KDSH_2026 (KGPLites).(Track A)

Team Project: Kharagpur Data Science Hackathon 2026
Collaborators: Arit Patra, Anish Saha, Sujoy Lohar

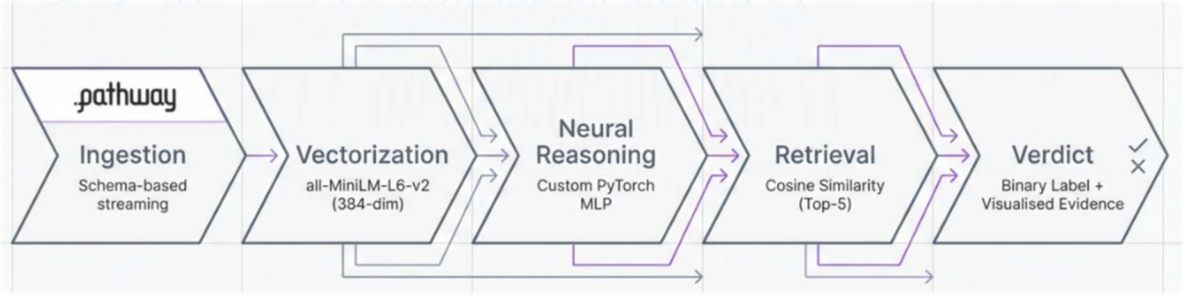
1. Overall Approach

Our solution is a high-precision **Retrieval-Augmented Generation (RAG)** system designed to validate narrative consistency between a character's backstory and their actions in a story text. The pipeline integrates a streaming data layer with a custom neural network to classify behavior as either "Consistent" or "Contradicted".

Pipeline Architecture

1. The system operates in **FIVE distinct stages**:

- **Ingestion:** Raw text is processed and stored using a Pathway schema-based streaming table.
- **Vectorization:** Text data is converted into 384-dimensional embeddings using the "all-MiniLM-L6-v2" Sentence-Transformer model.
- **Neural Reasoning:** Claims are processed by a custom PyTorch Consistency Model (MLP) to generate a confidence score.
- **Retrieval:** The system uses Cosine Similarity to extract the Top-5 most relevant passages from the vector store.
- **Final Verdict:** The system outputs a binary label paired with the retrieved evidence, visualized via a web interface.



2. Handling Long Context

Analyzing full-length stories requires managing context windows effectively to prevent information loss. We address this through a "Chunking and Retrieval" strategy:

- **Fixed-Size Chunking:** During ingestion, raw text is split into 400-word chunks. This ensures input fits within token limits while maintaining granular indexing.
- **Semantic Retrieval:** Rather than processing the entire story at once, the system retrieves only the Top-5 relevant passages based on vector similarity. This narrows the context to the specific scenes required for validation.

3. Distinguishing Causal Signals from Noise

To separate actual causal evidence (signals) from irrelevant narrative details (noise), the system employs a **two-tier filtering mechanism**:

A. Vector-Space Filtering (Semantic)

The first layer utilizes "**all-MiniLM-L6-v2**" to project text into a 384-dimensional space. Cosine Similarity ensures retrieved passages are semantically aligned with the query, filtering out unrelated scenes (noise).

B. Neural Architecture (Structural)

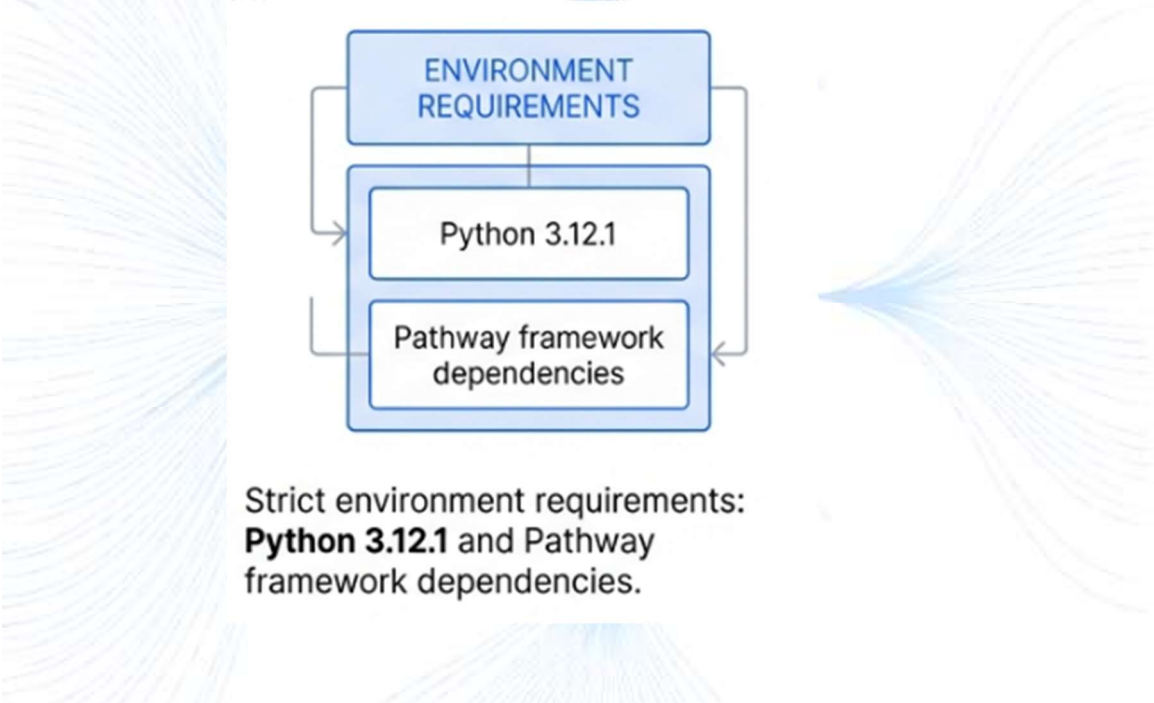
The second layer is a custom Multi-Layer Perceptron (MLP) trained to learn non-linear relationships. The architecture includes:

- **Hidden Layer (128 units):** We have used **ReLU-activated network** to capture complex feature mappings.
- **Regularization:** Dropout (0.3) is used to prevent overfitting, ensuring the model generalizes based on causal patterns rather than memorizing keywords.

4. Key Limitations or Failure Cases

Based on the current architecture, the following limitations are identified:

- **Chunk Boundary Loss:** The system splits text into 400-word chunks. Causal links spanning across two chunks (e.g., setup in chunk N, payoff in N+1) may result in incomplete context retrieval.
- **Model Dimensionality:** The 384-dimensional model (MiniLM) is efficient but smaller than larger transformers. It may struggle with deep ambiguity or highly subtle literary nuances compared to 768-dimension models.
- **Dependency Constraints:** The pipeline relies strictly on Python 3.12.1 and the Pathway framework, creating specific environment dependencies for deployment.



5. Frontend and Backend for the Web-Interface & UI

The application serves as a bridge between complex backend inference logic and a user-friendly frontend. Built using **Flask**, the application provides a "chatbot-style" environment where users can upload story text (.txt) and character backstory data (.csv).

- The app processes these inputs using an “run_inference.py”.
- Then it returns a structured CSV and JSON response.
- Here the JSON Response is processed to be made visible in card-like div(s) in the chatbot interface.
- The UI is coded with proper CSS and JS for it to work in sync with the main “APP.PY”.

The Actual Implementation:

The lifecycle of a user request follows this specific trajectory:

1. **User Input:** The user selects a story file and a backstory file via the frontend interface.
2. **Transmission:** JavaScript bundles these files into a **FormData** object and transmits them asynchronously via a POST request to the /chat endpoint.
3. **Server Processing:**
   The Flask server receives the files and saves them to a designated uploads/ directory as **story.txt** and **train.csv**. The server triggers the external **run_inference.py** module and executes **csvtojson.py** to format the inference results.
4. **Polling Mechanism:** The server enters a wait loop, checking for the existence of response.json before sending the response back to the client.
5. **Client Rendering:** The frontend receives the JSON data and dynamically constructs HTML elements (chat bubbles and result cards) to display the analysis.

Backend Implementation (Flask)

The backend is made using **Python (Flask)**, chosen for its lightweight nature and ease of integration with local file systems and external Python scripts.

6. Frontend Implementation of JS:

The client-side logic, contained in script.js, manages the state of the application and handles the transformation of raw JSON data into a rich user interface.

6.1 Asynchronous Communication

The **sendMessage()** function employs **the modern async/await pattern** with the Fetch API. This prevents the browser window from freezing during the potentially long inference process. A **"Thinking..."** state is visually rendered in the chat box to manage user expectations during this latency period.

6.2 Dynamic DOM Manipulation

A key feature of this application is that it does not reload the page to show results. Instead, it injects HTML directly into the DOM:

- **Result Cards:** The script iterates through the returned JSON array. For each entry, it creates <a .result-card> div.
- **Logic-Based Styling:**
    - **Consistency Checks:** The script checks “row.judgment”. If the string includes "inconsistent", it applies the “.pill-inconsistent” CSS class (red); otherwise, it applies “.pill-consistent” (green).
- **Interactive Evidence:** Each card is generated with a "Show Evidence" button attached to an event listener. This listener triggers a modal window (openModal()), populating it with the specific evidence text associated with that result.

7.Visual Identity and User Experience (CSS)

The visual design, defined in **style.css**, is critical to the application's usability. It moves beyond a standard utility tool, offering a polished, modern aesthetic that aligns with the theme of the hackathon.

- The **Glassmorphism** Effect in some portions of the site.
- The UI is made in a **Responsive Chatbot** Layout for better.
- Subtle **Animations** are applied across all elements of the UI such as the message bubbles and Judgement Cards of the Result giving it a modern and minimalistic look.

8. Conclusion:

This hackathon project involves various domains of ML/NLP and responsive web apps demonstrating use of popular libraries such as torch, transformers, pandas, numpy, flask, etc. This has given all three of us an opportunity to look into many aspects of modern programming in languages like Python, JavaScript, CSS, HTML.