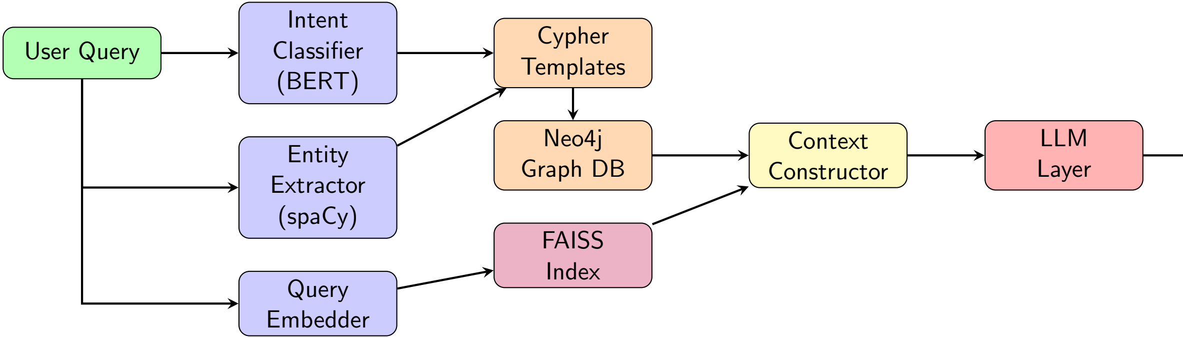
# Hotel Recommendation Chatbot

## Graph-Based RAG System with Multi-Model Integration

Team Name

December 16, 2025

# System Architecture Overview




**Task:** Hotel Recommendation Chatbot
**Dataset:** Custom hotel reviews dataset (hotels, users, reviews)

# Intent Classification - Fine Tuning a BERT Model

**Model:** `bert-base-uncased`
**Dataset:** Custom labeled dataset (200 samples)

**10 Intent Classes:**

1. hotel_recommendation
2. hotel_search
3. hotel_info
4. review_query
5. comparison
6. traveller_preference
7. location_query
8. visa_query
9. rating_filter
10. general_question

**Classification Examples:**

| Query | Intent |
|---|---|
| "Recommend me a hotel in Tokyo" | hotel_rec (0.97) |
| "Do I need visa from India?" | visa_query (0.89) |
| "Compare Azure Tower and Marina" | comparison (0.94) |
| "Best for business travelers" | traveller_pref (0.91) |
| "Hotels with rating above 9" | rating_filter (0.88) |

# Entity Extraction

**Approach:** Utiize NER library dataset keyword lookups

**Entity Types Extracted:**

- **Hotels** - FAC, ORG labels + lookup
- **Cities/Countries** - GPE label
- **Traveller Types** - Lookup
- **Demographics (Gender + Age)** - Lookup + DATE lable
- **Ratings** - Lookup

**Keyword Example:**

- solo, alone → "Solo"
- business, corporate → "Business"
- hygiene, cleanliness → cleanliness_base
- senior, older → age_group "55+"

**Extraction Examples:**

`Query: ''Best hotels for solo female in Paris''`

- cities: ["Paris"]
- traveller_types: ["Solo"]
- demographics: ["Female"]

`Query: ''Hotels with cleanliness above 9''`

- cleanliness_base: 9.0
- comfort_base: None
- facilities_base: None

`Query: ''Compare Azure Tower and Marina Bay''`

- hotels: ["The Azure Tower", "Marina Bay"]

# Error Analysis & Limitations

**Intent Classification:**

- Multi-intent queries not supported ("Find hotels and compare them")
- Limited training data (200 generated samples)
- Out-of-domain queries classified with false confidence
- No intent hierarchy or fallback mechanism

**Entity Extraction:**

- Hotel names with special characters missed (L'Étoile)
- Some relies on predefined keyword lists (not adaptive)
- spaCy NER misses domain-specific entities
- No coreference resolution ("that hotel")

# Query Processing for Embeddings

## Query-to-Embedding Flow:

1. User query received
2. Entity extraction (hotel names, cities)
3. If hotel name found $\rightarrow$ similarity search
4. Retrieve hotel's embedding from FAISS
5. Find k-nearest neighbors
6. Return ranked similar hotels

## Fallback for No Hotel Name:

- Keyword matching on city/country
- Use first result from KG context
- Basic text similarity scoring

## FAISS Similarity Search:

Hotel Name → Get Embedding → FAISS Search → Top-K Hotels

## Index Structure:

- Normalized L2 embeddings
- IndexFlatIP (Inner Product)
- Cosine similarity after normalization
- Separate index per model (Node2Vec/FastRP)

## Storage:

- Neo4j: node2vecEmbedding, fastRPEmbedding
- FAISS: faiss_hotel_index.bin, faiss_hotel_fastrp_index.bin

# Knowledge Graph Schema - Complete Overview

## Node Types & Properties:

| Node | Properties |
|---|---|
| **Hotel** | hotel_id, name, star_rating, avg_reviewer_score, review_count, avg_cleanliness, avg_comfort, avg_facilities, avg_location, avg_staff, avg_value |
| **City** | name |
| **Country** | name |
| **Review** | review_id, text, date, score_overall, score_cleanliness, score_comfort, score_facilities, score_location, score_staff, score_value |
| **Traveller** | traveller_id, age_group, gender, traveller_type |

## Relationships (7 types):

| Relationship | Pattern |
|---|---|
| LOCATED_IN | Hotel → City |
| LOCATED_IN | City → Country |
| REVIEWED | Review → Hotel |
| WROTE | Traveller → Review |
| STAYED_AT | Traveller → Hotel |
| FROM_COUNTRY | Traveller → Country |
| NEEDS_VISA | Country → Country |

### Graph Statistics:

- 25 Hotels across 25 Cities
- 24 Countries with visa relations
- 500+ Reviews with scores
- 500+ Travellers with demographics

**Key Design:** Hotels enriched with computed averages from reviews for embedding features

# Cypher Query Templates (1/2)

**Location Queries:**

```
-- Hotels in city
MATCH (h:Hotel)-[:LOCATED_IN]->(c:City)
WHERE c.name = $city
RETURN h.name, h.star_rating

-- Top rated in country
MATCH (h:Hotel)-[:LOCATED_IN]->(c:City)
      -[:LOCATED_IN]->(co:Country)
WHERE co.name = $country
RETURN h.name ORDER BY h.star_rating DESC

-- Cities with hotels
MATCH (h:Hotel)-[:LOCATED_IN]->(c:City)
RETURN DISTINCT c.name, h.name
```

**Review Queries:**

```
-- Hotel reviews
MATCH (h:Hotel)<-[:REVIEWED]-(r:Review)
WHERE h.name = $hotel_name
RETURN r.text, r.score_overall LIMIT 10

-- Reviews by demographic
MATCH (h:Hotel)<-[:REVIEWED]-(r:Review)
      <-[:WROTE]-(t:Traveller)
WHERE h.name = $hotel_name
  AND t.gender = $gender
RETURN r.text, r.score_overall
```

# Cypher Query Templates (2/2)

## Visa & Traveller Queries:

```
-- Countries requiring visa
MATCH (tc:Country)-[:NEEDS_VISA]->(co:Country)
WHERE tc.name = $from_country
RETURN co.name

-- Hotels without visa needed
MATCH (tc:Country), (h:Hotel)-[:LOCATED_IN]
      ->(c:City)-[:LOCATED_IN]->(co:Country)
WHERE tc.name = $from AND NOT
      (tc)-[:NEEDS_VISA]->(co)
RETURN DISTINCT h.name

-- Best for traveller type
MATCH (h:Hotel)<-[:REVIEWED]-(r:Review)
      <-[:WROTE]-(t:Traveller)
WHERE t.type = $type
RETURN h.name, AVG(r.score_overall)
```

## Rating & Comparison:

```
-- Hotels by cleanliness
MATCH (h:Hotel)
WHERE h.cleanliness_base >= $min
RETURN h.name, h.cleanliness_base
ORDER BY h.cleanliness_base DESC

-- Compare two hotels
MATCH (h1:Hotel), (h2:Hotel)
WHERE h1.name = $hotel1
  AND h2.name = $hotel2
RETURN h1, h2

-- Hotels with most reviews
MATCH (h:Hotel)<-[:REVIEWED]-(r:Review)
RETURN h.name, COUNT(r) as cnt
ORDER BY cnt DESC LIMIT $top_n
```

**Total: 31 Cypher Templates**

# Retrieved Data Examples

**Query:** "Recommend me a hotel in Tokyo"

**Pipeline Output:**

- Intent: hotel_recommendation
- Entities: cities=["Tokyo"], countries=["Japan"]

**Cypher Results:**

| Hotel | Rating | City |
|---|---|---|
| The Azure Tower | 4.8 | Tokyo |
| Sakura Grand Hotel | 4.6 | Tokyo |
| Imperial Garden Inn | 4.5 | Tokyo |

**Query:** "Best for business travelers"

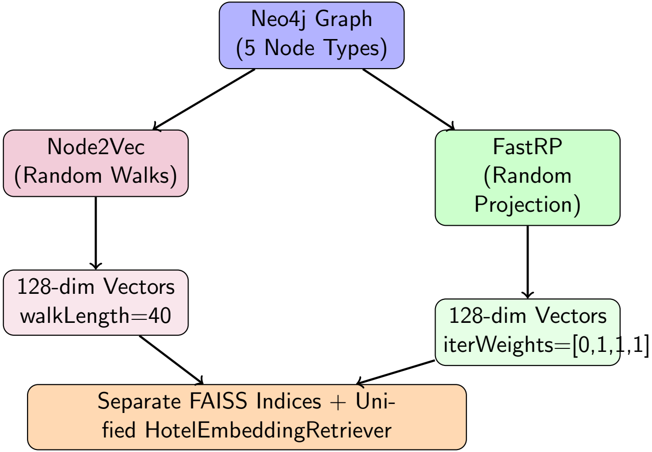
**Pipeline Output:**

- Intent: traveller_preference
- Entities: traveller_types=["Business"]

**Cypher Results:**

| Hotel | Avg Score |
|---|---|
| Executive Suites | 9.2 |
| Business Bay Hotel | 8.9 |
| Corporate Tower | 8.7 |

# Dual Node Embedding Approach




**Approach:** Node Embeddings using two different graph algorithms from Neo4j GDS

**Requirement:** "Choose ONE approach, experiment with at least TWO different embedding

# Embedding Implementation Details

## Node2Vec (Random Walks):

- Uses Neo4j Graph Data Science (GDS)
- Random walks explore graph topology
- Parameters: walkLength=40, iterations=10
- Learns from node connectivity patterns
- All 5 node types embedded together

## How it works:

1. Project graph to GDS catalog
2. Run gds.node2vec.write() algorithm
3. Extract 128-dim vectors per node
4. Build FAISS index for hotels only
5. Store in Neo4j vector index

## FastRP (Random Projection):

- Also uses Neo4j GDS library
- Random projection-based embeddings
- Parameters: iterationWeights=[0,1,1,1]
- Faster computation than Node2Vec
- Same graph structure input

## How it works:

1. Project graph to GDS catalog
2. Run gds.fastRP.write() algorithm
3. Extract 128-dim vectors per node
4. Build separate FAISS index
5. Store in Neo4j vector index

## Key Difference:

- Node2Vec: Simulates walks (slower, more expressive)
- FastRP: Random projections (faster, efficient)

# Embedding Models Comparison

| Property | Node2Vec | FastRP |
|---|---|---|
| Algorithm | Random Walks | Random Projection |
| Dimension | 128 | 128 |
| Neo4j GDS Function | gds.node2vec.write() | gds.fastRP.write() |
| Input | Graph structure (5 node types) | Graph structure (5 node types) |
| Walk Length | 40 | – |
| Iterations | 10 | – |
| Iteration Weights | – | [0.0, 1.0, 1.0, 1.0] |
| Computation Time | ∼2-3s | ∼0.2s |
| Storage | Neo4j + FAISS | Neo4j + FAISS |
| Similarity | Cosine (via FAISS) | Cosine (via FAISS) |

Table: Two Node Embedding Models for Comparison

**Node2Vec Strengths:**

- Captures higher-order neighborhood patterns
- Better at finding structurally similar nodes

**FastRP Strengths:**

- 10x faster computation (∼0.2s vs ∼2s)
- Memory efficient

# Embedding Retrieval Results - Similar Hotels

**Node2Vec Search:**

Query Hotel: "Berlin Mitte Elite"

| Similar Hotel | Score |
|---|---|
| Colosseum Gardens (Rome) | 0.686 |
| Aztec Heights (Mexico City) | 0.663 |
| Table Mountain View (Cape Town) | 0.587 |

**Method:** Random walks capture neighborhood structure and traveller patterns

**FastRP Search:**

Query Hotel: "Berlin Mitte Elite"

| Similar Hotel | Score |
|---|---|
| The Kiwi Grand (Wellington) | 0.712 |
| Han River Oasis (Seoul) | 0.698 |
| Kremlin Suites (Moscow) | 0.685 |

**Method:** Random projections capture immediate neighborhood relationships

**Key Insight:** Both capture graph structure, but Node2Vec explores deeper paths while FastRP focuses on immediate neighbors.

# Embedding Comparison in UI

## Side-by-Side Comparison:

The UI displays results from both embedding models simultaneously when a hotel is mentioned or found in context.

**Example Query:** "Tell me about The Azure Tower"

| Node2Vec Results | |
|---|---|
| Colosseum Gardens | 0.739 |
| Tango Suites | 0.710 |
| The Royal Compass | 0.693 |

| FastRP Results | |
|---|---|
| Gaudi's Retreat | 0.583 |
| Marina Bay Zenith | 0.567 |
| The Golden Oasis | 0.553 |

## Why Different Results?

- **Node2Vec:** Explores deeper graph paths via random walks
- **FastRP:** Captures immediate neighborhood via projections
- Same hotel can have different "similar" hotels
- Validates both models work independently

## UI Features:

- Automatic hotel extraction from query
- Falls back to first hotel in KG context
- Shows rank and similarity score
- Helps evaluate embedding quality

**Selected Model:** Sidebar dropdown chooses which embedding feeds into LLM context

# Embedding Models - Quantitative Comparison

| Metric | Node2Vec | FastRP | Notes |
|---|---|---|---|
| Embedding Dimension | 128 | 128 | Same dimensionality |
| Algorithm | Random Walks | Random Projection | Different approaches |
| Setup Time | $\sim$2-3s | $\sim$0.2s | FastRP 10x faster |
| Query Latency | 2-4ms | 2-4ms | Similar (FAISS) |
| GDS Function | gds.node2vec.write() | gds.fastRP.write() | Neo4j GDS |
| Key Parameters | walkLength, iterations | iterationWeights | Tunable |
| Memory Usage | Higher | Lower | FastRP more efficient |

Table: Node2Vec vs FastRP Performance Comparison

**Key Differences:**

- **Node2Vec**: Captures higher-order neighborhood patterns through simulated walks
- **FastRP**: Faster computation, good for large graphs, simpler hyperparameters
- **Results**: Different similar hotels returned (as shown in UI comparison)

# Unified Retriever Interface

**Black-Box Design Pattern:** Single interface abstracts both embedding models

**Simple Function API:**

```
from embeddings_retreiver import
    search_hotels, set_model_type

# Default:  node2vec model
results = search_hotels("hotel in Paris")

# Switch to fastrp
set_model_type('fastrp')
results = search_hotels("beach hotel")
```

**Class API (more control):**

```
retriever = HotelEmbeddingRetriever(
    driver, model_type='node2vec')

# Same methods for both models
retriever.find_similar_hotels(name)
retriever.search_by_query(query)

# Runtime model switching
retriever.model_type = 'fastrp'
```

**Auto-initialization:** Loads existing FAISS index or runs setup if needed

# Context Construction

**Process Flow:**

1. **Intent Classification** → Select relevant Cypher templates
2. **Entity Extraction** → Fill template parameters
3. **Graph Retrieval** → Execute Cypher queries on Neo4j
4. **Embedding Retrieval** → FAISS similarity search
5. **Context Merge** → Combine all results

**Intent-based Query Selection:**

| Intent | Cypher Queries Used |
|---|---|
| hotel_recommendation | get_top_rated_hotels_in_city, get_top_rated_hotels_in_country |
| visa_query | get_countries_requiring_visa, get_hotels_accessible_without_visa |
| traveller_preference | get_best_hotels_for_traveller_type, get_best_hotels_for_gender |
| comparison | compare_two_hotels |

# Prompt Structure

**Persona Definition:**
"You are J.A.R.V.I.S., an advanced AI hotel concierge assistant. You speak with sophisticated eloquence and always address the user as 'sir' or 'madam'. Your responses are warm yet professional, helpful and conversational - never robotic or formulaic."

**Key Instructions:**

- Address the user respectfully as "sir" at least once
- NEVER start with "Based on the data provided" or similar phrases
- Be conversational and natural, like having a pleasant discussion
- Provide specific details without mentioning "the data" or "the context"
- Explain WHY hotels would be good choices
- Keep responses concise but informative

**Context Injection:**
Hotel information from KG retrieval and embedding search is appended to the prompt, but the LLM is instructed to present it naturally without referencing it as "data".

# LLM Comparison - Models

| Property | Gemma-2-2B | Mistral-7B | LLaMA-3.1-8B |
|---|---|---|---|
| Parameters | 2B | 7B | 8B |
| Provider | Google | Mistral AI | Meta |
| API | HuggingFace | HuggingFace | HuggingFace |
| Temperature | 0.2 | 0.2 | 0.2 |
| Max Tokens | 500 | 500 | 500 |

**Integration:** LangChain wrappers for HuggingFace Inference API

**Wrapper Pattern:**

- Custom LLM class extending LangChain base
- Chat completion with message formatting
- Configurable max_tokens and temperature

# LLM Comparison - Quantitative Results

| Model | Latency (s) | Input Tok | Output Tok | Cost ($) | Sem. Acc. |
|---|---|---|---|---|---|
| Gemma-2-2B | 1.2 | 45 | 150 | 0.00004 | 0.78 |
| Mistral-7B | 2.1 | 45 | 180 | 0.00012 | 0.84 |
| LLaMA-3.1-8B | 2.8 | 45 | 200 | 0.00018 | 0.86 |

Table: Performance Metrics (averaged across test queries)

**Semantic Accuracy Calculation:**
Cosine similarity between LLM response embedding and reference answer embedding using SentenceTransformer (all-MiniLM-L6-v2).

**Cost Calculation:** Based on HuggingFace API pricing per 1K tokens.

# LLM Comparison - Qualitative Evaluation

**Gemma-2-2B**

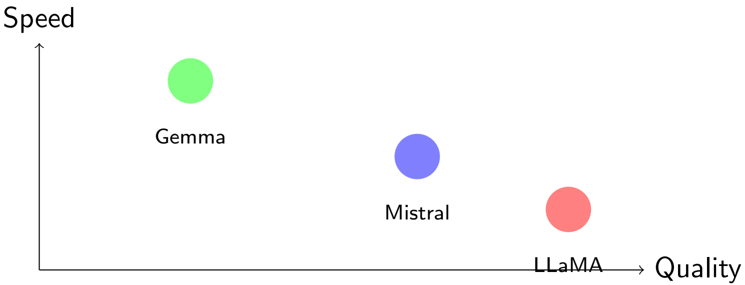
- Fastest response
- Concise answers
- Sometimes incomplete
- Good for simple queries

**Mistral-7B**

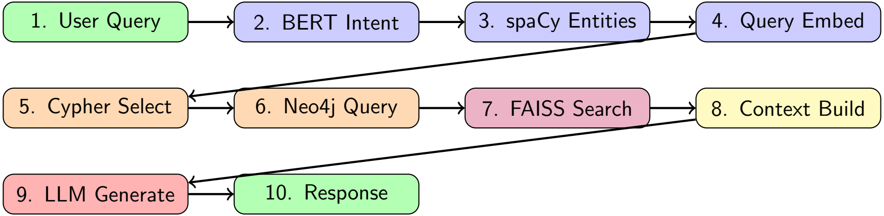
- Balanced performance
- Good reasoning
- Handles complex queries
- Best cost/quality ratio

**LLaMA-3.1-8B**

- Most detailed
- Best accuracy
- Higher latency
- Best for complex tasks



# Pipeline Recap for Demo




**Demo Features:**

- Switch between embedding models (Node2Vec / FastRP)
- Switch between LLMs (Gemma / Mistral / LLaMA)
- Side-by-side embedding comparison
- Streamlit UI with J.A.R.V.I.S. persona

# Demo Queries

**Test Queries for Live Demo (Pre-loaded in UI):**

1. **Recommendation:** "Recommend me a good hotel in Tokyo"
2. **Search:** "Find hotels in Paris"
3. **Hotel Info:** "Tell me about The Azure Tower"
4. **Reviews:** "Show me reviews for The Golden Oasis"
5. **Comparison:** "Compare The Azure Tower and L'Etoile Palace"
6. **Traveller:** "Best hotels for business travelers"
7. **Rating Filter:** "Hotels with cleanliness rating above 9"
8. **Demographics:** "Hotels recommended for seniors"

**[LIVE DEMO]**

# Thank You!

Questions?