerca piani per argomento
9. ask_priority_establishment: Priorità programmazione
10. ask_risk_based_priority: Priorità basata su rischio storico
11. ask_suggest_controls: Suggerimenti controlli
12. ask_delayed_plans: Piani in ritardo
13. fallback: Non classificabile

**OUTPUT**: JSON
{
  "intent": "intent_name",
  "slots": {"piano_code": "A1", ...},
  "needs_clarification": false
}
"""
```

## La funzione di Slot Extraction

La **slot extraction** è un processo fondamentale nell'elaborazione del linguaggio naturale che consiste nell'identificare ed estrarre frammenti specifici di informazioni da una frase per popolare parametri predefiniti. In un'architettura ad agenti come quella che stiamo analizzando, questo compito avviene tipicamente subito dopo che l'**LLM Router** ha identificato l'intento dell'utente. Nella tabella successiva è mostrato un esempio di slot estratto e frase corrispondente.

| Slot | Descrizione | Esempio |
|---|---|---|
| `piano_code` | Codice piano (A1, B2, C3_F) | "Di cosa tratta il piano **A22**?" |
| `topic` | Argomento ricerca | "Piani su **allevamenti bovini**" |
| `asl` | Azienda Sanitaria Locale | Da metadata |
| `uoc` | Unità Operativa Complessa | Da metadata o risolto da user_id |

Senza la slot extraction, l'orchestratore non saprebbe come alimentare i nodi successivi. In LangGraph, questo processo permette di verificare se lo stato della conversazione è completo: se mancano informazioni obbligatorie (slot vuoti), il grafo può decidere di tornare al nodo di risposta per chiedere chiarimenti all'utente, creando quel ciclo iterativo che caratterizza i sistemi intelligenti.

## *Layer 2: Tool Layer*

Il Tool Layer nell'architettura presentata rappresenta il braccio operativo del sistema, proposto, ovvero l'insieme di funzioni e capacità pratiche che permettono all'intelligenza artificiale di interagire con il mondo reale o con database esterni. In questo schema, gli strumenti sono definiti come funzioni Python "annotate" tramite decoratori specifici, i quali trasformano del semplice codice in componenti pronti per essere invocati dall'orchestratore LangGraph. Quando il flusso di lavoro raggiunge il **tool_node**, il sistema non si limita più a generare testo, ma esegue operazioni concrete basandosi sui parametri strutturati estratti precedentemente durante la fase di slot extraction.

Questi strumenti possono variare notevolmente per complessità e scopo, spaziando dall'interrogazione di un database aziendale nel Data Layer all'invio di email, fino alla chiamata di API di terze parti per recuperare informazioni in tempo reale da GISA. Il Tool Layer funge quindi da interfaccia tra il ragionamento logico dell'LLM e le procedure software deterministiche, garantendo che l'agente non "allucini" i dati ma utilizzi fonti verificate per compiere le proprie azioni.

L'aspetto più potente di questo livello è la sua modularità, poiché permette di aggiungere nuove funzionalità al bot semplicemente scrivendo nuove funzioni e registrandole nel grafo senza dover riaddestrare il modello linguistico sottostante. Una volta che lo strumento ha terminato il suo compito, restituisce il risultato al ConversationGraph, il quale utilizzerà quell'output nel nodo di **response_gen** per formulare una risposta finale accurata e contestualizzata per l'utente su GChat.

Esempio pratico in cui tutti i tool sono decorati con @tool di LangChain:

```
from langchain_core.tools import tool

@tool("piano_description")
def get_piano_description(piano_code: str) -> Dict[str, Any]:
    """
    Recupera descrizione completa di un piano.

    Args:
        piano_code: Codice piano (es. "A1")

    Returns:
        {
            "piano_code": "A1",
            "formatted_response": "Il piano A1 riguarda...",
            "total_variants": 15,
            "raw_data": [...]
        }
    """
```

## Esempio di possibili categorie di Tool

### - Piano Tools (tools/piano_tools.py)

Query su piani di controllo:

| Tool | Descrizione | Input |
|---|---|---|
| `get_piano_description` | Descrizione piano | `piano_code` |
| `get_piano_attivita` | Stabilimenti controllati | `piano_code` |
| `get_piano_correlation` | Correlazione piano-attività | `piano_code` |
| `compare_piani` | Confronto metriche | `piano1_code`, `piano2_code` |

**Esempio Output**:

```
{
  "piano_code": "A1",
  "formatted_response": "**Piano A1**: Controllo carni bovine fresche...",
  "total_controls": 2547,
  "unique_establishments": 134,
  "top_stabilimenti": [...]
}
```

### - Priority Tools (tools/priority_tools.py)

Analisi delle priorità e dei ritardi:

| Tool | Descrizione | Input |
|---|---|---|
| `get_priority_establishment` | Stabilimenti prioritari (programmazione) | `asl`, `uoc`, `piano_code` |
| `get_delayed_plans` | Piani in ritardo | `asl`, `uoc` |
| `suggest_controls` | Mai controllati ad alto rischio | `asl` |

**Business Logic**:

```
# Logica priorità basata su ritardi programmazione
delayed_plans = diff_prog_eseg_df[
    (diff_prog_eseg_df['descrizione_uoc'] == uoc) &
    (diff_prog_eseg_df['diff'] < 0)  # Ritardo = eseguiti < programmati
]
```

```
# Correlazione statistica piano → attività
correlations = controlli_df.groupby(['descrizione_piano', 'attivita_cu']).size()

# Filtra stabilimenti mai controllati correlati
priority_establishments = osa_mai_controllati_df[
    osa_mai_controllati_df['attivita'].isin(correlated_activities)
]
```

**- Risk Tools (tools/risk_tools.py)**

Analisi del rischio rischio storico:

| Tool | Descrizione | Input |
|---|---|---|
| `get_risk_based_priority` | Stabilimenti ad alto rischio NC | `asl`, `piano_code` |

**Risk Scoring**:

Nella soluzione iniziale sarà implementato un risk_score "semplificato", calcolato attraverso al seguente formula pesata

*risk_score = (tot_NC + 3 x tot_NC_gravi)*

Successivamente sarà realizzato un risk_score esplicitati i concetti classici di probabilità e impatto, basato (ipotesi) sui seguenti parametri:

**risk_score = P(NC) × Impatto**

dove:

- *P(NC) = (numero totale di NC)/(numero di controlli)*
- *Impatto = (numero NC gravi)/(numero di controlli)*

A seguire sarà realizzato un modello predittivo basato su tecniche di machine learning, in grado di supportare la pianificazione dei controlli. Nel dettaglio, a partire dai requisiti e dai vincoli descritti nel documento OCSE, è possibile ipotizzare una soluzione di machine learning che rimanga coerente con le esigenze di policy delivery, supporto decisionale e interpretabilità richieste in ambito regolatorio. L'obiettivo del modello sarà quello di stimare la probabilità che, in una futura procedura ispettiva, si verifichi una non conformità significativa o grave (target binario), utilizzando come segnali predittivi informazioni storiche e contestuali relative all'operatore e agli oggetti ispettivi. Poiché i dati hanno una forte struttura temporale ma sono costituiti da serie storiche brevi e irregolari, il modello non apprende direttamente la dinamica temporale tramite architetture ricorrenti, ma sfrutta feature autoregressive derivate dal passato, come ad esempio la presenza di non conformità precedenti, la loro severità media, la recidività, il tempo trascorso dall'ultimo esito negativo e l'intervallo dall'ultima ispezione. Queste informazioni consentiranno di catturare i comportamenti nel tempo degli operatori senza introdurre modelli complessi difficili da giustificare o validare in un contesto regolamentato. Il modello adottata sarà concepita come un modello interpretabile, in cui ciascuna variabile di input contribuisce in modo separato alla stima finale del rischio, fornendo uno strumento comprensibile e utilizzabile dagli ispettori.
Dal punto di vista dell'addestramento, il modello sarà ottimizzato per operare su una popolazione fortemente sbilanciata, nella quale le non conformità significative o gravi

rappresentano una minoranza. La funzione di loss dovrà essere quindi calibrata per privilegiare la capacità di intercettare i casi a rischio, accettando un numero maggiore di falsi positivi, che nel contesto operativo hanno un costo limitato rispetto ai falsi negativi. La validazione seguirà poi una logica temporale rigorosa, in cui il modello viene allenato su dati antecedenti a una certa data e valutato su eventi successivi, in modo da riflettere correttamente lo scenario reale di previsione del rischio futuro. Un elemento centrale dell'approccio dovrà essere la calibrazione delle probabilità predette, che viene effettuata con tecniche dedicate affinché i valori restituiti dal modello possano essere interpretati come stime affidabili della probabilità reale di non conformità e utilizzati direttamente nella pianificazione delle ispezioni. L'output del modello non sarà utilizzato con una soglia fissa, ma come base per costruire un ranking di rischio degli operatori o delle ispezioni, dal quale selezionare a seguire una quota prestabilita di casi ad alto rischio coerente con la capacità ispettiva dell'agenzia, ad esempio il 20% della popolazione. Questo consentirà di controllare la dimensione del campione ispezionato e di massimizzare il guadagno informativo rispetto a un campionamento casuale. La probabilità di non conformità stimata dal modello costituirà infine la componente probabilistica di un modello di rischio più ampio, che viene completato integrandola con una misura di impatto della non conformità, derivata dalle classificazioni di rischio in uso e da indicatori dinamici legati al contesto operativo. In questo modo il modello non si sostituirà al giudizio regolatorio, ma lo supporterà, fornendo una valutazione adattiva, spiegabile e orientata all'allocazione efficiente delle risorse di controllo.

Come suggerito non saranno utilizzati modelli complessi come le reti neurali, mentre saranno presi in considerazione modelli più classici, come i modelli auto regressivi, che permettono la costruzione di nuove feature in grado di far emergere la relazione tra le non conformità del passato e il futuro.

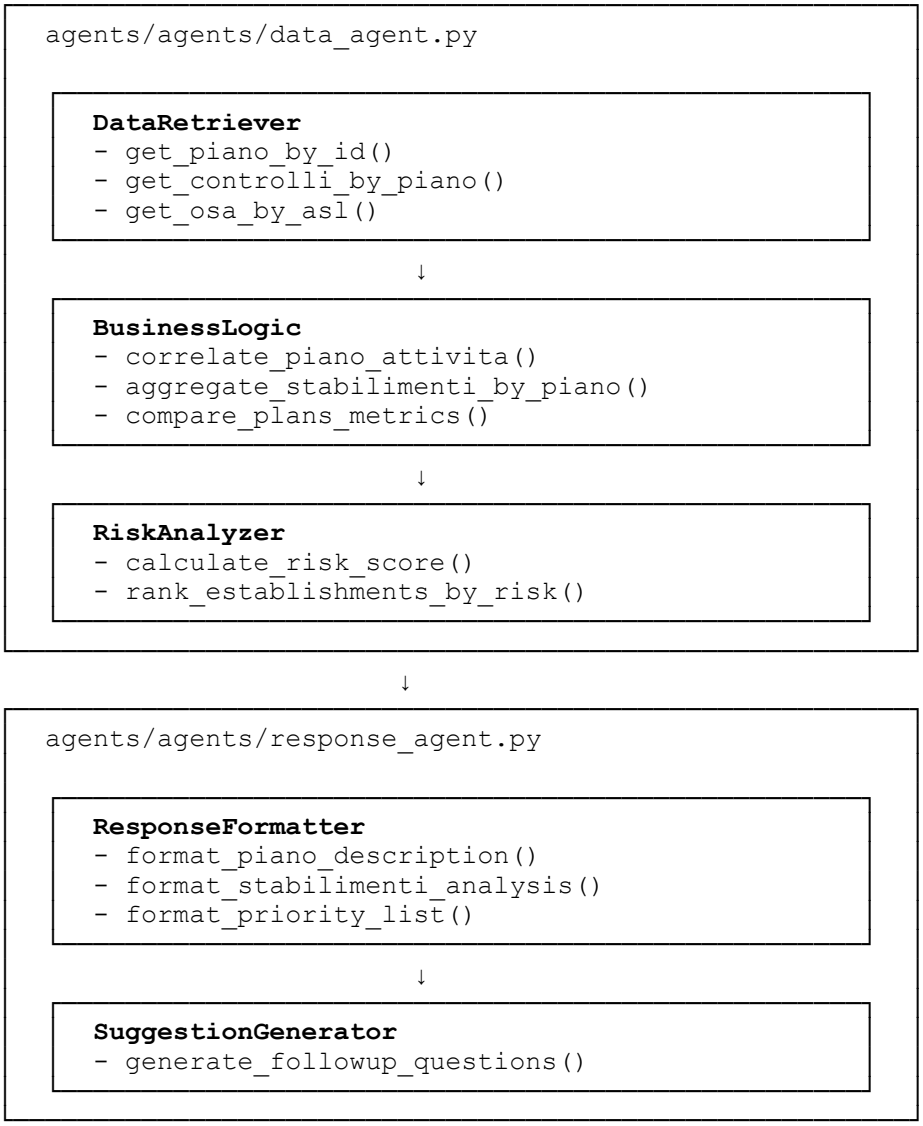
#### - Search Tools (tools/search_tools.py)

Ricerca semantica su piani:

| Tool | Descrizione | Input |
|---|---|---|
| `search_piani_by_topic` | Ricerca piani per argomento | `query` |

## Layer 3: Agents Layer - Business Logic

L'Agents Layer rappresenta il livello di competenza specialistica dell'architettura, dove risiedono entità autonome progettate per risolvere compiti verticali che richiedono un ragionamento più approfondito rispetto a una semplice chiamata a una funzione. A differenza del Tool Layer, che esegue comandi diretti e deterministici, questo strato ospita agenti intelligenti dotati di una propria logica interna e, spesso, di un proprio sotto-grafo di esecuzione. Quando l'orchestratore LangGraph identifica un problema complesso che non può essere risolto con un singolo passaggio, delega la gestione a uno di questi agenti specializzati, i quali operano come esperti di dominio indipendenti all'interno del sistema complessivo. Il funzionamento di questo strato si basa sulla capacità dell'agente di ricevere un obiettivo di alto livello e pianificare autonomamente i passi necessari per raggiungerlo, decidendo quali strumenti richiamare e come interpretare i risultati ottenuti. L'orchestratore funge da supervisore che smista il lavoro, mentre gli agenti nello strato dedicato portano a termine la missione specifica. Una volta completato il compito, l'agente restituisce l'output strutturato al ConversationGraph principale, che lo integra nello stato globale della conversazione per finalizzare la risposta all'utente. Di seguito è mostrato uno schema di esempio di alcuni possibili agenti per il contesto GISA.




## Esempio di esecuzione di un flusso (dalla query alla risposta)

**User Query**: “Di cosa tratta il piano A1?”

```
1. classify_node:
   ├─ LLM Router analizza messaggio
   ├─ Estrae slot: {"piano_code": "A1"}
   └─ Classifica: intent = "ask_piano_description"

2. route_by_intent:
   └─ Conditional edge → "piano_description_tool"

3. piano_description_tool:
   ├─ Chiama piano_tool(action="description", piano_code="A1")
   │  ├─ DataRetriever.get_piano_by_id("A1")
   │  ├─ BusinessLogic.extract_unique_piano_descriptions()
   │  └─ ResponseFormatter.format_piano_description()
   └─ Salva in state["tool_output"]

4. response_generator_node:
   ├─ Recupera tool_output["formatted_response"]
   ├─ Opzionale: LLM arricchisce risposta
   └─ Salva in state["final_response"]

5. END → Ritorna stato finale
```

## *Data Layer*

Il Data Layer funge da memoria storica e base di conoscenza dell'intera architettura proposta, rappresentando lo strato dove le informazioni vengono archiviate, recuperate e modificate in modo persistente. Mentre l’orchestratore LangGraph gestisce lo stato temporaneo della conversazione in corso, questo livello si occupa di interfacciarsi con sorgenti di dati strutturate e non strutturate, come database relazionali, database vettoriali per la ricerca semantica o sistemi di file esterni. Quando un agente o uno strumento nel Tool Layer necessita di un'informazione specifica per completare un compito, invia una richiesta a questo strato che traduce la necessità logica in una query tecnica verso il sistema di storage appropriato. In un sistema basato su modelli linguistici, il Data Layer gioca un ruolo fondamentale soprattutto nell'implementazione del Retrieval-Augmented Generation (RAG), fornendo al modello il contesto necessario per rispondere in modo accurato senza allucinazioni. A tal proposito si specifica che il RAG costituisce un elemento di ottimizzazione (non necessario). La possibilità di usarlo è data dalla disponibilità di alcune informazioni (es. tassonomie, ecc.) o da possibili altre informazioni non strutturate recuperabili (es. in PDF), come procedure, normative, ecc.

Oltre al recupero delle conoscenze, questo strato è responsabile della persistenza del profilo utente e dello storico dei dialoghi a lungo termine, permettendo al sistema di "ricordare" interazioni avvenute in sessioni passate. La comunicazione tra il Data Layer e il resto dell'architettura garantisce che ogni decisione presa dall'LLM Router o ogni azione dell'Agents Layer sia fondata su dati reali e aggiornati, rendendo l'intera struttura non solo intelligente, ma anche affidabile e coerente nel tempo.

## Esempio di stack software in produzione

Lo stack software ipotizzato delinea un'architettura moderna per agenti AI, distribuita su più livelli che vanno dall'esposizione esterna fino al cuore dell'inferenza linguistica.

### Livello Infrastrutturale e di Rete

Al vertice del sistema troviamo un **Reverse Proxy Nginx** che opera sulle porte standard 80/443. Questo componente è fondamentale per la sicurezza e la stabilità, occupandosi della terminazione SSL e del bilanciamento del carico verso i servizi interni. Dietro il proxy, l'architettura si divide in due rami principali: il frontend, gestito da un server (scritto ad esempio in **Go)** GChat sulla porta 8080, e il backend logico.

### Livello Backend e Orchestrazione

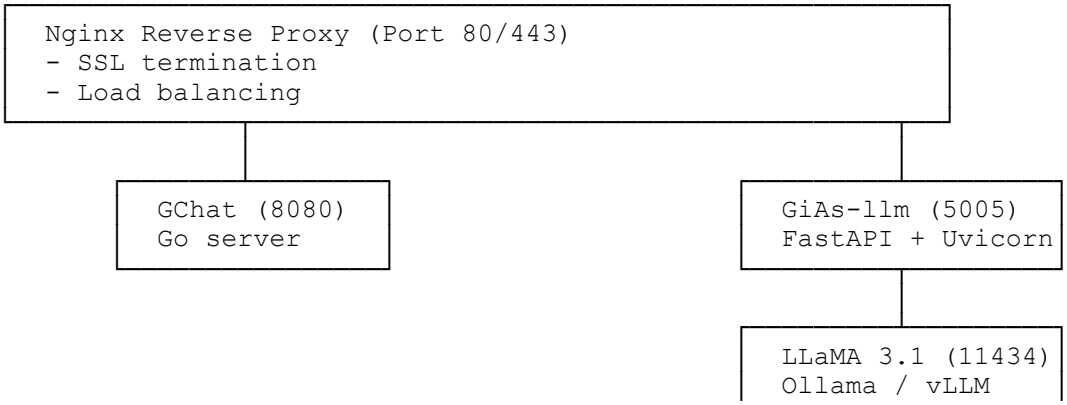
Il nucleo applicativo è servito da **FastAPI**, eseguito tramite il server ASGI **Uvicorn** sulla porta 5005. Questo strato funge da gateway per le richieste HTTP POST e ospita il **LangGraph Orchestrator**. LangGraph gestisce una macchina a stati complessa (**ConversationGraph**) che esegue il workflow logico:

- **Classificazione:** Utilizza un nodo router basato su LLM per identificare l'intento.
- **Routing Condizionale:** Indirizza il flusso in base alla decisione del router.
- **Esecuzione Tool:** Attiva strumenti specializzati decorati (Tool Layer).
- **Generazione Risposta:** Produce l'output finale per l'utente.

### Livello di Servizio e Inferenza LLM

Il sistema si appoggia a una gerarchia di strati funzionali che includono un **Agents Layer** per compiti autonomi, un **Tool Layer** per le operazioni pratiche e un **Data Layer** per la persistenza e il recupero dati. La potenza computazionale per l'intelligenza artificiale è fornita da **LLaMa 3.1**, servito tramite **Ollama** o **vLLM** sulla porta 11434. Questa separazione tra l'orchestratore logico (FastAPI/LangGraph) e il motore di inferenza (Ollama/LLaMa) permette di scalare le risorse GPU indipendentemente dalla logica applicativa.

Di seguito è mostrato lo schema della architettura GiAs-llm.



## Scalabilità dell’architettura

L'architettura mostrata garantisce un'elevata scalabilità grazie alla natura modulare e asincrona dei suoi componenti principali. Il punto d'ingresso, gestito da FastAPI, è nativamente progettato per la scalabilità orizzontale: essendo un framework ASGI, può gestire migliaia di connessioni simultanee con un uso minimo di risorse e può essere facilmente replicato su più container dietro un bilanciatore di carico per far fronte a picchi di traffico.

A livello logico, l'uso di LangGraph introduce una scalabilità di tipo funzionale. Poiché il ConversationGraph separa nettamente la gestione dello stato dalla logica dei singoli nodi, è possibile scalare indipendentemente i diversi layer sottostanti. Ad esempio, se il Tool Layer o l'Agents Layer richiedono operazioni computazionalmente intense (come l'elaborazione di documenti o chiamate a API esterne lente), questi possono essere isolati in microservizi dedicati, evitando che i colli di bottiglia di un singolo agente blocchino l'intero orchestratore.

Infine, il Data Layer garantisce la scalabilità del dato attraverso l'uso di database distribuiti e sistemi di caching. La separazione tra lo stato della sessione (gestito internamente al grafo) e i dati persistenti permette di mantenere il sistema reattivo anche all'aumentare del numero di utenti o della complessità della base di conoscenza. Questa struttura a strati assicura che il sistema possa crescere non solo in termini di volume di messaggi, ma anche in termini di nuove funzionalità, senza richiedere una ristrutturazione del nucleo centrale.

## Roadmap di sviluppo del prototipo base (3 mesi)

### Mese 1: Connettività

Il primo mese si concentra sulla creazione dello "scheletro" del sistema. Si inizia con la configurazione del server FastAPI e l'integrazione del webhook per ricevere messaggi dal frontend GChat. Parallelamente, si definisce la struttura base di LangGraph, implementando un primo ConversationGraph semplificato che contenga solo il nodo di classificazione e un generatore di risposte statiche. In questa fase, il Data Layer viene configurato per gestire la persistenza minima delle sessioni, assicurando che il sistema sia in grado di mantenere il filo del discorso tra un messaggio e l'altro.

### Mese 2: Intelligenza Operativa e Tooling

Nel secondo mese l'attenzione si sposta sulle capacità d'azione. Verrà implementato l'LLM Router con logiche di slot extraction avanzate per trasformare il linguaggio naturale in dati strutturati. Questi dati alimentano il Tool Layer, dove vengono sviluppati i primi strumenti reali, come l'interrogazione di database SQL o l'integrazione con API esterne. Si introduce anche la logica di RAG (Retrieval-Augmented Generation) nel Data Layer, permettendo al sistema di attingere a documenti e basi di conoscenza per fornire risposte precise e documentate.

### Mese 3: Orchestrazione Multi-Agente ed Ottimizzazione

L'ultimo mese è dedicato alla complessità e al raffinamento. Si sviluppa l'Agents Layer, introducendo agenti specializzati capaci di gestire sotto-task autonomi e complessi che richiedono più passaggi di ragionamento. Viene perfezionato l'orchestratore per gestire i cicli (loop) e le correzioni degli errori nel caso in cui un agente o un tool falliscano. Infine, si effettuano test di carico per verificare la scalabilità dell'intera infrastruttura, ottimizzando i tempi di risposta e rifinendo l'interfaccia utente finale per garantire un'esperienza fluida e professionale.