Project Blueprint: A Multilingual RAG-Based Conversational Assistant for Educational Institutions

Section 1: System Architecture and Technology Stack

This section outlines the foundational architecture and the recommended technology stack for the Language Agnostic Chatbot. The selection of technologies is guided by the core project requirements: rapid development for a hackathon, robust multilingual capabilities, ease of deployment for end-users (universities), and long-term maintainability by student volunteers. The architecture is designed to be modular, scalable, and resilient, separating concerns to allow for parallel development by the team.

1.1. Architectural Overview

The system is designed around a decoupled, service-oriented architecture consisting of four primary subsystems. This separation ensures that components can be developed, tested, and scaled independently, which is a significant advantage in a time-constrained environment like a hackathon.

1. **Frontend Services:** This layer represents all user-facing interfaces. It is composed of two distinct applications:
    - **Admin Dashboard:** A web application for university administrators to configure their chatbot instance, manage knowledge sources (websites, PDFs), and review conversation logs.
    - **Embeddable Chatbot Widget:** A lightweight, self-contained component that universities can easily embed into their existing websites to provide the chat interface for students.
2. **Backend API (The Core):** This is the central nervous system of the application. It is a web server that exposes a set of RESTful API endpoints. Its responsibilities include

handling chat requests from the student widget, processing administrative actions from the dashboard, managing user authentication for admins, and orchestrating the AI pipeline.

3. **Asynchronous Ingestion Worker:** This is a background process that handles computationally expensive and time-consuming tasks. When an administrator adds a new knowledge source (e.g., a website URL or a set of PDFs), the Backend API offloads the actual processing to this worker. This prevents the API from becoming unresponsive and provides a better user experience for the administrator. The worker's tasks include web scraping, PDF text extraction, document chunking, and generating embeddings.
4. **AI Core (RAG Pipeline):** This is the intelligence layer of the system. It is not a single service but a collection of models and data stores that work together to understand and answer student queries. Its components include a specialized multilingual embedding model, a vector database for efficient similarity search, and a generative Large Language Model (LLM) for synthesizing final answers. The entire process is orchestrated using a dedicated framework to manage the flow of data.

Data and Logic Flow

To understand how these subsystems interact, consider two primary user journeys:

- **Administrator Journey (Onboarding & Knowledge Management):**
  1. An administrator from a university logs into the Admin Dashboard.
  2. They navigate to the "Knowledge Sources" section and provide the URL of the university's main website and upload several PDF documents containing FAQs, circulars, and fee schedules.
  3. Submitting this form sends a request to the Backend API's /admin/ingest endpoint.
  4. The API validates the request, authenticates the admin, and creates a new task for the Asynchronous Ingestion Worker, passing along the provided URLs and file paths. It immediately returns a "Task Accepted" response to the dashboard.
  5. The Ingestion Worker picks up the task. It systematically scrapes the website content, extracts text from the PDFs, cleans the text, splits it into manageable chunks, converts these chunks into numerical vectors using the multilingual embedding model, and finally stores these vectors along with their source metadata in the Vector Database.
- **Student Journey (Querying the Chatbot):**
  1. A student visits the university website and interacts with the Embeddable Chatbot Widget.
  2. The student types a query in their native language (e.g., Hindi: "फीस जमा करने की आखिरी तारीख क्या है?").
  3. The widget sends this query to the Backend API's public /chat endpoint, along with a

unique identifier for the university.
4. The API first detects the language of the query.
5. It then uses the appropriate multilingual embedding model to convert the student's query into a vector.
6. This query vector is used to search the Vector Database for the most semantically similar document chunks previously ingested for that specific university.
7. The API constructs a detailed prompt containing the original query, the retrieved document chunks (the "context"), and a strict instruction to answer only in the detected language.
8. This prompt is sent to the generative LLM.
9. The LLM generates a response based on the provided context, in the requested language.
10. The Backend API streams this response back to the chatbot widget, which displays it to the student. Simultaneously, the interaction (query, response, language) is logged for continuous improvement.

1.2. Recommended Technology Stack and Justification

The choice of technology for each component is critical to balancing performance, development speed, and maintainability.

- **Backend Framework: FastAPI**
  - **Justification:** While Flask is known for its simplicity, FastAPI is the superior choice for this API-centric project.[1] FastAPI is built on ASGI (Asynchronous Server Gateway Interface), which allows for native asynchronous request handling. This results in significantly higher performance and concurrency compared to Flask's traditional WSGI model, making it ideal for handling many simultaneous chat connections.[2] For a hackathon, FastAPI's most compelling features are its developer experience enhancements. It uses Python type hints and the Pydantic library for automatic request validation, serialization, and deserialization. This drastically reduces the amount of boilerplate code needed for data validation and error handling.[3] Furthermore, FastAPI automatically generates interactive API documentation (using OpenAPI and Swagger UI) from the code itself.[4] This feature directly addresses the project's "maintainability" requirement. Future student volunteers will not need to reverse-engineer the API; they can simply navigate to the /docs endpoint to see a complete, interactive specification of all available endpoints, their parameters, and their response models. This creates a self-documenting system that is far easier to onboard new developers to than a Flask application that requires manual documentation efforts.
- **Vector Database: ChromaDB**

    - **Justification:** The selection of a vector store is a pivotal decision for any RAG system. The primary contenders in the open-source space are FAISS (a library) and ChromaDB (a database). FAISS is a low-level library from Meta, optimized for extreme speed and scalability on massive datasets (tens of millions to billions of vectors).[5] However, it requires developers to manually handle data persistence, metadata storage, and complex index tuning, adding significant engineering overhead.[5] ChromaDB, in contrast, is a high-level, open-source vector *database* specifically designed for ease of use in AI applications and RAG pipelines.[5] It offers a developer-friendly API, built-in persistence, and, crucially, integrated support for storing and filtering by metadata alongside the vectors.[8] For a hackathon project where the dataset size for a single university is unlikely to exceed a few million vectors, the raw performance advantage of a highly tuned FAISS index is unnecessary. The development velocity gained from ChromaDB's simplicity, plug-and-play persistence, and seamless integration with frameworks like LangChain makes it the pragmatic and strategically sound choice.[8] It allows the team to focus on the application logic rather than on the intricacies of vector index management.
- **RAG Orchestration Framework: LangChain**
    - **Justification:** Both LangChain and LlamaIndex are powerful frameworks for building LLM applications. However, they have different primary focuses. LlamaIndex is highly specialized and optimized for the data indexing and retrieval parts of a RAG pipeline.[10] LangChain, on the other hand, is a more general-purpose and flexible framework designed for orchestrating complex LLM workflows, or "chains".[10] Given the project requirements—which include multi-turn conversation context management, potential integration of agents, and a "fallback to human" mechanism—LangChain's broader toolkit and more modular architecture are a better fit.[10] It provides a more extensive set of integrations and abstractions for managing prompts, models, and memory, which will be essential for building a feature-rich conversational assistant.
- **Web Scraping: requests and BeautifulSoup**
    - **Justification:** For scraping the content of university websites, the combination of requests and BeautifulSoup is the industry standard in Python. The requests library provides a simple and elegant way to send HTTP requests and retrieve the raw HTML content of a webpage.[11] BeautifulSoup is a powerful library for parsing that HTML, creating a navigable tree structure that allows for the robust extraction of text content by selecting elements based on their tags (e.g., <p>, <h1>), classes, or IDs.[7] For websites that rely heavily on JavaScript to render content, this stack can be augmented with Selenium, which automates a real web browser to execute the JavaScript before parsing the final page source.[13] The initial implementation should start with requests and BeautifulSoup for simplicity and performance, adding Selenium only if necessary for specific, dynamic sites.
- **Frontend Framework (Admin): React or Vue.js**

    - **Justification:** The Admin Dashboard requires a rich, interactive user interface for managing data sources and viewing logs. A modern Single Page Application (SPA) framework like React or Vue.js is the ideal choice for building this. The specific choice between them can be left to the team's existing expertise, as both are highly capable. Using a framework will enable the creation of a polished and responsive user experience for administrators.
- **Embeddable Widget: Native Web Components**
    - **Justification:** The project's unique value proposition is the ease with which any university can integrate the chatbot. A simple script or React component is a viable but suboptimal solution, as it can lead to CSS conflicts or require the host university to use a specific frontend framework. The most robust and professional solution is to build the widget using native Web Components.[14] This W3C standard allows for the creation of custom, reusable HTML elements with encapsulated functionality. By using the Shadow DOM, the widget's styles and scripts are completely isolated from the host page's styles and scripts, and vice-versa.[14] This guarantees that the chatbot will look and function correctly on any website, regardless of the technologies it uses (WordPress, Drupal, Angular, etc.). This framework-agnostic approach provides a true "plug-and-play" experience, fulfilling the core deployment goal in the most resilient way possible.

Section 2: The AI Core: A Deep Dive into the Multilingual RAG Pipeline

The success of this project is fundamentally dependent on the quality and accuracy of its AI core. This section details the design of the multilingual Retrieval-Augmented Generation (RAG) pipeline, from initial data ingestion to the final generation of a language-specific response. The strategies outlined here are specifically chosen to address the challenges of working with multiple Indian languages.

2.1. Data Ingestion and Processing

The foundation of any RAG system is a comprehensive and clean knowledge base. The ingestion process must be robust enough to handle various data formats and structures found in university documents.

- **Web Scraping:** The asynchronous ingestion worker will be responsible for this process. It

will use the requests library to fetch the HTML of a given URL. To be systematic, the scraper should first attempt to find and parse the website's /sitemap.xml file, which provides a structured list of all pages. For each page, BeautifulSoup will be used to parse the HTML. To minimize noise and extract only relevant information, the parser will be configured to target main content containers (e.g., elements with tags like <main>, <article>, or IDs like content) and extract text from semantic tags within them (e.g., <p>, <h1> through <h6>, <li>). It will be programmed to explicitly ignore common noisy sections like navigation bars (<nav>), footers (<footer>), and sidebars (<aside>).

- **PDF Processing:** University circulars, forms, and prospectuses are often distributed as PDFs. The worker will use a dedicated Python library such as PyMuPDF or pdfplumber to extract text from these documents. These libraries are more effective than simpler tools as they can often preserve tabular data and handle complex layouts, which is crucial for accurately interpreting fee structures or timetables.
- **Text Chunking Strategy:** LLMs have a limited context window, meaning they cannot process entire documents at once. Therefore, the extracted text from both web pages and PDFs must be split into smaller, more manageable pieces. This will be accomplished using LangChain's RecursiveCharacterTextSplitter.[15] This splitter intelligently tries to break text along semantic boundaries (paragraphs, sentences) to keep related concepts together. A recommended starting configuration is a chunk size of approximately 1000 characters with an overlap of 200 characters. The overlap is critical; it ensures that a sentence or idea that starts at the end of one chunk and finishes at the beginning of the next is fully contained in at least one of the chunks, preventing the loss of semantic context at chunk boundaries.

2.2. The Embedding Strategy for Indian Languages

Choosing the right embedding model is the most critical decision for the multilingual capabilities of this chatbot. Standard multilingual models are often trained on web-scale data where English is overrepresented, leading to weaker performance for many Indian languages. Therefore, leveraging models specifically trained or fine-tuned for the Indic language family is essential.

A careful analysis of available models reveals a strategic path forward. Instead of relying on a single model, a hybrid approach will yield the most robust and user-centric results.

- **Model Comparison and Selection:**
  - **ai4bharat/indic-bert:** This model from AI4Bharat is a strong foundational choice, pretrained exclusively on 12 major Indian languages, including Assamese, Bengali, Gujarati, Hindi, Kannada, Malayalam, Marathi, Oriya, Punjabi, Tamil, and Telugu.[7] Its performance on the IndicGLUE benchmark is competitive with or better than larger

models like mBERT and XLM-R, despite having fewer parameters.[7]
    - **Vyakyarth by Ola Krutrim:** This is a more modern sentence-transformer model built upon the powerful XLM-RoBERTa architecture and fine-tuned for Indic languages.[17] It is specifically designed for tasks like semantic search and similarity, mapping text into a 768-dimensional vector space. Its modern architecture makes it a prime candidate for the primary embedding model.
    - **AkshitaS/bhasha-embed-v0:** This model possesses a unique and critical feature: explicit support for **Romanized Hindi** (also known as "Hinglish").[18] Many students, particularly when typing on mobile devices, will use Roman script to write Hindi queries (e.g., "scholarship form last date kya hai?"). Most embedding models trained on Devanagari script will fail to understand these queries correctly. This model is the first of its kind to be cross-lingually aligned for Hindi, English, and Romanized Hindi, enabling queries in one script to retrieve documents in another.[18]
- **Recommended Hybrid Strategy:**
    1. **Primary Model:** Use **Vyakyarth** as the main embedding model for all document ingestion and for embedding user queries in all supported languages. Its strong performance and modern architecture provide a high-quality baseline for semantic retrieval across the board.
    2. **Specialized Handling for Hindi:** After the initial language detection step (detailed in 2.4), if a user's query is identified as Hindi (hi), the system will perform a conditional enhancement. In addition to embedding the query with Vyakyarth, it will also generate a second embedding using **bhasha-embed-v0**. The system will then perform two separate similarity searches in the vector store—one with each query vector—and merge the results. This dual-query approach ensures that the system can robustly handle both Devanagari and Romanized Hindi inputs, addressing a massive real-world use case and significantly improving the user experience for a large segment of the target audience.

Model Name	Base Architecture	Supported Indic Languages	Key Features
ai4bharat/indic-bert	ALBERT	Assamese, Bengali, English, Gujarati, Hindi, Kannada, Malayalam, Marathi, Oriya, Punjabi, Tamil, Telugu	Lightweight, strong baseline performance on IndicGLUE [7]
Vyakyarth	XLM-RoBERTa	Multiple Indic	Modern

		languages (fine-tuned)	architecture, optimized for semantic search, 768-dim vectors [17]
bhasha-embed-v0	Sentence Transformer	Hindi (Devanagari), English, Romanized Hindi	First model with Romanized Hindi support, cross-lingual alignment [18]

2.3. Vector Storage and Retrieval

Once the document chunks are converted into vector embeddings, they must be stored in a way that allows for fast and efficient similarity search.

- **Implementation:** The system will use the chromadb Python client. For each university that signs up, a new, separate collection will be created in ChromaDB, using a unique university ID as the collection name. This multi-tenancy approach ensures strict data isolation, meaning a query for one university will never retrieve documents from another.
- **Ingestion:** During the asynchronous ingestion process, each text chunk and its corresponding vector embedding (generated by Vyakyarth) are inserted into the appropriate university's collection. Critically, each vector will be stored with associated metadata. This metadata will include, at a minimum, the source of the text (e.g., the specific URL of the webpage or the filename of the PDF) and potentially the page number or section title. This metadata is vital for fulfilling the requirement of response accuracy, as it allows the system to cite its sources.
- **Retrieval:** When a student's query is received and embedded, the resulting vector(s) are used to perform a similarity search against the relevant ChromaDB collection. The collection.query() function will be used to find the top k most similar document chunks, where k is a configurable number (a good starting point is 5). The distance metric used for similarity, such as cosine similarity or L2 distance, can be configured within ChromaDB to optimize retrieval quality.

2.4. Language-Agnostic Generation

The final and most delicate step is generating a coherent, accurate, and

language-appropriate response for the user. This involves two key sub-processes: language detection and carefully structured prompt engineering.

- **1. Intent and Language Detection:**
  - **Tool Selection:** The first action performed on any incoming query is to determine its language. While several libraries exist, lingua-py stands out for its high accuracy, especially on short texts, which are common in chat applications.[19] It is significantly faster than older libraries like
    langdetect and supports a wide range of languages, including many relevant to the Indian context like Hindi, Bengali, Gujarati, Marathi, Punjabi, Tamil, Telugu, and Urdu.[19]
  - **Process:** The raw query string from the user will be passed to the lingua-py detector. The detected language's ISO 639-1 code (e.g., hi, en, bn, gu) will be captured and stored as a variable for the duration of the request. This variable will be the single source of truth for all subsequent language-specific operations.
- **2. Context-Aware Prompt Engineering:**
  - **The Challenge:** The core requirement is that the chatbot must respond *only* in the user's detected language. A common failure mode of multilingual LLMs is to revert to English, especially if the retrieved context documents are in English or if the query is complex. This behavior cannot be left to chance; it must be explicitly controlled.
  - **Technique:** The most effective way to control an LLM's output is through precise and forceful prompt engineering. The system will use a structured prompt template that leverages several best practices, including assigning a role to the model, providing clear and specific instructions, and structuring the context clearly.[20] The final prompt sent to the generative LLM will be dynamically constructed as follows:

    ```
    Code snippet
    You are a helpful and accurate academic assistant for {{ university_name }}.
    Your primary instruction is to answer the student's question based *only* on the information provided in the "CONTEXT" section below.
    You must respond exclusively and entirely in the following language: [{{ detected_language }}].
    Do not use English or any other language in your response unless it is the specified language. If the answer is not found in the context, state that you do not have the information, but do so in the [{{ detected_language }}] language.

    CONTEXT:
    ---
    {% for doc in retrieved_documents %}
    Source Document: {{ doc.metadata.source }}
    Content: {{ doc.content }}
    ---
    {% endfor %}

    QUESTION:
    {{ user_query }}
    ```

FINAL ANSWER IN [{{ detected_language }}]:

- **Rationale:** This prompt design is deliberate and multi-faceted. The role-playing ("You are a helpful... assistant") sets the tone. The instruction to use *only* the provided context grounds the model, minimizing the risk of "hallucination" or making up incorrect information. The repeated and emphasized instruction about the output language (Respond exclusively in..., FINAL ANSWER IN [...]) creates a strong directive that significantly increases the likelihood of compliance. By providing the source metadata, the system also lays the groundwork for potentially citing sources in its response, further enhancing trust and accuracy.

Section 3: Backend Implementation with FastAPI

This section provides a detailed blueprint for building the server-side application using the FastAPI framework. The focus is on creating a clean, modular, and maintainable codebase that can be developed rapidly during the hackathon.

3.1. Application and API Structure

A well-organized project structure is essential for team collaboration and long-term maintainability. The following modular structure is recommended:

```
/campus-chatbot-api

|-- /app
| |-- __init__.py
| |-- main.py             # FastAPI app instance and middleware
| |-- /api
| | |-- __init__.py
| | |-- /v1
| | | |-- __init__.py
| | | |-- chat.py         # /chat endpoint
| | | |-- admin.py        # /admin endpoints (ingestion, logs)
```

```
| |-- /core
| | |-- __init__.py
| | |-- config.py       # Environment variable loading
| | |-- security.py     # JWT authentication logic
| |-- /services
| | |-- __init__.py
| | |-- rag_service.py    # Core RAG pipeline logic
| | |-- ingestion_service.py # Data scraping and processing logic
| |-- /models
| | |-- __init__.py
| | |-- chat_models.py    # Pydantic models for chat req/res
| | |-- admin_models.py   # Pydantic models for admin req/res
| |-- /workers
| | |-- __init__.py
| | |-- tasks.py        # Celery task definitions
|-- requirements.txt
|--.env
```

This structure separates concerns effectively: api defines the HTTP layer, services contains the core business logic, models defines the data structures, and core handles cross-cutting concerns like configuration and security.

3.2. Core API Endpoints Specification

The API will expose a set of well-defined endpoints. FastAPI's use of Pydantic models will ensure all incoming and outgoing data is strongly typed and validated.

- **POST /api/v1/chat (Public)**
  - **Description:** The primary endpoint for student interaction with the chatbot.
  - **Request Body:** A Pydantic model ChatRequest defined as:
    ```
    Python
    class ChatRequest(BaseModel):
      query: str
      university_id: str
      conversation_id: Optional[str] = None
    ```
  - **Logic:**
    1. Receives and validates the ChatRequest.
    2. Calls the rag_service to execute the full RAG pipeline: detect language, embed query, retrieve context from the ChromaDB collection corresponding to

university_id, and generate a response from the LLM.
    3. The response should be streamed back to the client using StreamingResponse for a better, more immediate user experience.
    4. Logs the interaction details (anonymized query, response, language) to a database for later review.
  - **Response Body:** A streamed text response.
- **POST /api/v1/admin/ingest/sources (Admin, Secured)**
  - **Description:** Triggers the asynchronous ingestion of new knowledge sources for a university.
  - **Request Body:** A Pydantic model IngestionRequest defined as:

```
Python
class IngestionRequest(BaseModel):
    source_urls: List[str]
    # PDF file handling would be done via multipart/form-data upload
```

  - **Logic:**
    1. Verifies the JWT token to ensure the user is an authenticated administrator.
    2. Validates the request body.
    3. Dispatches a new task to the asynchronous worker (e.g., Celery) with the provided URLs and file paths.
    4. Immediately returns a 202 Accepted response with a task ID. The admin dashboard can use this ID to poll a status endpoint if needed.
- **GET /api/v1/admin/logs (Admin, Secured)**
  - **Description:** Retrieves conversation logs for review and analysis.
  - **Logic:**
    1. Verifies the JWT token.
    2. Queries the logging database, applying filters for date ranges or keywords and implementing pagination to handle large numbers of logs.
    3. Returns a paginated list of log entries.
- **Authentication:** All endpoints under the /admin/ path will be protected. The implementation will use JWT (JSON Web Tokens). An administrator will first authenticate via a /login endpoint with their credentials, receiving a short-lived access token. This token must then be included as a Bearer token in the Authorization header of all subsequent requests to protected endpoints. The security.py module will contain the logic for creating, decoding, and verifying these tokens.

3.3. Building the Admin Panel

Developing a full-featured CRUD (Create, Read, Update, Delete) interface for the admin dashboard from scratch can be time-consuming. A strategic approach to accelerate

development during the hackathon is to leverage one of the many third-party admin panel libraries available for FastAPI.

- **Strategy:** Instead of dedicating frontend resources to building tables, forms, and data management logic, the project will use a pre-built admin solution that integrates directly with the FastAPI backend.
- **Recommendation:** Libraries such as **FastAPI-Admin** [22] or **SQLAdmin** are excellent choices. These libraries allow developers to define "Resource" or "ModelView" classes that correspond to database models. The library then automatically generates a complete, secure, and user-friendly web interface for performing CRUD operations on that data.[7] This admin interface can be "mounted" as a sub-application within the main FastAPI app, instantly providing a functional backend for managing universities, admin users, and viewing logs.[22] This approach can save days of development effort, freeing up the team to focus on the core AI and chatbot functionality, which is the novel part of the project.

Section 4: Frontend and Deployment Strategy

This section details the user-facing components of the system: the Admin Dashboard where universities configure their chatbot, and the Embeddable Chatbot Widget that students interact with. The deployment strategy for the widget is a key part of the project's value proposition.

4.1. The Admin Dashboard Interface

The Admin Dashboard will be a web-based Single Page Application (SPA) providing a centralized control panel for university administrators.

- **Key Features:**
  - **Secure Login:** An authentication page where administrators can log in with their credentials to obtain a JWT session token.
  - **Dashboard Overview:** A landing page that presents key usage statistics, such as the total number of queries per day, a list of the most frequently asked questions (or recognized intents), and any queries that the chatbot failed to answer, which can highlight gaps in the knowledge base.
  - **Knowledge Source Management:** An intuitive interface where admins can:
    - Add one or more root URLs for the web scraper.

        - View the status of scraping and ingestion tasks (e.g., "in progress," "completed," "failed").
        - Upload PDF documents directly.
        - View and delete existing knowledge sources (URLs and PDFs).
    - **Conversation Logs Viewer:** A searchable and filterable table displaying all anonymized student interactions. This is crucial for fulfilling the "daily conversation logs for continuous improvement" requirement. Admins can review conversations to understand student needs and identify areas where the chatbot's knowledge or performance can be improved.
    - **Installation/Deployment Page:** A simple page that provides the copy-pasteable HTML code snippet required to embed the chatbot widget on the university's website. This page will prominently display the university's unique ID, which is a required parameter in the snippet.

4.2. The Embeddable Chatbot Widget

The widget is the primary student-facing component and its design must prioritize ease of integration and non-interference with the host website.

- **Technology: Native Web Components**
    - To achieve true framework-agnostic plug-and-play functionality, the widget will be built as a native Web Component (also known as a Custom Element). This involves defining a JavaScript class that extends the base HTMLElement class and registering it with the browser using customElements.define().[14] This approach is superior to providing a React or Angular component, as it has no external framework dependencies and will work on any webpage that supports modern JavaScript.
- **Encapsulation with the Shadow DOM**
    - A core feature of Web Components is the Shadow DOM. The component's constructor will call this.attachShadow({mode: 'open'}) to create an encapsulated DOM tree.[14] All of the widget's HTML (the chat window, message bubbles, input field) and CSS will be injected into this shadow root. This is the critical mechanism that prevents style collisions. The university's global CSS cannot "leak in" and break the chatbot's layout, and the chatbot's CSS cannot "leak out" and unintentionally alter the styles of the university's website.[14] This ensures the widget is a robust, self-contained black box.
- **Deployment Script and Configuration**
    - The script provided to universities on the Admin Dashboard's installation page will be exceptionally simple, designed for a non-technical user to copy and paste into their website's HTML. It will consist of two parts:
        1. A <script> tag that loads the JavaScript file defining the Web Component from a

central CDN or the project's server.
2. The custom HTML tag for the component itself.

- **Example Snippet:**

```HTML
<script src="https://your-chatbot-service.com/widget.js" defer></script>

<campus-chatbot
university-id="a1b2c3d4-e5f6-7890-gh12-i3j4k5l6m7n8"></campus-chatbot>
```

- The widget.js file contains all the logic for the Web Component. The component will read the university-id attribute from itself. This ID is then included in every API call to the /api/v1/chat endpoint, telling the backend which university's knowledge base to query. This simple attribute-based configuration makes the widget highly reusable and easy to deploy across multiple institutions.

Section 5: Operational Excellence: Privacy, Maintainability, and Execution Plan

A successful hackathon project is not just about a functional demo; it's about demonstrating a mature approach to software engineering. This section addresses the non-functional requirements of maintainability and privacy, and provides a concrete execution plan for the six-person team.

5.1. Designing for Maintainability

The solution must be easily understood and managed by student volunteers after the hackathon. This principle should inform every development decision.

- **Clean and Documented Code:** The codebase must be clean, well-structured, and commented. FastAPI's enforcement of Python type hints is a significant aid here, making function signatures self-explanatory. Complex business logic, especially within the RAG pipeline, should be accompanied by comments explaining the "why" behind the implementation.
- **Configuration Management:** No secrets, API keys, model names, or database connection strings should ever be hardcoded in the source code. All configuration must be managed through environment variables. A .env file will be used for local

development, and in a production environment, these variables would be set by the hosting platform. This allows future maintainers to update critical settings (like changing an LLM API key) without needing to modify and redeploy the application code.

- **Comprehensive README.md:** The project's root directory must contain a detailed README.md file. This document is the primary entry point for new developers. It should include:
  - A high-level architectural overview.
  - A list of all technology dependencies.
  - Step-by-step instructions for setting up the development environment and running the application locally.
  - An explanation of the project's structure.
  - A link to the auto-generated API documentation.

5.2. Privacy by Design

Handling student queries necessitates a proactive and transparent approach to data privacy. Building trust with both students and university administrators is paramount.

- **Legal and Platform Requirements:** If the chatbot collects any personal information, even inadvertently, a Privacy Policy is a legal requirement under laws like the California Online Privacy Protection Act (CalOPPA) and a best practice under GDPR.[25] Many platforms, such as Facebook Messenger, require a privacy policy for any integrated bots, regardless of data collection practices.[25] Adopting this practice from the start demonstrates professionalism and legal compliance.
- **Data Logging Strategy:** To fulfill the requirement for "continuous improvement" while respecting privacy, the logging strategy must be carefully defined.
  - **Data to Log:** The system should log the anonymized text of the user's query, the detected language, the final response generated by the LLM, and a non-identifying session ID to group turns of a single conversation. This data is sufficient for analyzing chatbot performance and identifying areas for improvement.
  - **Data to Exclude:** The system must be explicitly designed *not* to log any Personally Identifiable Information (PII). This includes IP addresses, browser user agent strings, or any other device-specific metadata that could be used to track an individual user.[26] If a user provides PII in their query (e.g., "My student ID is 12345, can you check my fee status?"), this data will be processed to generate a response but should ideally be scrubbed or anonymized before being stored in long-term logs.
- **Privacy Policy Implementation:** A clear and simple Privacy Policy must be created and made accessible from within the chatbot widget. This policy should state, in plain language:
  - What data is collected (anonymized conversation text).

    - The purpose of the collection (to improve the chatbot's accuracy and helpfulness).
    - How the data is stored (anonymously and securely).
    - That no PII is intentionally stored.[26]

5.3. Hackathon Execution Strategy (6-Person Team)

A structured plan with clear roles and phases is essential to maximize productivity and ensure all project goals are met within the tight timeframe of a hackathon.

- **Team Roles and Task Division:**
    - **Team Lead (1 person):** Responsible for the overall technical architecture, managing the project timeline, resolving integration issues, and leading the final presentation.
    - **Backend & API Team (2 people):** Tasked with building the entire FastAPI application. This includes setting up the project structure, defining all API endpoints with Pydantic models, implementing JWT authentication, and integrating the third-party admin panel.
    - **AI / RAG Pipeline Team (2 people):** This team focuses on the most complex part of the project. Their responsibilities include implementing the asynchronous ingestion worker, writing the web scraping and PDF parsing logic, researching, selecting, and implementing the hybrid multilingual embedding strategy, setting up ChromaDB, and fine-tuning the prompt engineering for language control.
    - **Frontend Team (1 person):** A dedicated frontend developer responsible for building both the React/Vue Admin Dashboard UI and the embeddable Web Component chatbot widget. This person will work closely with the Backend team to integrate with the API.
- **Phased Development Plan:**
    - **Phase 1: Setup & Core Pipeline (First 4-6 hours):** The immediate goal is to establish a "tracer bullet" — a thin, end-to-end slice of functionality.
        - All teams set up their development environments and project scaffolding.
        - The AI team's primary objective is to create a minimal, single-language (English) RAG pipeline. This involves hardcoding a small text document, chunking it, embedding it into ChromaDB, and creating a simple script that can take a query, retrieve context, and get a response from the LLM.
        - The Backend team sets up the basic FastAPI app and creates a single, unprotected /chat endpoint that calls the AI team's script.
    - **Phase 2: Feature Implementation (Next 10-12 hours):** This is the main development phase where teams work in parallel.
        - **Backend Team:** Implements all required API endpoints (/admin/ingest, /admin/logs, etc.), adds JWT security, and integrates the chosen admin panel library.

        - **AI Team:** Builds out the full, robust data ingestion pipeline (web scraper, PDF parser) within the asynchronous worker. They implement the language detection and the hybrid embedding strategy for all target languages. They refine the prompt template for maximum control.
        - **Frontend Team:** Builds the functional components for the Admin Dashboard (login, source management, log viewer) and creates the structure and styling for the embeddable Web Component.
    - **Phase 3: Integration, Testing, and Polish (Final 6-8 hours):** The focus shifts from building new features to connecting and refining the existing ones.
        - The Frontend team connects the Admin Dashboard and the chatbot widget to the live backend API endpoints.
        - The entire team participates in end-to-end testing, submitting queries in all supported languages (including Romanized Hindi) to verify the system works as expected.
        - Bugs are identified and fixed. The UI is polished.
        - The Team Lead oversees the creation of the README.md and prepares the final presentation and demo script.

Works cited

1. Flask vs fastapi : r/flask - Reddit, accessed September 18, 2025, https://www.reddit.com/r/flask/comments/13pyxie/flask_vs_fastapi/
2. FastAPI vs Flask: what's better for Python app development? - Imaginary Cloud, accessed September 18, 2025, https://www.imaginarycloud.com/blog/flask-vs-fastapi
3. Flask vs FastAPI: An In-Depth Framework Comparison | Better Stack Community, accessed September 18, 2025, https://betterstack.com/community/guides/scaling-python/flask-vs-fastapi/
4. FastAPI vs Flask: Key Differences, Performance, and Use Cases | Codecademy, accessed September 18, 2025, https://www.codecademy.com/article/fastapi-vs-flask-key-differences-performance-and-use-cases
5. ChromaDB vs FAISS: A Comprehensive Guide for Vector Search and AI Applications | by Mohamed Bakrey Mahmoud | Aug, 2025, accessed September 18, 2025, https://mohamedbakrey094.medium.com/chromadb-vs-faiss-a-comprehensive-guide-for-vector-search-and-ai-applications-39762ed1326f
6. Chroma vs FAISS - Zilliz, accessed September 18, 2025, https://zilliz.com/comparison/chroma-vs-faiss
7. ai4bharat/indic-bert · Hugging Face, accessed September 18, 2025, https://huggingface.co/ai4bharat/indic-bert
8. FAISS vs ChromaDB - Vector Search Comparison - YouTube, accessed September 18, 2025, https://www.youtube.com/watch?v=AUrJerdKMU8
9. Which database to use for semantic search? : r/LocalLLaMA - Reddit, accessed

September 18, 2025, https://www.reddit.com/r/LocalLLaMA/comments/13tp2sr/which_database_to_use_for_semantic_search/
10. LlamaIndex vs LangChain: Key Differences, Features & Use Cases, accessed September 18, 2025, https://www.openxcell.com/blog/llamaindex-vs-langchain/
11. Implementing Web Scraping in Python with BeautifulSoup - GeeksforGeeks, accessed September 18, 2025, https://www.geeksforgeeks.org/python/implementing-web-scraping-python-beautiful-soup/
12. Ultimate Guide to Web Scraping with Python Part 1: Requests and BeautifulSoup, accessed September 18, 2025, https://www.learndatasci.com/tutorials/ultimate-guide-web-scraping-w-python-requests-and-beautifulsoup/
13. Python Web Scraping Using Selenium and Beautiful Soup: A Step-by-Step Tutorial, accessed September 18, 2025, https://www.codecademy.com/article/web-scrape-with-selenium-and-beautiful-soup
14. Web Components - Web APIs | MDN, accessed September 18, 2025, https://developer.mozilla.org/en-US/docs/Web/API/Web_components
15. How to Implement RAG With Amazon Bedrock and LangChain | TigerData, accessed September 18, 2025, https://www.tigerdata.com/blog/how-to-implement-rag-with-amazon-bedrock-and-langchain
16. Build a Retrieval Augmented Generation (RAG) App: Part 1 - Python LangChain, accessed September 18, 2025, https://python.langchain.com/docs/tutorials/rag/
17. Vyakyarth - Multilingual Sentence Embedding Model - AIKosh - IndiaAI, accessed September 18, 2025, https://aikosh.indiaai.gov.in/home/models/details/vyakyarth_multilingual_sentence_embedding_model.html
18. AkshitaS/bhasha-embed-v0 · Hugging Face, accessed September 18, 2025, https://huggingface.co/AkshitaS/bhasha-embed-v0
19. pemistahl/lingua-py: The most accurate natural language ... - GitHub, accessed September 18, 2025, https://github.com/pemistahl/lingua-py
20. Tips to Write Effective LLM Prompts and Generate Multilingual Content - Lilt, accessed September 18, 2025, https://lilt.com/blog/tips-to-write-effective-llm-prompts-and-generate-multilingual-content
21. Multilingual Prompt Engineering for Semantic Alignment - Ghost, accessed September 18, 2025, https://latitude-blog.ghost.io/blog/multilingual-prompt-engineering-for-semantic-alignment/
22. Quick Start - FastAPI Admin, accessed September 18, 2025, https://fastapi-admin-docs.long2ice.io/getting_started/quickstart/
23. Awesome FastAPI | | Curated list of awesome lists | Project-Awesome.org, accessed September 18, 2025,

https://project-awesome.org/mjhea0/awesome-fastapi
24. FastAPI + Django Admin is the best practice?, accessed September 18, 2025, https://www.reddit.com/r/django/comments/1drj08o/fastapi_django_admin_is_the_best_practice/
25. Privacy Policy for Chatbots - TermsFeed, accessed September 18, 2025, https://www.termsfeed.com/blog/privacy-policy-chatbots/
26. Chatbot Privacy Policy - Mainstay, accessed September 18, 2025, https://mainstay.com/legal-chatbotprivacypolicy/
27. Chatbot Privacy Policy | StayDry Med Centers, accessed September 18, 2025, https://staydrycenters.com/chatbot-privacy-policy/