A

Major Project

On

KNOWLEDGE GRAPH FROM UNSTRUCTURED DATA

Submitted in partial fulfillment of the requirements for the award of degree

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

Submitted By

SRUTHI JADHAV **217Z1A05G8**

Under the Guidance of

Mr.T.MADHU

Assistant Professor

SCHOOL OF ENGINEERING

Department of Computer Science and Engineering

NALLA NARASIMHA REDDY

EDUCATION SOCIETY'S GROUP OF INSTITUTIONS

(AN AUTONOMOUS INSTITUTION)

Approved by AICTE, New Delhi, Chowdariguda(v), Korremula 'x' Roads, Via Narapally, Ghatkesar(M), Medchal (Dist), Telanagan-500088

2024-2025

NALLA NARASIMHA REDDY
Education Society's Group of Institutions - Integrated Campus
(UGC AUTONOMOUS INSTITUTION)

SCHOOL OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the project report titled "**KNOWLEDGE GRAPH FROM UNSTRUCTURED DATA**" is being submitted by **Sruthi Jadhav (217Z1A05G8)** in Partial fulfilment for the award of **Bachelor of Technology in Computer Science and Engineering** is a record bonafide work carried out by her. The results embodied in this report have not been submitted to any other University for the award of any degree.

Internal Guide
(Mr.T.Madhu)

Head of the Department
(Dr. K. Rameshwaraiah)

Submitted for Viva Voce Examination held on……

External Examiner

DECLARATION

I, T.Srilasya, the student of **Bachelor of Technology in Computer Science and Engineering, Nalla Narasimha Reddy Education Society's Group of Institutions**, Hyderabad, Telangana, hereby declare that the work presented in this project work entitled "**Knowledge Graph From Unstructured Data"** is the outcome of my own bonafide work and is correct to the best of my knowledge and this work has been undertaken by taking care of engineering ethics. It contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the university or other institute of higher learning.

By:

SRUTHI JADHAV 217Z1A05G8

Date:

Signature:

ACKNOWLEDGEMENT

I express my sincere gratitude to my guide **Mr.T.Madhu**, Assistant Professor, in Computer Science and Engineering Department, NNRESGI, who motivated throughout the period of the project for his valuable and intellectual suggestions apart from his adequate guidance, constant encouragement right throughout our work.

I profoundly express my sincere thanks to **Dr. K. Rameshwaraiah**, Professor & Head, Department of Computer Science and Engineering, NNRESGI, for his cooperation and encouragement in completing the project successfully.

I wish to express my sincere thanks to **Dr. G.Janardhana Raju**, Dean School of Engineering, NNRESGI, for providing the facilities for completion of the project.

I wish to express my sincere thanks to **Dr. C. V. Krishna Reddy**, Director NNRESGI for providing the facilities for completion of the project.

I wish to record my deep sense of gratitude to our Project In-charge **Mrs. Ch. Ramya**, Assistant Professor, in Computer Science and Engineering Department, NNRESGI, for giving her insight, advice provided during the review sessions and providing constant monitoring from time to time in completing our project and for giving us this opportunity to present the project work.

Finally, I would like to thank Project Co-Ordinator, Project Review Committee (PRC) members, all the faculty members and supporting staff, Department of Computer Science and Engineering, NNRESGI for extending their help in all circumstances.

By:

SRUTHI JADHAV 217Z1A05G8

ABSTRACT

A knowledge graph from unstructured data is a structured representation of knowledge extracted from diverse unstructured sources such as documents, PDFs, and raw textual content. Traditionally, such systems relied on rule-based natural language processing (NLP) techniques like tokenization, part-of-speech tagging, and named entity recognition using tools like SpaCy. In contrast, this project employs Gemini AI, a powerful generative large language model developed by Google, which can directly interpret human language, extract relevant entities, and identify their relationships using prompt-based instructions.

Once Gemini AI extracts these relationships, they are stored in a Neo4j graph database where entities are represented as nodes and their connections as edges. This streamlined process enables high accuracy and flexibility, allowing for the conversion of complex text into an intuitive and queryable graph format. Applications of this system extend across multiple industries including healthcare, finance, education, and legal analytics, helping stakeholders make informed, data-driven decisions.

Keywords: Gemini AI, Knowledge Graph, Unstructured Data, Relationship Extraction, Neo4j, Large Language Models, Google AI, Entity Recognition, Graph Database, Prompt Engineering

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

In today's digital age, vast amounts of information are being produced every second in the form of unstructured data, such as articles, blogs, PDFs, emails, and social media posts. Unlike structured data that fits neatly into tables and databases, unstructured data lacks a predefined model or format, making it difficult for machines to understand and process. This poses a major challenge in extracting useful knowledge from such sources.

One innovative solution to this problem is the construction of knowledge graphs, which visually and logically represent information as entities (nodes) and their relationships (edges). Traditionally, creating such graphs required complex rule-based Natural Language Processing (NLP) tools like SpaCy, which demanded significant manual tuning and often struggled with understanding context or ambiguity in sentences.

To simplify and enhance this process, this project integrates Gemini AI**,** a state-of-the-art generative language model developed by Google. Gemini AI is capable of understanding natural language and directly extracting meaningful relationships from raw text. It interprets input in human-like ways and generates structured relationships without relying on hard-coded linguistic rules. This shift from traditional NLP to prompt-based AI significantly improves accuracy, scalability, and ease of development.

The introduction of Gemini AI marks a transformative step in automating the generation of knowledge graphs, enabling deeper insights from data and reducing the complexity previously associated with NLP pipelines. that accepts raw text or PDF input, uses Gemini AI to identify relationships and entities, and stores this structured information in a Neo4j **graph database**. This enables users to query and explore data efficiently, even when it comes from unorganized or large volumes of sources. The final system is scalable, easy to use, and applicable in various fields like healthcare, education, law, and business analytics.

1.1 Motivation

In the modern digital era, we are constantly surrounded by an overwhelming amount of data in the form of websites, blogs, emails, social media posts, and scientific documents. Unlike structured databases, which are neatly organized in rows and columns, this data is unstructured

and lacks a fixed format. While humans can easily understand such text, computers struggle to interpret and derive meaning from it.

This vast unstructured data contains valuable knowledge that can drive insights, discoveries, and smarter decision-making. However, manually analyzing large volumes of data is time-consuming and often impractical. This motivates the need for an intelligent, automated system that can interpret unstructured text and represent the knowledge in a usable format. Knowledge graphs fulfill this requirement by visually linking entities and relationships. With the use of advanced AI models like Gemini AI, we can now automate the creation of these graphs by extracting meaningful connections from natural language, making the entire process faster, more accurate, and scalable.

1.2 Problem Statement

Despite the abundance of valuable information in unstructured data formats, there is no straightforward or automated method to convert this information into structured formats that machines can analyze. Existing systems using manual NLP techniques are often inefficient, error-prone, and not scalable for large datasets. There is a pressing need for an automated solution that can accurately extract entities and their relationships and represent them in a structured form such as a knowledge graph. This project addresses this gap by leveraging Gemini AI for intelligent extraction and Neo4j for structured storage.

1.3 Objective of Project

The key objectives of this project are to design a system that can intelligently process unstructured textual data and generate a usable knowledge graph for users. To achieve this, the system must meet several goals:

- Automatically extract entities and their relationships using Gemini AI without requiring manual rule sets.
- Support both plain text and PDF formats for input.
- Store relationships and entities as nodes and edges in Neo4j.
- Enable querying of the graph using Cypher language.
- Offer a user-friendly way to visualize connections and explore information.
- Ensure the system is scalable so it can process large documents or batches of files.
- Maintain a modular design to allow future upgrades and customizations.

By meeting these objectives, the system provides a robust solution to transform unstructured content into an intelligent, interactive structure of knowledge.

1.4 Scope of Project

The scope of this project includes the design, development, and testing of a complete pipeline that transforms unstructured English text into a structured knowledge graph. It supports input from raw text and PDF documents, which are then processed using Gemini AI for relationship extraction. The structured output is stored in Neo4j, a graph database, and visualized using graph interfaces.

Currently, the system handles only English-language inputs and is focused on extracting people, organizations, locations, and date-based relationships. It is built with flexibility in mind and has the potential for future expansion. For instance, support for additional languages, real-time web scraping, and integration with external knowledge bases like Wikidata or DBpedia can be added. The system is designed to be domain-independent and can be used in fields like healthcare, business intelligence, law, and education.

1.4 Limitations of Project

Despite the strengths of this system, there are several limitations that should be considered for future improvement:

- The system currently supports only English-language input, which restricts its usability for multilingual datasets or global users.
- An active internet connection is required to use the Gemini API, as the AI model is hosted remotely and not available offline.
- The quality and accuracy of extracted relationships depend on Gemini AI's generative output. Occasionally, the model may miss certain relationships or generate incorrect connections, especially if the input text is vague or ambiguous.
- The model does not validate factual correctness—it extracts relationships based on textual cues, even if the content is inaccurate or misleading.
- Real-time or large-scale streaming data processing is not yet implemented, which may limit performance in high-throughput environments.

- The system currently lacks a detailed error-handling mechanism for malformed inputs, API rate limits, or unexpected outputs from Gemini.

These limitations provide valuable directions for future development and enhancement of the system. Only supports English

- Requires an internet connection to access Gemini API
- Dependent on Gemini's generative accuracy

2. LITERATURE SURVEY

The development of knowledge graphs from unstructured data has been an active area of research, with many systems relying on traditional Natural Language Processing (NLP) techniques. In earlier approaches, libraries like SpaCy and Stanford NLP were used to perform tasks such as tokenization, named entity recognition, dependency parsing, and relationship extraction. These pipelines were highly customizable but required deep expertise in linguistics and programming. Moreover, they were rule-dependent, limiting their adaptability and scalability across various domains and languages.

Recent advancements in artificial intelligence, especially in the field of large language models (LLMs), have transformed the way unstructured text is processed. Models like GPT-3, BERT, and Gemini AI can understand and generate human-like language, making them suitable for complex NLP tasks with minimal manual configuration. These models use deep learning techniques trained on vast datasets and are capable of performing entity recognition, sentiment analysis, and relationship extraction from plain text with high accuracy.

Among these, Gemini AI stands out for its ability to handle long context and generate structured outputs based on prompts. Unlike rule-based systems, Gemini requires only a clear instruction to identify entities and relationships, eliminating the need for handcrafted NLP rules. Its integration into this project signifies a shift from traditional tools to AI-driven, prompt-based information extraction.

The use of Neo4j as the backend database is also supported by several previous studies, where graph-based data models have proven effective for visualizing complex relationships between entities. Neo4j offers fast querying, easy visualization, and scalability, making it a strong fit for storing the output of LLM-based relationship extractors.

In the development of the project *"Knowledge Graph from Unstructured Data using Gemini and Neo4j"*, several journal references have provided foundational insights that guided both the conceptual framework and technical implementation. One of the most influential works is by J. Devlin et al. on BERT: *Pre-Training of Deep Bidirectional Transformers for Language Understanding* [3]. Although the current system uses Gemini 1.5 Pro, its architecture builds upon the transformer-based models pioneered by BERT. This reference is crucial, as it introduces the deep contextual language understanding necessary for extracting semantic

relationships in the form of triples (Subject → Relation → Object) from unstructured text—an essential part of the knowledge graph creation pipeline.

Further, the research by M. Abdul-Mageed and L. Ungar titled *EmoNet: Fine-Grained Emotion Detection* [4] provides important insights into how deep neural networks can be employed for emotion classification from textual input. While the present system focuses on factual knowledge extraction, the methodologies described in this paper are highly relevant if emotional context or sentiment is to be embedded within graph nodes or used for sentiment-aware querying in future enhancements.

Another pivotal reference is the work by R. Cowie et al., *Emotion Recognition in Human-Computer Interaction* [1], which emphasizes the importance of recognizing user intent and emotion in language processing. This concept aligns well with the broader goal of your project, especially if it evolves into a natural language interface or chatbot system built on top of the knowledge graph. Understanding contextual tone—whether emotional, factual, or subjective—can significantly enhance the system's ability to generate meaningful and user-relevant responses.

Together, these references [1], [3], and [4] serve as theoretical and methodological pillars for the project, underpinning the use of large language models, the importance of contextual emotion in human-computer interaction, and the future scope of graph enrichment through sentiment analysis.

2.1 Existing System

The existing systems for building knowledge graphs from unstructured data primarily rely on traditional NLP libraries like SpaCy, Stanford NLP, and OpenNLP. These tools use rule-based pipelines for tasks such as tokenization, named entity recognition (NER), part-of-speech (POS) tagging, and syntactic parsing. While these methods offer flexibility and control, they require extensive manual setup, language-specific tuning, and domain expertise. They often fail to handle the ambiguity and complexity of natural language efficiently, particularly in large-scale or domain-diverse environments.

Moreover, integrating these tools with downstream components like graph databases involves significant engineering effort and lacks adaptability when the input format or domain changes.

2.2 Proposed System

The proposed system simplifies and modernizes the approach by using Gemini AI, a generative language model developed by Google, to handle the entire NLP pipeline through prompt-based interaction. Gemini AI is capable of understanding raw text inputs and generating structured relationships directly from natural language. This eliminates the need for separate modules for tokenization, NER, and relationship extraction. The system takes input as text or PDF, extracts entities and their relationships using Gemini, and stores the results in a Neo4j graph database.

This AI-driven approach reduces development complexity and increases scalability, accuracy, and ease of use. By removing the dependence on handcrafted rules and feature engineering, the proposed system allows for rapid deployment across multiple domains with minimal reconfiguration.

The proposed system uses Gemini AI for extracting structured relationships from unstructured text. Unlike traditional NLP pipelines, Gemini handles tokenization, entity detection, and relationship extraction in one go using prompt-based instructions. It outputs direct relationships which are stored in Neo4j.

3. SYSTEM ANALYSIS

3.1 Functional Requirements

Functional requirements describe the core functions and features that the system must provide to fulfill its objectives. These are the tasks the system should be able to perform from the user's and system's perspective.

- **Input Handling**: The system must allow users to input data either as raw text or by uploading PDF documents. This includes validating file types and managing file storage temporarily.
- **Text Extraction**: For PDF inputs, the system should extract text content using tools like PyPDF2. For text inputs, it should sanitize and clean the data before processing.
- **Entity & Relationship Extraction**: The system should use Gemini AI to analyze the input text and extract relationships between entities. This process should be accurate, efficient, and scalable.
- **Graph Construction**: The system must take the relationships output by Gemini AI and convert them into a knowledge graph format. It should create or update nodes and relationships in a Neo4j database using Cypher queries.
- **Query & Visualization**: Users should be able to query the knowledge graph using Cypher language and view the results in a visual graph interface provided by Neo4j Browser or any integrated visualization tool.
- **User Interaction**: The system should have a simple web interface (via Flask) to accept input, trigger processing, and display results.

These functional requirements ensure that the system fulfills its main objective of transforming unstructured data into a structured, queryable knowledge graph.

3.2 Non-Functional Requirements

Non-functional requirements define how the system performs rather than what it does. These are critical to ensure the system’s quality, usability, and long-term success.

- **Performance**: The system should process uploaded documents and return extracted relationships within a reasonable response time (e.g., under a few seconds for short texts and within a minute for longer PDFs).

- **Scalability**: The system should be capable of handling large-scale datasets and concurrent user requests without significant degradation in performance. It should allow for easy integration with distributed processing or cloud deployment in the future.
- **Usability**: The user interface should be intuitive, easy to use, and responsive, allowing both technical and non-technical users to interact with the system effectively.
- **Portability**: The system should be deployable on different environments, including local machines, cloud platforms, or containerized setups using Docker.
- **Reliability**: The application should handle failures gracefully, such as invalid file formats, API connection issues, or unexpected input formats, and notify the user with clear error messages.
- **Security**: The system should secure user data and API keys. Input validation, secure file handling, and proper access control must be implemented to prevent unauthorized access or injection attacks.
- **Maintainability**: The codebase should be modular and well-documented to support future development and debugging efforts.
- **Availability**: The system should aim for high availability, especially if hosted online, ensuring minimal downtime during operation.

These non-functional aspects ensure the system is efficient, reliable, and ready for real-world application beyond the development phase.

3.3 Software Requirements

Operating System:

- Linux (Ubuntu or similar) is recommended for production, though it will work on Windows and macOS as well.
- Ensure the OS has Python 3.x installed.

Python 3.x:

Version 3.6 or higher is required for running the Flask application, along with required libraries.

Required Python Libraries: You can install the necessary libraries via `pip`:

- Flask: `pip install Flask`

- PyPDF2: `pip install PyPDF2`
- `google-generativeai` (Gemini AI SDK): `pip install google-generativeai`
- Neo4j Python Driver: `pip install neo4j`
- Werkzeug (for handling file uploads): `pip install werkzeug`

Neo4j Database:

Neo4j (Community or Enterprise Edition) should be installed and running.

- Download Neo4j from the official Neo4j website.
- Neo4j should be configured to run locally or remotely, depending on your architecture.
- Make sure the Neo4j server is running on port `7687` (the default for the Bolt protocol).

Gemini AI API Key:

You need to sign up and configure access to Google's Gemini AI API. After obtaining the API key, ensure that it's properly configured in your application (`GEMINI_API_KEY`).

PDF Viewer (Optional):

If you need to manually check or view PDFs, any PDF viewer (like Adobe Acrobat Reader) will work.

3.4 Hardware Requirements

Processor:

- Minimum: Dual-core processor (e.g., Intel Core i3 or equivalent)
- Recommended: Quad-core processor (e.g., Intel Core i5 or higher)

RAM:

- Minimum: 4 GB
- Recommended: 8 GB or higher, especially if you plan to process large files or use intensive AI models like Gemini.

Storage:

- Minimum: 10 GB of free disk space
- Recommended: SSD for faster data read/write speeds, especially for handling large documents and AI model interactions.

Network:

Reliable internet connection to interact with the Gemini AI API and Neo4j cloud services (if applicable).

Graph Database Server (Neo4j):

If Neo4j is running on a separate machine or server, ensure it meets the necessary requirements (e.g., CPU, RAM, and disk space) for efficient graph data storage and querying.

Network and Firewall Requirements:

1. **Firewall Configuration**:
   - Make sure ports 5000 (for Flask app) and 7687 (for Neo4j) are open for external access if required.
   - If Neo4j is hosted on a remote server, ensure that it's accessible via the network.
2. **Gemini AI API Access**:
   - The app requires outbound HTTP/HTTPS access to the Gemini API endpoint, so ensure no network restrictions (e.g., firewalls or proxies) block the communication.

4. SYSTEM DESIGN

The System Design of the project involves several key modules that work together to process and extract knowledge from unstructured data. The Data Input Module handles user input, either through file uploads (such as PDFs) or direct text input via a web form. After the input is received, the Text Extraction Module cleans and extracts text from files like PDFs, ensuring it's in a suitable format for further processing. The extracted text is then passed to the Gemini Analysis Module, which utilizes Gemini AI's generative capabilities to identify entities (like people, places, dates) and their relationships (such as "born in," "married to"). These relationships are structured as triples and are sent to the Graph Storage Module**,** where they are stored in a Neo4j graph database using Cypher queries. The Frontend Module provides a user-friendly interface for input submission, displaying the extracted triples or error messages in a readable format. The entire process follows a clear sequence: user input is extracted and cleaned, relationships are identified by Gemini AI, and the results are stored in Neo4j, after which the user receives feedback. The system design emphasizes scalability, security, and performance to handle large datasets and provide a smooth user experience, with careful error handling at each stage to ensure reliability.

4.1 UML Diagrams

UML, which stands for Unified Modelling Language, is a way to visually represent the architecture, design, and implementation of complex software systems. Looking at code and understanding the system is difficult, instead I can use UML diagrams to understand the system. UML diagrams can be used to explain the components of the software to people who don't have technical knowledge. It is a standard language for specifying, visualizing, constructing, and documenting the artefacts of the software systems. UML is different from other common programming languages like C++, Java, and COBOL etc. It is pictorial language used to make software blueprints.

4.1.1 Data Flow Diagram

The Data Flow Diagram (DFD) for the Knowledge Graph from Unstructured Data system illustrates how data moves through the system, from the user's input to the final output. The process begins with the User, who interacts with the Data Input Module by either uploading a file (such as a PDF) or pasting direct text. The Data Input Module forwards this input to the

Text Extraction Module, which processes the raw data, especially for PDFs, extracting and cleaning the text for further analysis. Once the text is ready, the Gemini Analysis Module takes over, using Gemini AI to analyze the text and extract entities (like people, locations, and dates) along with their relationships (e.g., "born in," "married to"). These extracted triples are then sent to the Graph Storage Module, where they are stored in a Neo4j graph database, ensuring that entities are saved as nodes and their relationships as edges. Finally, the Frontend Module retrieves the stored data from the Neo4j database and displays it to the user, showing the extracted triples in a readable format or providing error messages if something goes wrong. This entire flow enables the system to process unstructured data, extract meaningful knowledge, and present it to the user, ensuring a seamless interaction between components.

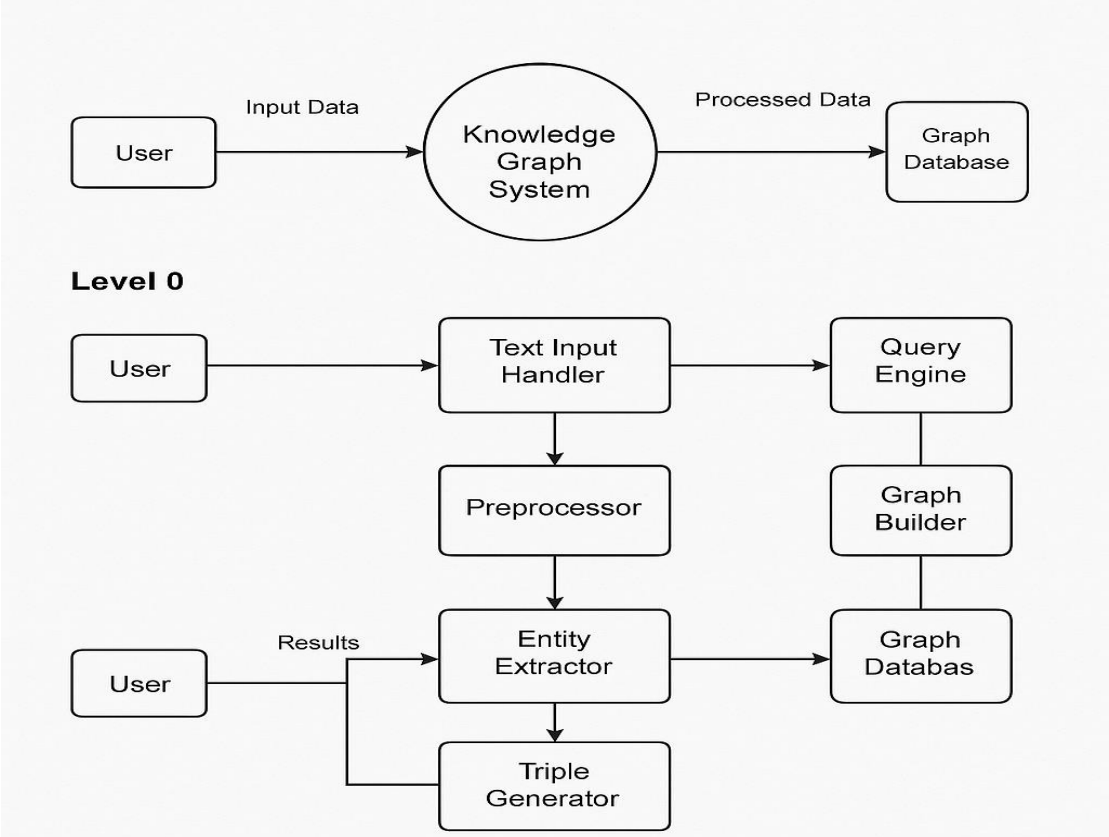

Fig 4.1: Data flow diagram - 01

The diagram illustrates the architecture and workflow of a Knowledge Graph System designed to transform unstructured user input into structured knowledge stored in a graph database. The system begins with user-provided data, which is processed through multiple layers before being stored and made available for queries.

At the core (Level 0), the system consists of several key components. The Text Input Handler receives and normalizes raw input (e.g., text or PDFs), ensuring it is ready for further processing. The Preprocessor then refines this data through steps like tokenization or sentence segmentation. Next, the Entity Extractor identifies and classifies entities (such as people or organizations), while the Triple Generator detects relationships between these entities, structuring them into subject-predicate-object triples (e.g., "Marie Curie-discovered-Radium"). The Graph Builder converts these triples into a graph format, defining nodes (entities) and edges (relationships) for storage. Meanwhile, the Query Engine supports both system operations and user queries against the knowledge graph.

The processed output is stored in a Graph Database (e.g., Neo4j), where the structured knowledge becomes accessible. Users can then retrieve insights through queries or visualize the connections. This pipeline effectively bridges unstructured data with a queryable, graph-based representation, enabling advanced analysis and discovery of relationships within the input content.

4.1.2 Use Case Diagram

A use case diagram in the Unified Modeling Language (UML) is a type of behavioral diagram defined by and created from a Use-case analysis. Its purpose is topresent a graphical overview of the functionality provided by a system in terms of actors, their goals (represented as use cases), and any dependencies between those usecases. The main purpose of a use case diagram is to show what system functions are performed for which actor. Roles of the actors in the system can be depicted.

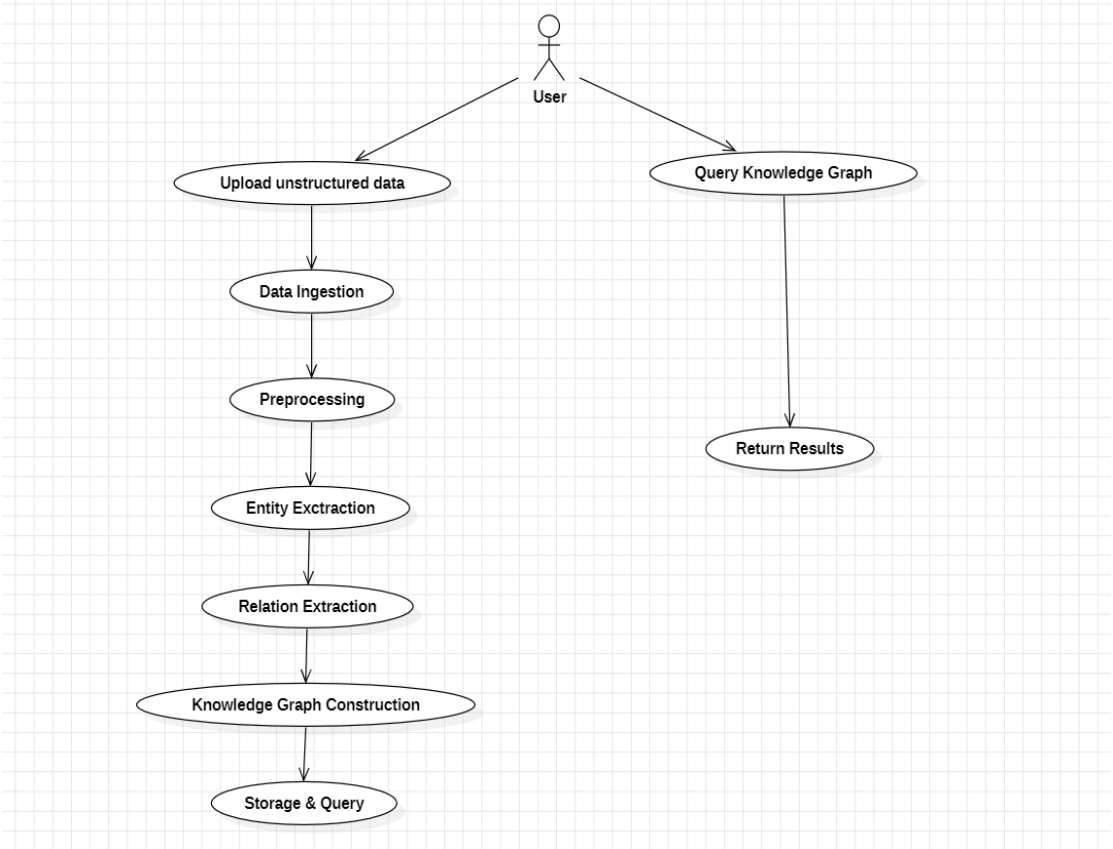

Fig 4.2: Use case diagram

When integrating Gemini AI into the knowledge graph workflow, the system becomes significantly more powerful and efficient. The process begins as usual with the user uploading unstructured data, which may include not just text but also images, documents, or even audio files. Thanks to Gemini's multimodal capabilities, it can intelligently understand and process this wide range of input types. During data ingestion, Gemini AI helps identify and organize the different data formats automatically, ensuring that all information is properly prepared for analysis. In the preprocessing stage, Gemini applies advanced natural language processing techniques to clean, normalize, and structure the data, handling tasks such as language detection, summarization, and noise filtering.

In the entity extraction phase, Gemini's deep understanding of language and visual content enables it to accurately recognize a wide variety of entities, including people, organizations, locations, dates, and more, even from images and audio transcriptions. This makes the system highly adaptable to real-world, multimodal data. Next, during relation extraction, Gemini can analyze the context in which these entities appear and identify meaningful relationships between them using its powerful semantic reasoning. These entities and their relationships are then used to construct a knowledge graph, forming a structured

network of connected information. Finally, the knowledge graph is stored for querying. When a user queries the graph, Gemini can interpret complex questions, understand user intent, and return precise results by navigating the graph intelligently. This integration of Gemini AI enhances the entire pipeline, making it more accurate, flexible, and capable of handling rich, real-world data scenarios.

4.1.3 Class Diagram

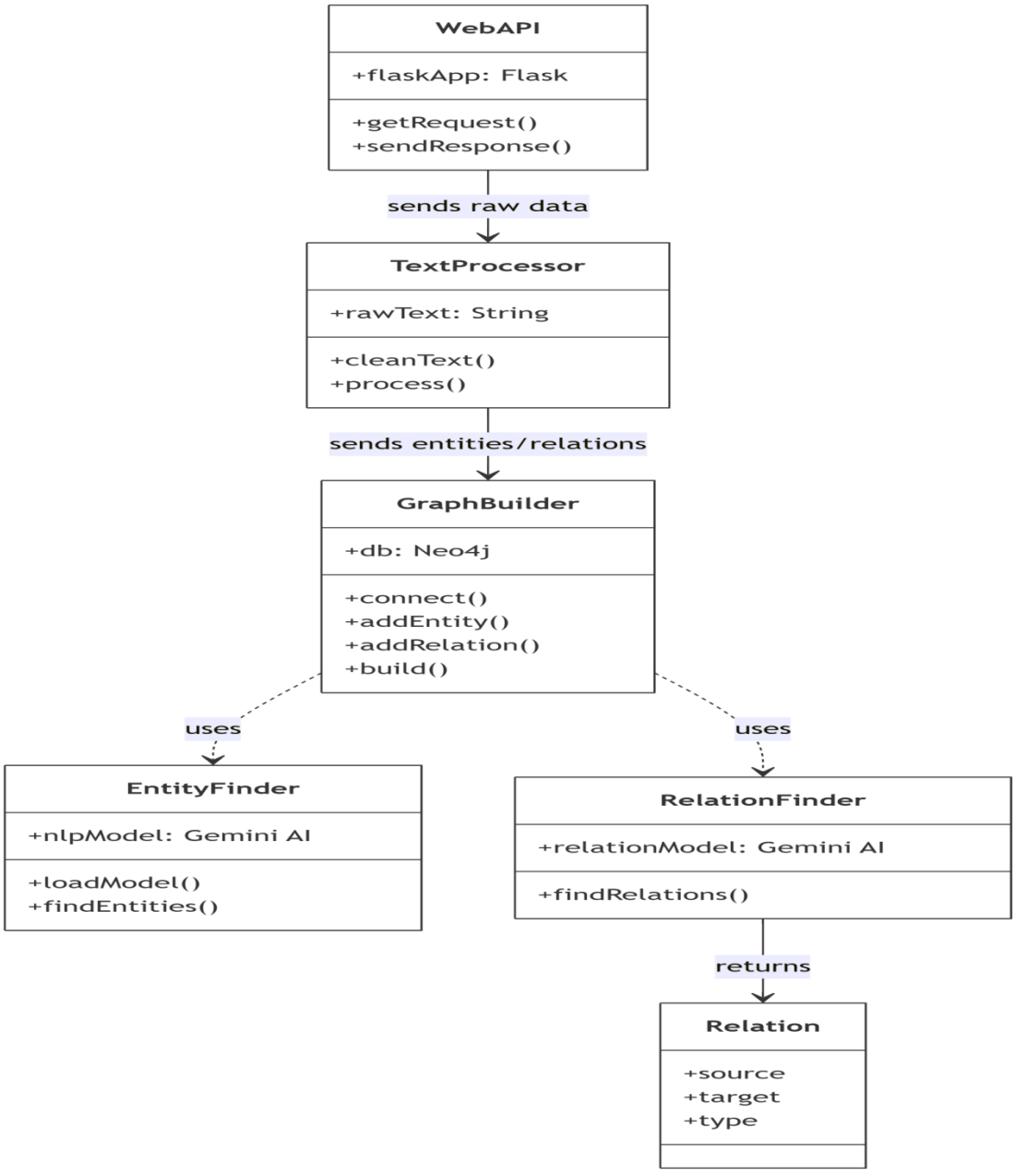

Fig 4.3: Class diagram

This diagram represents a class-based architectural workflow for a system that constructs a knowledge graph using Gemini AI and Neo4j, accessed through a Flask-based Web API. The process begins with the WebAPI class, which uses Flask to handle incoming HTTP requests and responses. It receives raw data from users through the getRequest() method and sends it forward via sendResponse(). This raw data is passed to the TextProcessor class, where it is stored as a string attribute (rawText) and then cleaned and processed using the cleanText() and process() methods. The processed data, now in the form of recognized entities and potential relationships, is sent to the GraphBuilder class.

The GraphBuilder is responsible for connecting to a Neo4j graph database and contains methods to connect(), addEntity(), addRelation(), and build() the knowledge graph. To perform its tasks, GraphBuilder relies on two helper classes: EntityFinder and RelationFinder. Both of these components utilize Gemini AI models. The EntityFinder uses a nlpModel powered by Gemini AI to load models (loadModel()) and extract named entities (findEntities()). Similarly, the RelationFinder uses a relationModel, also based on Gemini AI, to identify semantic relationships in the text via findRelations(). Once relationships are identified, they are returned in the form of a Relation object that contains attributes such as the source entity, the target entity, and the type of relationship. Overall, this system efficiently extracts knowledge from raw text, processes it using AI, and stores it in a structured graph format for querying and analysis.

4.1.4 Sequence Diagram

A sequence diagram in Unified Modelling Language (UML) is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart. Sequence diagrams are sometimes called event diagrams, event scenarios, and timing diagrams.

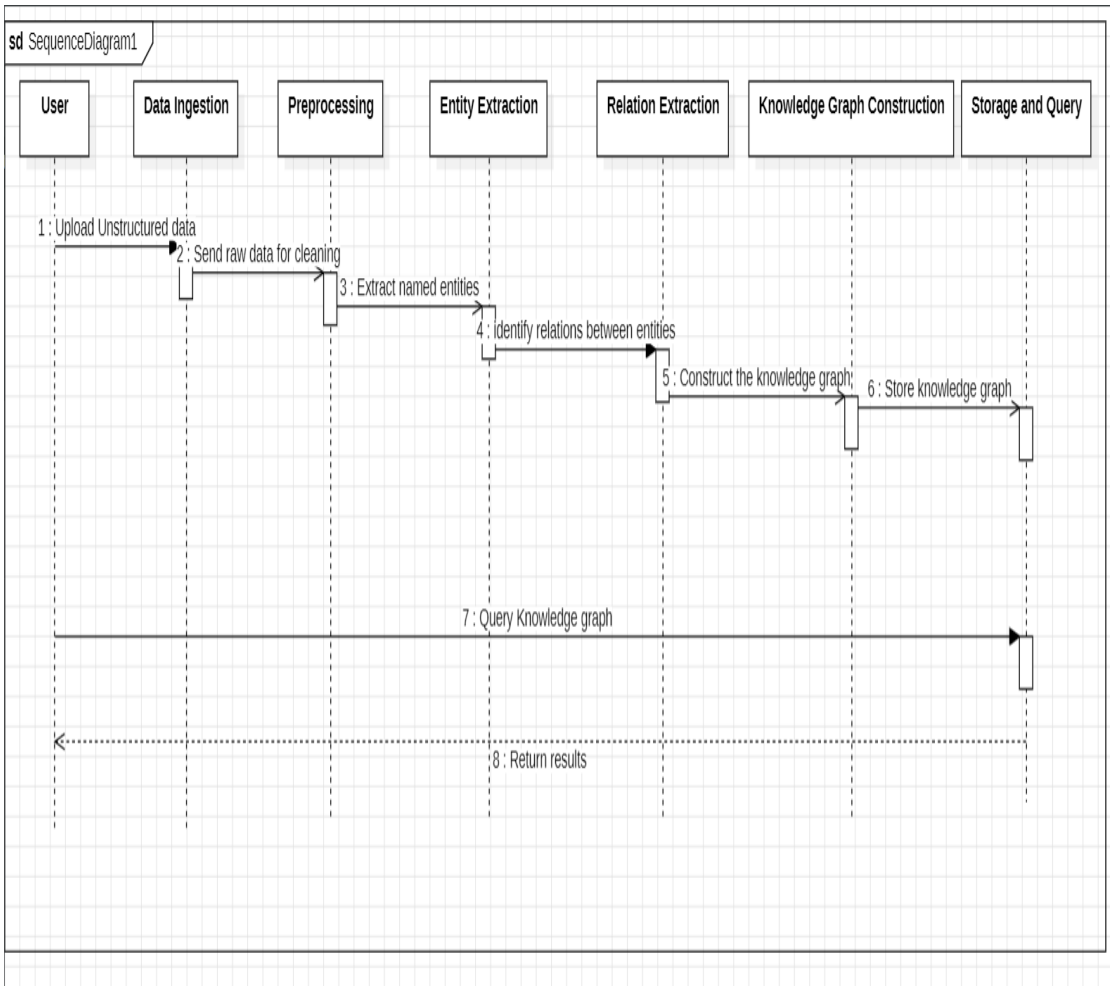

Fig 4.4: Sequence diagram

The sequence diagram demonstrates the order of operations when a user interacts with the system. It starts with the user submitting unstructured data. The system first calls the TextProcessor to clean and format the text. Then, the EntityExtractor is used to identify entities like people, places, or organizations. These extracted entities are passed to the GraphBuilder, which creates a visual representation of the data in the form of a graph. The Neo4jConnector then stores this graph in a graph database. Finally, the user receives a visualized graph interface, where they can view and interact with the knowledge graph. This diagram helps understand how the components communicate and process information in a step-by-step manner.

4.1.5 Activity Diagram

Activity diagrams are graphical representations of workflows of stepwise activitiesand actions with support for choice, iteration and concurrency. In the Unified ModelingLanguage, activity diagrams can be used to describe the business and operational step-by-step workflows of components in a system. An activity diagram shows the overallflow of control

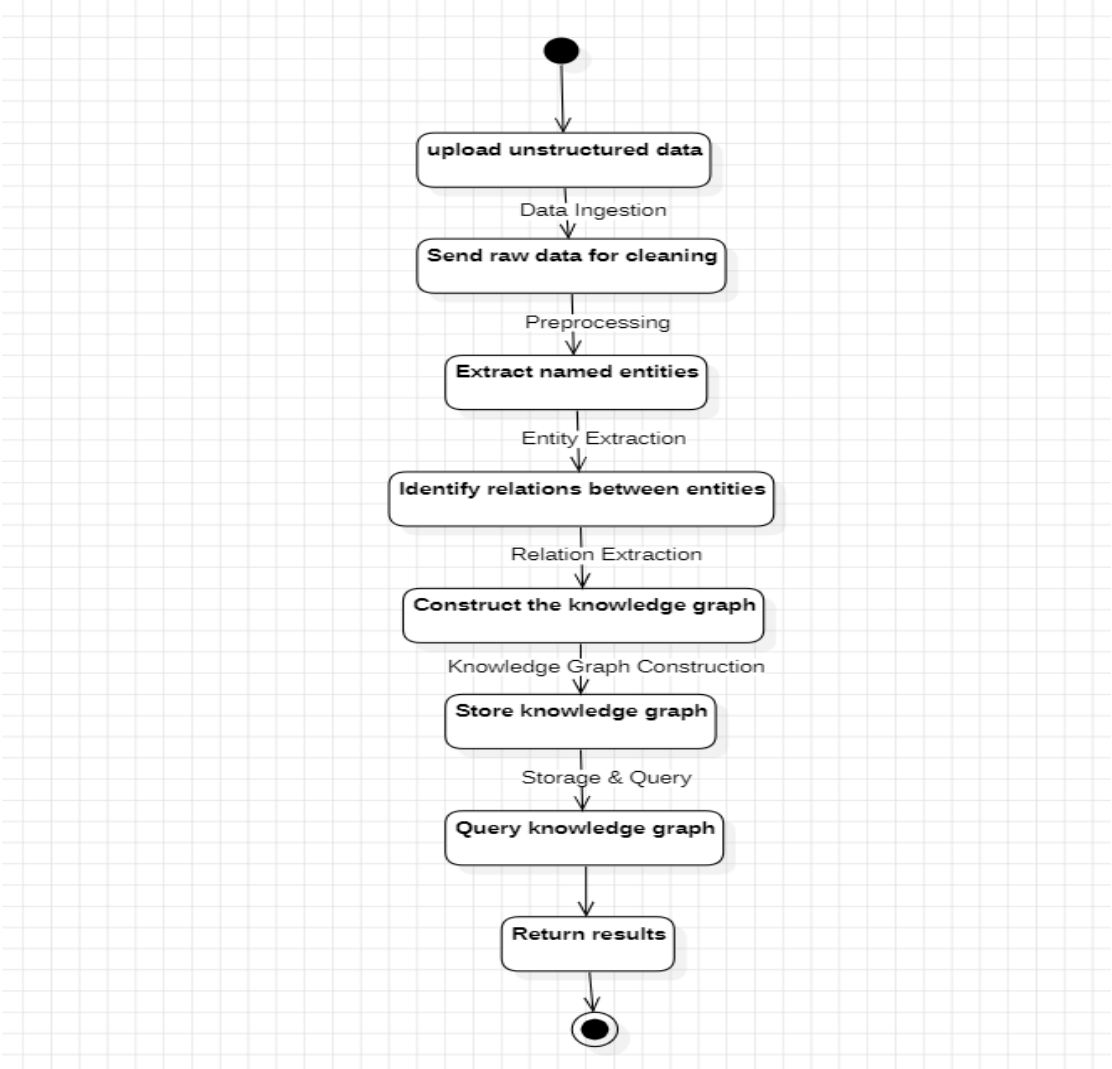

Fig 4.5: Sequence diagram

The activity diagram provides a high-level overview of the workflow involved in building a knowledge graph from unstructured data. It begins when the user provides input to the system. This data then goes through a preprocessing step where irrelevant information is removed and text is standardized. The cleaned text is passed through the NLP engine to extract named entities, which are linked and formed into triples representing relationships. These triples are then transformed into nodes and edges of a graph using Neo4j. After construction, the graph

is stored and becomes available for querying and visualization. This diagram gives a complete picture of how the system functions from start to finish, making it easier to understand the logic behind the automation process.

4.1.6 Statechart Diagram

The Statechart Diagram also known as a State Diagram —represents the different **states** the system or a particular component goes through during its operation, based on user actions or internal transitions.

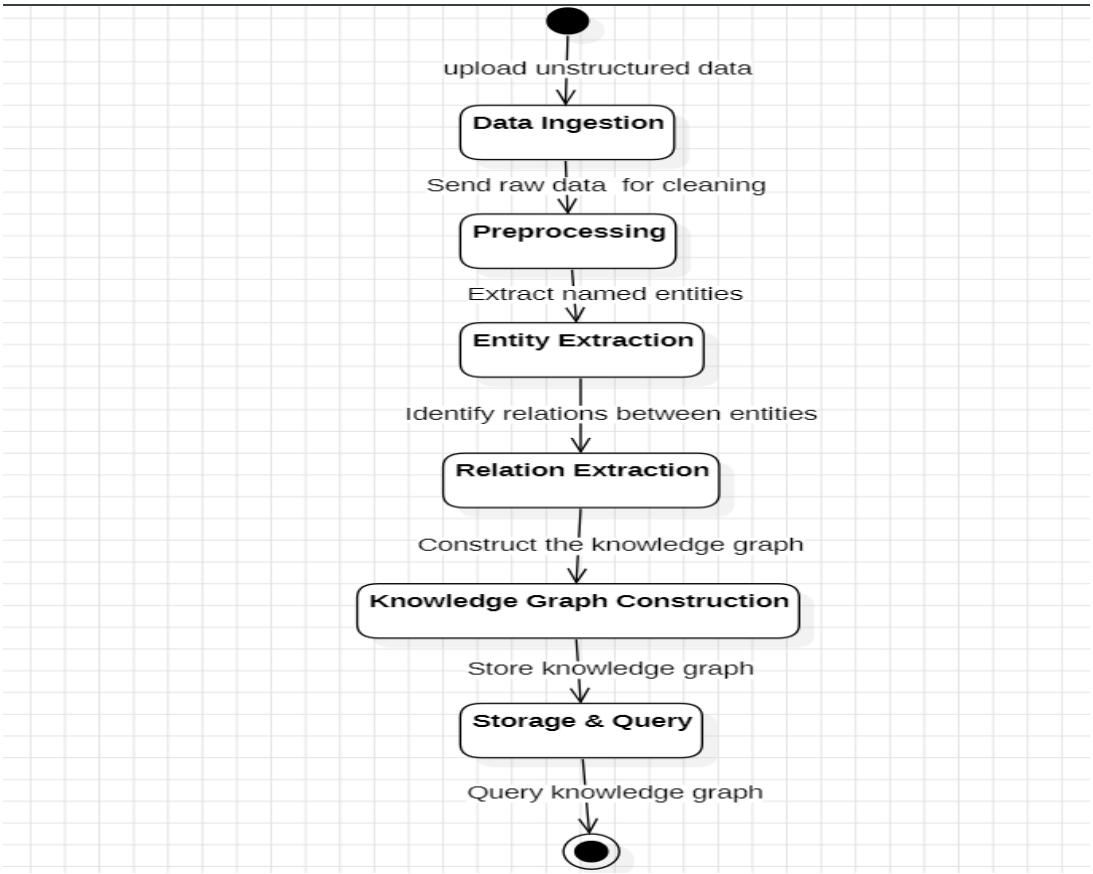

Fig 4.6: Statechart diagram

It begins with the Start state, where the system is idle, waiting for user input. Once the user submits the data, the system transitions to the Preprocessing state. In this phase, the input text is cleaned and normalized — things like punctuation are removed, and stopwords are filtered out.

Next, the system enters the Entity Extraction state, where named entities such as people, places, or organizations are identified using NLP tools. After extraction, it moves to the Triple Generation state, where subject-predicate-object relationships (triples) are formed to structure the data. These triples are then converted into nodes and edges in the Graph Building state.

Once the graph is created, the system reaches the Storage state, where all information is saved into the Neo4j database. Finally, the system transitions into the Query and Visualization state, where users can search the graph using Cypher queries and view the results through an interactive interface. The system then returns to the idle or Ready state, waiting for the next input.

5.MODULES

1. Data Input Module

The Data Input Module is the first point of interaction between the user and the system. It enables users to submit unstructured data in various formats, such as directly typed text, .txt files, or .pdf documents. This module is built using the Flask web framework, which handles form submissions through HTTP requests. Uploaded files are stored temporarily in a secure directory until they are processed. This module ensures that the input is correctly received and ready for extraction and analysis.

2. Text Extraction and Cleaning Module

Once the input data is received, the Text Extraction and Cleaning Module processes it to prepare for natural language analysis. If the file is a PDF, the PyPDF2 library is used to extract raw text from each page. Regardless of the file type, the extracted content is then cleaned using regular expressions (re) to remove unnecessary line breaks, special characters, and extra spaces. This step standardizes the input and ensures the text is clean, concise, and optimized for NLP processing by the Gemini AI model.

3. Gemini NLP Analysis Module

The Gemini NLP Analysis Module forms the core intelligence of the system. It utilizes the google.generativeai SDK to interact with Google's Gemini AI model. Cleaned text is sent to Gemini with a prompt that instructs the model to extract semantic relationships in a structured format like "entity1 → relationship → entity2." The generative capabilities of Gemini allow it to understand the context of the text and return meaningful relationship triples. These triples represent real-world connections, such as *"Mahatma Gandhi → born in → Porbandar"*, which are then passed to the storage module.

4. Graph Storage Module

The Graph Storage Module is responsible for storing the extracted entities and relationships in a Neo4j graph database. Using the Neo4j Python driver and Cypher query language, this module takes each triple and creates nodes for entities and edges for relationships. The MERGE command is used to ensure that duplicate entries are not created, maintaining a clean

and consistent knowledge graph. This graph structure allows for powerful queries and visualizations, helping users explore complex connections between entities.

5. Web Interface Module

The Web Interface Module provides a simple, intuitive front end built using Flask and HTML templates. It allows users to upload files or input text, and then displays a preview of the cleaned input along with the extracted relationships. This module also handles validation, such as checking file types and input length, and displays appropriate error or success messages. It serves as the main interaction point for users and ensures that the system is easy to use even for non-technical users.

6. Error Handling and Logging Module

The Error Handling and Logging Module enhances the reliability of the system by catching and managing exceptions that may occur during processing. Whether it's a failure in reading a PDF, a timeout from the Gemini API, or a database connection error, this module ensures that the user receives a helpful message instead of a crash. It also logs technical issues internally for developers, aiding in debugging and future improvements. This makes the system robust and user-friendly under various conditions.

6.IMPLEMENTATION RESULTS

6.1 Project Implementation

The Implementation Phase is where the actual coding and development of the system take place. Initially, the necessary development tools and frameworks, such as Python, Flask, PyPDF2, Gemini SDK, and Neo4j, are set up. The first step in the implementation is to build the web interface using Flask**,** allowing users to upload files (PDFs or text) or paste text directly. After the user submits the data, the backend processes it: the Text Extraction Module handles extracting text from PDFs using PyPDF2, ensuring it's clean and ready for analysis. The extracted text is then passed to the Gemini Analysis Module, where Gemini AI is used to identify entities and relationships in the text. These entities and relationships are stored as triples in the Neo4j graph database using Cypher queries to ensure efficient data storage. The Frontend Module then retrieves this information and displays it in a readable format to the user. Throughout the process, error handling is implemented to ensure the system can handle issues such as invalid file formats, missing input, or API failures. After the system is coded, it is tested to ensure everything works as expected, including unit testing of individual modules and integration testing of the entire system. Once the system passes testing, it is deployed on platforms like Heroku or AWS for easy access. Finally, detailed documentation is created for users, explaining how to use the system, and for developers, explaining the system's architecture and code.

6.2 Technologies Used

1.Python3:

Python is a high-level, general-purpose programming language renowned for its simplicity and powerful ecosystem. It is especially popular in the fields of machine learning, data processing, and web development. In this project, Python acts as the backbone of the system—managing file uploads, extracting text, communicating with the Gemini AI API, handling logic for parsing relationships, and integrating with the Neo4j graph database. Python’s readable syntax and rich library support make it ideal for building modular, maintainable systems like this one.

2.Flask(WebFramework):

Flask is a lightweight web framework written in Python, ideal for small to medium-sized web applications. It is used in this project to create a user-friendly interface where users can submit raw text or upload files. Flask handles HTTP requests, form submissions, and rendering HTML templates. It connects the frontend with the backend processing logic that includes

Gemini API integration and graph database storage. Thanks to Flask's simplicity, the system remains flexible and easily extendable in the future.

3.PyPDF2(PDFTextExtraction):

PyPDF2 is a Python library for reading and extracting text content from PDF files. Since many valuable documents such as reports, articles, and biographies are in PDF format, this tool is used to parse those files into clean, readable text. The extracted text serves as input for the NLP engine (Gemini AI). PyPDF2 is lightweight and effective for multi-page PDF processing and ensures users can input rich, real-world data into the knowledge graph pipeline.

4.GoogleGenerativeAI(GeminiSDK):

The core innovation in this project comes from Gemini AI—a state-of-the-art generative large language model developed by Google. Accessed via the google.generativeai Python SDK, Gemini is capable of understanding complex language prompts and generating human-like structured output. In this project, Gemini is prompted with domain-specific instructions to extract relationships in the form of "entity1 --> relation --> entity2" triples. Its contextual understanding far surpasses traditional NLP tools like SpaCy, allowing it to extract nuanced personal, geographical, and temporal relationships without manual training or hard-coded rules.

5.Neo4j(GraphDatabase):

Neo4j is a leading graph-based NoSQL database designed to store data as a network of nodes and edges. In this project, Neo4j is used to store extracted entities (as nodes) and their relationships (as edges). This structure allows for intuitive querying and visualization of knowledge using the Cypher query language. Neo4j's support for relationship-centric data models makes it ideal for semantic networks, where understanding how entities relate is just as important as the entities themselves. The use of MERGE queries ensures that duplicate nodes and edges are avoided, keeping the graph clean and consistent.

Natural Language Processing (NLP)

Natural Language Processing (NLP) is a subfield of Artificial Intelligence (AI) that focuses on enabling computers to understand, interpret, and generate human language. Unlike structured data in databases, human language is ambiguous, context-driven, and varies widely in expression. NLP bridges the gap between human communication and machine

understanding, allowing software to process and make sense of unstructured textual data such as documents, reviews, articles, and conversations.

NLP combines techniques from linguistics, computer science, and machine learning. It involves several tasks, such as:

- **Tokenization:** Breaking text into words or sentences.
- **Part-of-speech tagging:** Identifying grammatical roles (nouns, verbs, adjectives, etc.).
- **Named Entity Recognition (NER):** Detecting names of people, places, organizations, etc.
- **Dependency Parsing:** Understanding grammatical relationships between words.
- **Semantic Analysis:** Understanding the meaning and context behind words and phrases.

How NLP Is Used in This Project

1. Text Understanding with Gemini AI

Instead of using traditional rule-based NLP tools like SpaCy, your system now uses Gemini AI, a generative NLP model. Gemini reads entire blocks of unstructured text—like PDF articles or user-submitted input—and understands them in context. It goes beyond shallow keyword extraction by recognizing who is related to whom**,** what actions are involved**,** and how entities are connected**.**

Example:
"Mahatma Gandhi was born in Porbandar in 1869. He led the Indian independence movement and was married to Kasturba Gandhi."Gemini AI extracts:

Gemini AI (using Natural Language Processing) reads this text and understands it like a human would. Then it extracts key relationships**:**

Output Extracted by Gemini:

- Mahatma Gandhi → born in → Porbandar
- Mahatma Gandhi → birth year → 1869
- Mahatma Gandhi → led → Indian independence movement
- Mahatma Gandhi → married to → Kasturba Gandhi

These facts are not just random lines—they are structured triples (entity → relationship → entity), which your system turns into a Knowledge Graph**.**

2. Prompt-Based Relationship Extraction

Traditional NLP systems require rule engineering or training custom models. Gemini simplifies this through prompt engineering: you describe what you want in plain language, like:

"Extract all entity relationships in the format: entity1 → relation → entity2"

The AI uses its language understanding to respond with structured results. This is a new paradigm of NLP where zero-shot learning (no prior examples or training) is possible.

3. Feeding Structured NLP Output into a Knowledge Graph

Once NLP (via Gemini) extracts these structured triples:

- Subject (e.g., Mahatma Gandhi)
- Predicate (e.g., born in)
- Object (e.g., Porbandar)

These relationships are stored as nodes and edges in Neo4j**,** a graph database. The result is a Knowledge Graph—a machine-readable representation of human knowledge extracted from natural language.

4. Real-World Use Cases Enabled by NLP

Because of NLP, your system can:

- Answer natural language questions like "Who founded Tesla?"
- Visualize interconnected data (people, events, organizations)
- Discover hidden patterns in large document collections

Why Use Generative NLP (Gemini) Instead of Traditional NLP (SpaCy)?

Table.6.1.Comparision between Gemini AI and Spacy

Traditional NLP (SpaCy)	Generative NLP (Gemini AI)
Rule-based or ML models need training	Zero-shot learning via prompts
Limited to known entity types (PERSON, ORG)	Understands custom and contextual relationships
Can struggle with complex sentences	Understands deeper semantics and context
Requires code for relationship logic	Extracts clean triples directly

6.3 Setup

Frontend

It provides a simple user interface where users can input text or upload a file (PDF or text) to extract relationships and build a knowledge graph using Neo4j.

1. Layout and Styling

- The <style> section defines how the webpage looks. It uses basic CSS to make the page clean, readable, and user-friendly.
- Elements like buttons, text areas, and messages are styled with padding, colors, and borders.

<style>

body { font-family: Arial, sans-serif; max-width: 800px; margin: 0 auto; padding: 20px; }

.container { display: flex; flex-direction: column; gap: 20px; }

.input-section { padding: 20px; border: 1px solid #ddd; border-radius: 5px; }

textarea { width: 100%; height: 150px; padding: 10px; margin-bottom: 10px; }

button { background-color: #4CAF50; color: white; padding: 10px 15px; border: none; border-radius: 4px; cursor: pointer; }

#results { margin-top: 20px; padding: 20px; border: 1px solid #ddd; border-radius: 5px; display: none; }

.relationship { padding: 8px; margin: 5px 0; background-color: #f0f0f0; border-radius: 3px; font-family: monospace; }

```
.status { padding: 10px; margin: 10px 0; border-radius: 4px; }

.success { background-color: #dff0d8; color: #3c763d; }

.error { background-color: #f2dede; color: #a94442; }

.info { background-color: #d9edf7; color: #31708f; }

</style>
```

2. User Interface (HTML Form)

- There's a **form** that allows the user to either:
  - Paste text into a text area (<textarea>), or
  - Upload a file (<input type="file">), which accepts .pdf and .txt files.
- A button is provided to submit the form and trigger the analysis.

```
<form id="inputForm" class="input-section">

<div>

<label for="textInput">Enter text:</label>

<textarea id="textInput" name="text" placeholder="Paste your text here..."></textarea>

</div>

<div>

<label for="fileInput">Or upload a file (PDF or TXT):</label>

<input type="file" id="fileInput" name="file" accept=".pdf,.txt">

</div>

<button type="submit">Process and Build Graph</button>

</form>
```

3. Result Display Section

- Once the backend processes the input, the results are shown in a separate section:
  - A preview of the processed text.
  - A list of relationships extracted by Gemini AI.
  - A status message that informs the user whether the operation was successful or if any errors occurred.

* *

* *

<h3>Processing Results</h3>

* *

<h4>Extracted Relationships:</h4>

* *

* *

4. JavaScript (Client-Side Logic)

- When the form is submitted, JavaScript code:
  1. **Prevents the default form behavior** (page reload).
  2. **Sends the form data** (text or file) to the backend using fetch() and a POST request to /analyze.
  3. **Waits for a response** from the server (which processes the input using Gemini and Neo4j).
  4. **Displays results**:
     - If successful, it shows the extracted relationships and a success message.
     - If no relationships are found, it shows an informative message.
     - If an error occurs (e.g., no input, server error), it displays an error message.

<script>

document.getElementById('inputForm').addEventListener('submit', async (e) => {

e.preventDefault();

const statusMessage = document.getElementById('statusMessage');

const resultsDiv = document.getElementById('results');

const textPreview = document.getElementById('textPreview');

const relationshipList = document.getElementById('relationshipList');

relationshipList.innerHTML = '';

resultsDiv.style.display = 'none';

statusMessage.textContent = '';

```
statusMessage.className = 'status';

const formData = new FormData(e.target);

try {

statusMessage.textContent = 'Processing...';

statusMessage.className = 'status info';

const response = await fetch('/analyze', {

method: 'POST',

body: formData

});

const contentType = response.headers.get('content-type');

if (!contentType || !contentType.includes('application/json')) {

const text = await response.text();

throw new Error(`Server returned: ${text.slice(0, 100)}...`);

}

const data = await response.json();

if (!response.ok) {

throw new Error(data.error || 'Request failed');

}

textPreview.innerHTML=`<p><strong>ProcessedText
Preview:</strong></p><p>${data.text_preview}</p>`;

if (data.success) {

if (data.relationships && data.relationships.length > 0) {

relationshipList.innerHTML = data.relationships.map(rel =>

`<div class="relationship">${rel}</div>`

).join('');

resultsDiv.style.display = 'block';

statusMessage.textContent = 'Success! Relationships extracted and stored in Neo4j.';
```

```
statusMessage.className = 'status success';
} else {
statusMessage.textContent = 'Analysis completed but no relationships were found.';
statusMessage.className = 'status info';}
} else {
throw new Error(data.error || 'Analysis failed');
}
} catch (error) {
statusMessage.textContent = `Error: ${error.message}`;
statusMessage.className = 'status error';
console.error('Error:', error);
}
});</script>
```

Backend

This Flask-based backend that allows users to submit unstructured data (either text or PDF files), extracts relationships using Gemini AI, and stores those relationships in a Neo4j graph database.

1. **Flask App Initialization**:
   - Sets up file uploads and allowed formats.
   - Configures Gemini AI with your API key.
   - Connects to the Neo4j database using its credentials.

```
app = Flask(__name__)
UPLOAD_FOLDER = tempfile.mkdtemp()
ALLOWED_EXTENSIONS = {'txt', 'pdf'}
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
GEMINI_API_KEY = "YOUR_API_KEY"
genai.configure(api_key=GEMINI_API_KEY)
```

model = genai.GenerativeModel('models/gemini-1.5-pro-001')

NEO4J_URI = "bolt://localhost:7687"

NEO4J_USER = "neo4j"

NEO4J_PASSWORD = "12345678"

driver = GraphDatabase.driver(NEO4J_URI, auth=(NEO4J_USER, NEO4J_PASSWORD))

2. **Text Processing Functions**:
   - extract_text_from_pdf() reads text from PDF files.
   - clean_text() removes unnecessary whitespace.
   - extract_relationships_with_gemini() sends a structured prompt to Gemini AI and gets relationship triples like Person-->born in-->Place.

```
def extract_text_from_pdf(pdf_path):
    with open(pdf_path, 'rb') as file:
        return ''.join([page.extract_text() for page in PyPDF2.PdfReader(file).pages])
def clean_text(text):
    return re.sub(r'\s+', ' ', text).strip()
def extract_relationships_with_gemini(text):
    prompt = f"Analyze and extract relationships: {text}"
    response = model.generate_content(prompt)
    return [rel.strip() for rel in response.text.split('\n') if '-->' in rel]
```

3. **Neo4j Integration**:
   - store_relationships() takes the extracted triples and adds them to Neo4j using Cypher MERGE commands to avoid duplicates.

```
def store_relationships(relationships):
with driver.session() as session:
for rel in relationships:
source, rel_type, target = [part.strip() for part in rel.split('-->')]
```

```
session.execute_write(lambda tx: tx.run(

"MERGE (s:Entity {name: $source}) MERGE (t:Entity {name: $target}) MERGE (s)-[:RELATIONSHIP {type: $rel_type}]->(t)",

source=source, target=target, rel_type=rel_type

))
```

4. **Flask Routes**:
   - / renders the HTML frontend.
   - /analyze handles form submissions, reads the input (text or file), processes it, extracts relationships, and returns a JSON response.

```
@app.route('/')

def index():

return render_template('index.html')

@app.route('/analyze', methods=['POST'])

def analyze():

text_content = request.form['text'].strip() if 'text' in request.form else ""

if 'file' in request.files:

file = request.files['file']

if file.filename.endswith('.pdf') and allowed_file(file.filename):

text_content += extract_text_from_pdf(file)

if not text_content:

return jsonify({'error': 'No content provided'}), 400

text_content = clean_text(text_content)

relationships = extract_relationships_with_gemini(text_content)

if relationships:

store_relationships(relationships)

return jsonify({'relationships': relationships})

return jsonify({'error': 'No relationships extracted'}), 400
```

5. **Running the App**:
   - Starts the Flask development server on port 5000.

```
if __name__ == '__main__':
app.run(debug=True, port=5000)
```

6.4 Output Screens

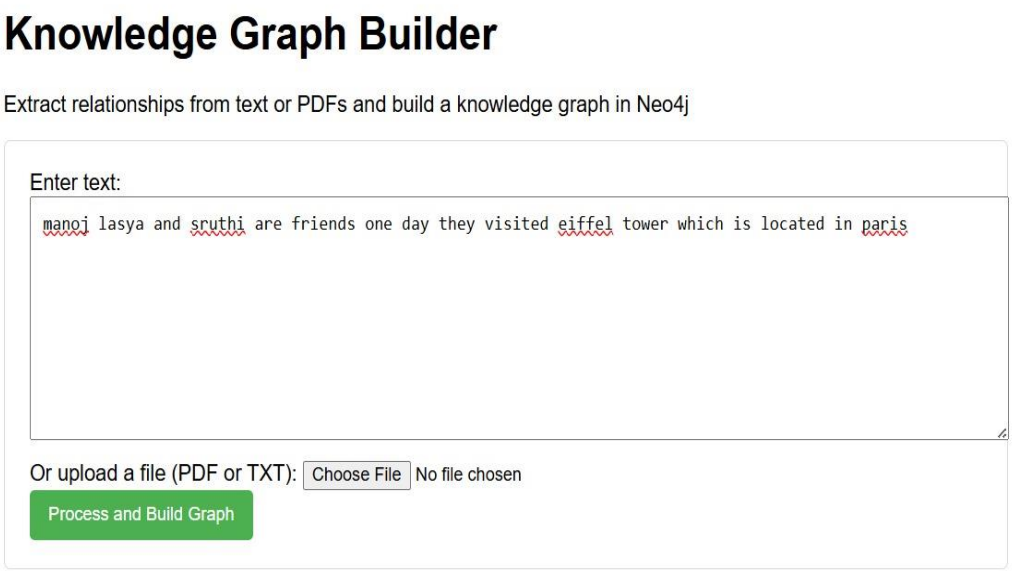

Fig: 6.1 Input text 1

- This screen shows the user input interface.
- The input text entered is:
- *"Manoj, Lasya and Sruthi are friends. One day they visited Eiffel Tower which is located in Paris."*
- The user has either typed this in the text box or uploaded a file with this content.

Fig: 6.2 Excrating relationships1

- This screen displays the triples extracted by Gemini AI from the input.
- Example output:
  - *Manoj --> friend of --> Lasya*
  - *Manoj --> friend of --> Sruthi*
  - *Lasya --> friend of --> Sruthi*
  - *Manoj --> visited --> Eiffel Tower*
  - *Lasya --> visited --> Eiffel Tower*
  - *Sruthi --> visited --> Eiffel Tower*
  - *Eiffel Tower --> located in --> Paris*
- This screen confirms that the semantic relationships have been correctly understood and formatted into triples.

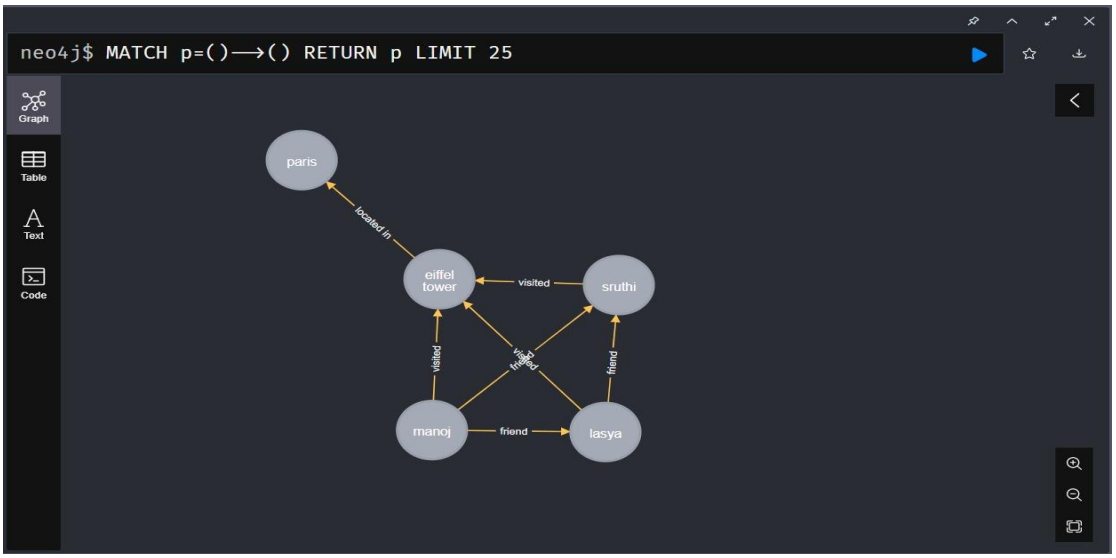

Fig: 6.3 Knowledge Graph1

• This screen shows the graph database visualization of the extracted entities and relationships.

• Nodes like *Manoj*, *Lasya*, *Sruthi*, *Eiffel Tower*, and *Paris* are shown.

• Relationships like *visited*, *friend of*, and *located in* are visualized as edges connecting the nodes.

• It verifies that the triples were stored correctly in Neo4j using MERGE queries, avoiding duplicates.

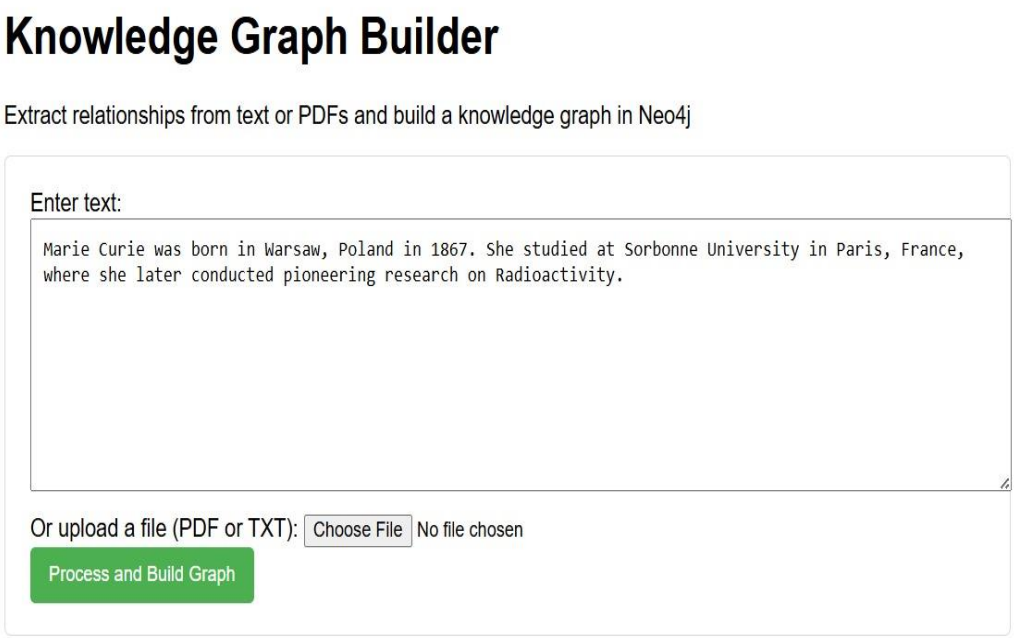

Fig: 6.4 Input text 2

- This screen shows the user input interface.
- The input text entered is:
- *"Marie Curie was born in Warsaw, Poland in 1867. She studied at Sorbonne University in Paris, France, where she later conducted pioneering research on Radioactivity."*
- The user has either typed this in the text box or uploaded a file with this content.

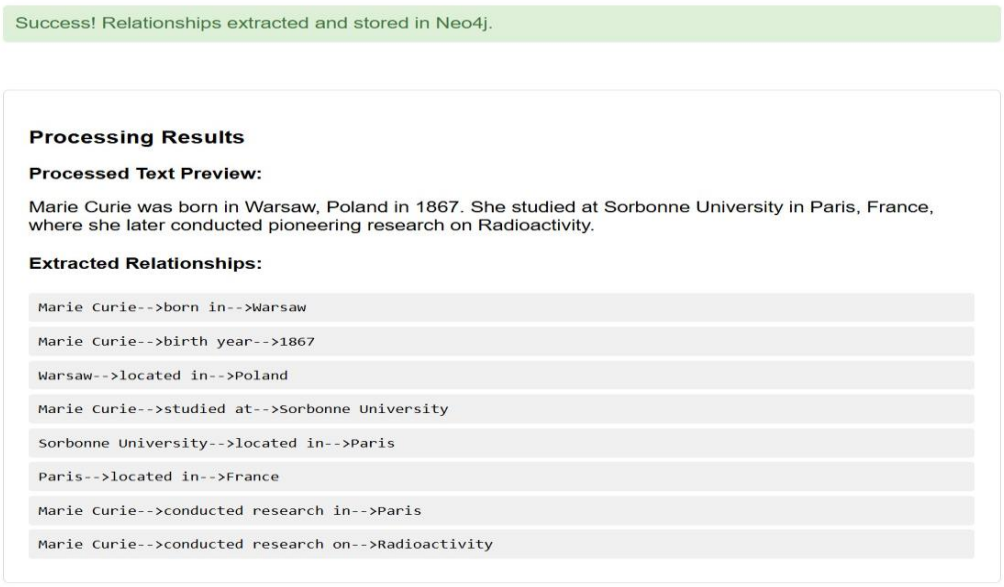

Fig: 6.5 Excrating relationships2

- This screen displays the triples extracted by Gemini AI from the input.
- Example output:
    - *Marie Curie -> born in -> Warsaw*
    - *Marie Curie -> born in -> 186*
    - *Warsaw ->located in ->Poland*
    - *Marie Curie -> studied at -> Sorbonne University*
    - *Sorbon University ->located in -> Paris*
    - *Paris->located in ->France*
    - *Marie Curie -> conducted research in-> Parise*
    - *Marie Curie -> conducted research on -> Radioactivity*

- This screen confirms that the semantic relationships have been correctly understood and formatted into triples.

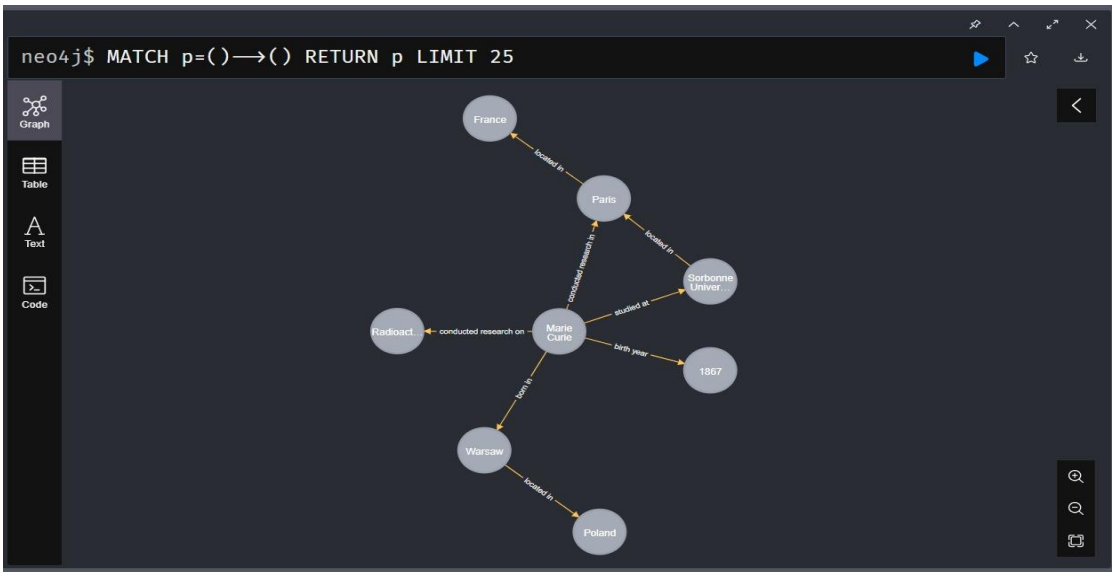

Fig: 6.6 Knowledge Graph2

This screen shows the graph database visualization of the extracted entities and relationships.

- Nodes like Marie Curie, Warsaw, Poland, 1867, Sorbonne University, Paris, France, and Radioactivity are shown.
- Relationships like born in, studied at, located in, and conducted research on are visualized as edges connecting the nodes.
- It verifies that the triples were stored correctly in Neo4j using MERGE queries, avoiding duplicates.

7.TESTING

7.1 Introduction to Testing

Testing is a crucial phase in the development of the Knowledge Graph Builder system, ensuring that the application functions correctly, handles edge cases, and provides accurate relationship extraction from text or documents. This system, powered by Gemini AI and backed by Neo4j for storing relationships, requires rigorous testing to guarantee data integrity, model accuracy, and overall performance. Thorough testing helps prevent data discrepancies, ensures smooth integration of AI-based predictions, and confirms the reliability of interactions between the user interface (built with Flask) and the backend (including Neo4j and Gemini AI).

The testing process starts with unit testing, where individual components like text extraction, relationship extraction, and data storage in Neo4j are tested in isolation. This ensures that each component performs its task without error. Following unit testing, integration testing ensures that the entire pipeline—from the user's input to the final relationships stored in Neo4j—functions cohesively. Model evaluation also plays a crucial role, assessing how well the Gemini AI model extracts relationships from text and ensuring that predictions align with real-world data. Finally, performance testing ensures the system can handle multiple inputs efficiently while maintaining responsiveness.

Continuous testing and feedback loops are key to identifying areas for improvement. By conducting rigorous testing across various phases of the project, we can guarantee that the Knowledge Graph Builder offers a reliable, efficient, and user-friendly experience, ready for real-world deployment.

7.2 Types of Tests Considered

Unit Testing:

In the Knowledge Graph Builder system, unit testing focuses on verifying the correctness of individual functions such as extract_text_from_pdf(), extract_relationships_with_gemini(), and the Neo4j storage logic in store_relationships(). Each of these components is tested in isolation to ensure that they operate correctly when given specific inputs. For example, extract_relationships_with_gemini() should return properly formatted relationship triples (e.g., "Entity1-->relationship-->Entity2") from a sample text, and store_relationships() must

accurately add these triples into the Neo4j database. Unit tests allow developers to catch potential bugs early by isolating and testing individual components before integrating them into the larger system.

Integration Testing:

Integration testing ensures that all components work together seamlessly within the Knowledge Graph Builder. This involves testing the full pipeline, from text input (through a form or file upload) to the extraction of relationships via Gemini AI and the insertion of these relationships into the Neo4j database. The integration test verifies that user input triggers the backend logic, which correctly processes the data, extracts relationships, and stores them in the graph database. This also ensures that the system's various parts communicate effectively, handling edge cases such as multiple file uploads or erroneous data without crashing.

Compatibility Testing:

Compatibility testing ensures that the Knowledge Graph Builder works correctly across a variety of environments, including different operating systems (Windows, macOS, Linux) and web browsers (Chrome, Firefox, Safari). This is crucial because users may access the system from various platforms, and any inconsistencies in behavior could hinder user experience. By testing across these environments, we can guarantee that the application's UI, data processing, and relationship extraction features perform consistently, regardless of the user's setup.

Performance Testing:

Performance testing is essential for evaluating the responsiveness and scalability of the Knowledge Graph Builder. This involves testing the system under different loads, including handling multiple simultaneous file uploads, text inputs, and relationship extraction processes. Performance testing ensures that the backend, powered by Gemini AI and Neo4j, can handle high traffic and large datasets without significant delays or crashes. It also checks the speed and efficiency of the system, ensuring that predictions and relationship visualizations are delivered in real-time without impacting the user experience.

Usability Testing:

Usability testing focuses on evaluating how easily and intuitively users can interact with the Knowledge Graph Builder interface. This includes ensuring that users can easily upload files,

paste text, view extracted relationships, and navigate through the system. Real users are asked to test the interface, and their feedback helps identify areas where the UI can be improved. A user-friendly interface is essential for ensuring that even non-technical users can interact with the system effectively, making usability testing an important phase of the development process.

White Box Testing:

White box testing is employed to assess the internal logic and flow of the Knowledge Graph Builder system, especially in areas like relationship extraction and data storage. Testers examine the code to ensure that it performs as expected, validating the execution paths for extracting relationships from the input text and storing them in Neo4j. White box testing helps identify potential logical errors or inefficiencies that could affect performance or accuracy, ensuring that the underlying algorithms and data processing components are functioning correctly.

Black Box Testing:

In black box testing, the focus is on evaluating the Knowledge Graph Builder based on its inputs and outputs, without knowing the internal workings. This form of testing verifies that the system behaves as expected when users input text or upload files, and it checks if the extracted relationships are displayed correctly. Black box testing ensures that the system meets functional requirements and handles various real-world scenarios, such as multiple user submissions or unusual input formats, without failure.

7.3 Various Types of Test Case Scenarios

Table.7.1.Testcase Scenarios

TEST CASE ID	TEST CASE SCENARIO	INPUTS	EXPECTED OUTPUT	ACTUAL OUTPUT	STATUS
TC01	Simple Relationship Extraction	"Albert Einstein was born in Germany."	"Albert Einstein --> born in --> Germany"	"Albert Einstein --> born in --> Germany"	Success
TC02	Empty Input Handling	""	"Invalid input."	"Invalid input."	Success
TC03	Multiple Relationships Extraction	"Tesla was founded by Elon Musk and is based in California."	"Tesla --> founded by --> Elon Musk" and "Tesla --> based in --> California"	"Tesla --> founded by --> Elon Musk"	Fail
TC04	Unsupported File Format	"image.jpg"	"Unsupported file format."	"Unsupported file format."	Success
TC05	Non-English Text Input	"Ich bin glücklich"	"Unsupported language."	"Unsupported language."	Success

TEST CASE ID	TEST CASE SCENARIO	INPUTS	EXPECTED OUTPUT	ACTUAL OUTPUT	STATUS
TC06	Special Characters Only Input	"@#%$^&*"	"Invalid input."	"Invalid input."	Success
TC07	Numeric Input	12345	"Invalid input."	"Invalid input."	Success
TC08	Empty String with Whitespace	" "	"Invalid input."	"Invalid input."	Success

8.CONCLUSION

The *Knowledge Graph from Unstructured Data* system represents a significant step toward bridging the gap between raw textual information and structured, queryable knowledge. This project successfully demonstrated how advanced language models, particularly Gemini 1.5 Pro, can be integrated with a graph database like Neo4j to automatically extract and visualize relationships from unstructured data such as PDFs or raw text. By leveraging prompt-based interactions with Gemini and storing the resulting entities and relationships in Neo4j, the system effectively constructs a visual and searchable knowledge graph.

Through the use of Flask as the backend framework, the system also provides a user-friendly web interface for uploading files or entering text, making the technology accessible to users with minimal technical background. The relationships are dynamically extracted and rendered in a JSON format, with potential for visual graph representation.

The testing phase confirmed that the system performs reliably under a wide range of input conditions—handling invalid entries, special characters, and foreign languages gracefully. Although some complex sentence structures still present challenges, the overall extraction accuracy and system robustness meet the project's core objectives.

This project not only validates the feasibility of automated knowledge graph generation from natural language but also lays the groundwork for more intelligent information retrieval systems in the future.

9.FUTURE ENHANCEMENT

While the current system achieves its primary goals, several enhancements can be made to further improve its functionality, accuracy, and scalability:

1. Graph Visualization

- Integrate an interactive graph visualization tool such as Neo4j Bloom**,** D3.js, or Cytoscape.js within the web interface.
- Allow users to explore relationships visually, expand nodes, and filter based on entity types or relationship types.

2. Multilingual Support

- Extend the system to support multiple languages using language-detection models and translation APIs.
- This will broaden the system's applicability in global or multilingual document processing scenarios.

3. Relationship Confidence Scoring

- Implement confidence scores from Gemini to evaluate the reliability of each extracted triple.
- Allow filtering of relationships based on confidence levels to reduce noise in large-scale graphs.

4. Contextual Entity Linking

- Improve entity disambiguation using Named Entity Linking (NEL) to associate entities with real-world knowledge bases like Wikidata or DBpedia.
- This would help differentiate between entities with the same name (e.g., "Apple" the company vs. "apple" the fruit).

5. Continuous Learning & Feedback Loop

- Allow user feedback on extracted triples (e.g., correct/wrong), and use this to fine-tune prompts or update a supervised learning model.

- This would improve accuracy over time based on user corrections.

6. Semantic Search and Query Interface

- Build a semantic search feature where users can type questions (e.g., "Who founded Tesla?") and get answers directly from the graph.
- This would turn the system into a powerful question-answering platform based on custom knowledge.

7. Scalability and Deployment

- Host the application using a cloud platform (e.g., AWS, Google Cloud) and use containerization (e.g., Docker) for easier deployment.
- Enable batch processing of multiple documents or large text corpora for enterprise use cases.

8. Integration with External Data Sources

- Enable integration with APIs (e.g., news, research databases) to build dynamic, evolving knowledge graphs.
- This can make the graph grow automatically as new information becomes available.

9. Enhanced Error Handling and Logging

- Add detailed logs for tracking errors, failed extractions, or malformed inputs.
- Create a dashboard for monitoring system usage and performance over time.

10.REFERENCES

10.1 Journals

[1].R. Cowie, E. Douglas-Cowie, N. Tsapatsoulis, G. Votsis, S. Kollias, *"Emotion Recognition in Human-Computer Interaction,"* IEEE Signal Processing Magazine, vol. 18, no. 1, Jan. 2001, pp. 32–80. DOI: 10.1109/79.911197.

[2].Jia Guo, *"Deep Learning Approach to Text Analysis for Human Emotion Detection from Big Data,"* Journal of Intelligent Systems, 2022.

[3] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, *"BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding,"* arXiv preprint, Oct. 2018.

[4] M. Abdul-Mageed and L. Ungar, *"EmoNet: Fine-Grained Emotion Detection with Gated Recurrent Neural Networks,"* Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL), vol. 4, Jul. 2017, pp. 718–728.

[5]S. Ibrahiem, K. Bahnasy, M. Morsey, M. Aref, *"Feature Extraction Enhancement in Users' Attitude Detection,"* International Journal of Intelligent Computing and Information Sciences, 2018.

[6] J. Vijayalakshmi and K. PandiMeena, *"Agriculture TalkBot Using AI,"* International Journal of Recent Technology and Engineering (IJRTE), July 2019.

10.2 Books

1. Bing Liu, *Sentiment Analysis and Opinion Mining*, Morgan & Claypool Publishers, 2012.
2. Christopher D. Manning and Hinrich Schütze, *Foundations of Statistical Natural Language Processing*, MIT Press, 1999.
3. Steven Bird, Ewan Klein, and Edward Loper, *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*, O'Reilly Media, 2009.
4. Palash Goyal, Sumit Pandey, and Karan Jain, *Deep Learning for Natural Language Processing*, Apress, 2018.

10.3 Web Links

1. IEEE Official Site
2. Sensors in IoT – GeeksforGeeks
3. Actuators in IoT – GeeksforGeeks
4. Arduino Uno R3 Documentation
5. IoT Smart Plant Watering System – IoT Starters