Build an AI Icebreaker Bot with LlamaIndex & IBM Granite

Estimated time needed: ***45*** minutes

Imagine you're heading to a big networking event, surrounded by potential employers and industry leaders. You want to make a great first impression, but you're struggling to come up with more than the usual, "What do you do?"

Now, picture having an AI-powered tool that does the research for you. You input a name, and within seconds, the bot, powered by **LlamaIndex** and **IBM watsonx**, searches LinkedIn, generating personalized icebreakers based on someone's career highlights, interests, and even fun facts. Instead of generic questions, you start with something unique and meaningful.

The AI icebreaker bot uses **natural language processing (NLP)** to create tailored conversation starters that help you stand out. By the end of this project, you'll have built a tool that helps make introductions smoother, more personal, and more memorable for networking, job interviews, or any social setting.

After completing this lab, you will be able to:

- Understand how to use **LlamaIndex** and **IBM watsonx** for personalized information retrieval.
- Learn how to search and extract relevant data from **LinkedIn** for icebreaker generation.
- Develop skills in using **AI and natural language processing (NLP)** to generate customized conversation starters based on a person's online presence.
- Gain practical experience in applying **AI-powered tools** to automate social interactions, making introductions more engaging and memorable.

Disclaimer: As of February 2025, Proxycurl has been discontinued, so the online approach requiring a valid Proxycurl API key is no longer functional (please disregard the video demonstrations that still reference this option). However, the offline option using mock data remains fully functional. The remainder of the lab still effectively demonstrates how to build a RAG application with LlamaIndex and IBM Granite, including the option to integrate an API. Please use the mock data to complete the lab. We have retained the code sections demonstrating API usage, should you wish to substitute Proxycurl with another service for your own development purposes.

A quick look at AI Icebreaker bot

What Does the AI Icebreaker bot do?

The LinkedIn Icebreaker Bot is an AI-powered tool that generates personalized conversation starters from LinkedIn profiles. Using IBM watsonx and LlamaIndex, it extracts profile data (via built-in mock data), processes it through a RAG pipeline, and produces tailored insights about a person's career. With the convenient mock data option, you can demonstrate the bot's capabilities without an API key - perfect for testing or presentations. Simply select "Use Mock Data" in the interface to instantly analyze a pre-loaded professional profile.

Available as both a command-line tool and web interface, the bot helps you make meaningful professional connections by replacing generic small talk with relevant, personalized conversation starters based on someone's actual experience and achievements.

Tips for the best experience - please read!

Keep the following tips handy and refer to them at any point of confusion throughout the tutorial. Do not worry if they seem irrelevant now. We will go through everything step by step later.

- At any point throughout the project, if you are lost, click on **Table of Contents** icon on the top left of the page and navigate to your desired content.
- Whenever you make changes to a file, be sure to save your work. Cloud IDE automatically saves any changes you make to your files immediately. You can also save from the toolbar.
- At the end of each section, you will be given the fully updated script for that part. And at the end of the project, you will be prompted to pull the complete codebase of the project as well.
- For running the application always ensure `app.py` is running in the background before opening the Web Application.
- You run a code block by clicking on the >_ button on bottom right.

  ```
  python app.py
  ```

- Always ensure that your current directory is `/home/project/icebreaker`. If you are not in the `icebreaker` folder, certain code files may fail to run. Use the `cd` command to navigate to the correct location.
- Make sure you are accessing the application through port `5000`. Clicking on the purple Web Application button will run the app through port 5000 automatically.

Web Application

- If you get an error about not being able to access another port (for example, 8888) just `refresh` the app by clicking on the small refresh icon. In case of other errors related to the server, simply refresh the page as well.
- To stop execution of `app.py` in addition to closing the application tab, hit `Ctrl+C` in terminal.
- If you encounter an error running the application or after you entered your desired keyword, try refreshing the app using the button on the top of the application's page. You can try inputting a different query too.
- Typically, using the models provided by Watsonx.ai would require Watsonx credentials, including an API key and a project ID. However, in this lab, these credentials are not needed.

Setting up your development environment

Before we dive into development, let's set up your project environment in the Cloud IDE. This environment is based on Ubuntu 22.04 and provides all the tools you need to build your AI-driven Gradio application.

Step 1: Create your project directory

1. Open the terminal in Cloud IDE and run:

   ```
   mkdir icebreaker
   wget -qO- https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/DPp7742-U7sqx0rrjMluVw/icebreaker.tar | tar -xf - -C icebreaker
   cd icebreaker
   ```

Once you are all set up, select File Explorer, then the `icebreaker` folder. You should have all the files structured as below.

2. Next, we will set up a virtual environment for the project and install all the required libraries.

Note, you can run the snippet directly by clicking on the run button >_. Approximately it will take **3-5 minutes** to install all the required libraries. Feel free to go grab a coffee while you wait!

This set of commands installs essential packages for building an IBM watsonx-powered retrieval-augmented generation (RAG) system. `ibm-watsonx-ai` provides seamless access to foundation models hosted on watsonx, while `llama-index` and its modular components — such as `llama-index-core`, `llama-index-llms-ibm`, and `llama-index-embeddings-ibm` — enable structured document indexing, integration with IBM models, and embedding generation. `llama-index-readers-web` adds the ability to ingest and process content directly from websites, and `llama-hub` offers a collection of pre-built connectors and loaders for diverse data sources. Finally, `requests` is used for making HTTP requests, supporting API communication and custom integrations. Together, these libraries create a flexible and production-ready pipeline for intelligent information retrieval and interaction.

```
python3.11 -m venv venv
source venv/bin/activate # activate venv
pip install -r requirements.txt
```

Now that your environment is set up, you're ready to start building your AI Icebreaker bot!

Part 1: Setting up your configuration

In this step, we'll set up the configuration file for our Icebreaker Bot. The `config.py` file centralizes all our settings in one place, making it easier to manage our application parameters.

Step 1: Open the config.py file

First, click the purple button below to open the `config.py` file. You'll see it already contains some boilerplate code and placeholders:

Open **config.py** in IDE

Step 2: Define the Prompt Templates

The most important part of this step is defining the prompt templates that will guide the LLM in generating responses. These templates use placeholders like `{context_str}` and `{query_str}` that will be replaced with actual content during execution. **The starter file already contains the prompt templates for you.**

Step 3: Understanding the Prompt Templates

Let's break down what each prompt template does:

Initial Facts Template:

- Purpose: Generates 3 interesting facts about a person's career or education
- Context: Uses the `{context_str}` placeholder where LinkedIn profile data will be inserted
- Instructions: Tells the LLM to answer based only on the provided context
- Output Format: Requests detailed answers about the person

User Question Template:

- Purpose: Answers specific questions about the LinkedIn profile
- Context: Uses the `{context_str}` placeholder for LinkedIn data
- Query: Uses the `{query_str}` placeholder for the user's question
- Instructions: Tells the LLM to answer based only on the context, and to say "I don't know" if the information isn't available

Step 4: Additional Configuration Options (Optional)

You might want to adjust some of the other settings:

- `CHUNK_SIZE`: Controls how large each text chunk is when splitting the LinkedIn data. If you want more granular retrieval, you could reduce this (e.g., to 300).
- `SIMILARITY_TOP_K`: Determines how many similar chunks to retrieve when answering queries. Increasing this could give more comprehensive but potentially noisier responses.

```
# Adjust these settings if needed
CHUNK_SIZE = 400  # Smaller chunks for more granular retrieval
SIMILARITY_TOP_K = 7  # Retrieve more chunks for more comprehensive answers
```

Step 5: Save Your Configuration

After making these changes, save the `config.py` file. Your configuration is now complete and ready to be used by the other modules in the application.

Testing Your Configuration

To verify your configuration is correct, you can create a simple test script. Click on the purple button below to open the `test_config.py` file.

Open **test_config.py** in IDE

Then, copy paste the code below into `test_config.py` and save the file.

```
import config
print("Initial Facts Template defined:", bool(config.INITIAL_FACTS_TEMPLATE))
print("User Question Template defined:", bool(config.USER_QUESTION_TEMPLATE))
print("Chunk Size:", config.CHUNK_SIZE)
print("Similarity Top K:", config.SIMILARITY_TOP_K)
```

Run this script to make sure all your settings are properly defined. You should see the configurations printed in your terminal. If any configuration is incorrect, go back and modify `config.py`.

```
python3.11 test_config.py
```

What We've Accomplished

In this step, we've:

- Set up our service connections
- Defined prompt templates that will guide our LLM's responses
- Configured retrieval parameters for our RAG workflow
- Created a centralized configuration that all our modules can use

Next, we'll implement the data extraction module to fetch LinkedIn profile data using these settings.

Part 2: Understanding the Core Concepts

1. Large Language Models and RAG (Optional - Skip if you already understand these concepts)

What are Large Language Models (LLM)?

Large language models are a type of artificial intelligence (AI) models that are trained on a large corpus of text data. LLMs are designed to generate human-like text responses to a wide range of questions. They are based on the Transformer architecture and are pretrained on a variety of language tasks to improve their performance.

What is IBM watsonx?

IBM watsonx is a suite of AI tools and services that are designed to help developers build and deploy AI-driven applications. watsonx provides a range of APIs and tools that make it easy to integrate AI capabilities into applications, including natural language processing, computer vision, and speech recognition. In this project, we are using two different foundational LLMs supported by watsonx.ai: `ibm/granite-3-2-8b-instruct` and `meta-llama/llama-3-3-70b-instruct`. You are free to switch between the models and compare their performances.

What is LlamaIndex?

LlamaIndex is an open-source data orchestration platform for creating large language model (LLM) applications. LlamaIndex is accessible in Python and TypeScript, and it uses a set of tools and features to ease the context augmentation process for generative AI (genAI) use cases via a Retrieval-Augmented (RAG) pipeline.

What is RAG?

While LLMs are built using extensive datasets, they don't naturally include your specific data. Retrieval-Augmented Generation (RAG) solves this issue by merging your data with the existing knowledge of LLMs. Throughout this guide, you'll frequently see mentions of RAG, as it's a critical technique used in query and chat engines, as well as agents, to boost their performance.

In a RAG setup, your data is first loaded, processed, and indexed for quick retrieval. When a user submits a query, the system searches the index to find the most relevant information from your data. This contextual information is then combined with the user's query and sent to the LLM, which generates a response based on this refined data.

source

What are RAG stages?

In the RAG framework, there are five key stages, though this lab will concentrate on the first four. These stages are essential for most large-scale applications you may develop. They include:

- **Loading**: This step involves importing your data into the workflow, regardless of the source — such as text files, PDFs, websites, databases, or APIs. LlamaHub provides various connectors to simplify this process.
- **Splitting**: After loading, data is divided into smaller, manageable chunks to improve the relevance of retrieval and the quality of responses from the LLM. Splitting strategies vary depending on the data type and use case — for example, using fixed-size windows, semantic segmentation, or recursive character splitting for text documents.
- **Indexing**: In this phase, a data structure is built for efficient querying. For LLMs, this often means creating vector embeddings, numerical representations that capture the meaning of the data, along with metadata strategies to ensure precise and context-aware data retrieval.
  - An `Index` is a key data structure in **LlamaIndex** that allows us to retrieve the relevant context in response to a user query. Indexing is vital because it enables quick and efficient access to relevant chunks of the data, which makes question-answering both faster and more accurate.
  - At a high level, `Indexes` are built from the `Nodes` created in the previous splitting stage. These indexes are then used to construct `Retrievers` and `Query Engines`, which power question-answering interactions and conversational experiences based on your data.
  - **What is vector embedding?** Before we can index the nodes, we need to convert them into vector representations. This process, known as `embedding`, allows the model to better understand the text data and compare its semantic similarity to user queries. Vectors enable efficient search and retrieval operations, as similar pieces of text will have similar vector representations.
  - To embed the nodes, we'll use an `IBM watsonx Embedding model`. Once the data is embedded, we store these vectors in a vector database for retrieval during query processing.

  source
- **Storing**: Once indexing is complete, it's important to save the index and its associated metadata to avoid re-indexing the data later.
- **Querying**: Based on the indexing strategy, there are various ways to perform queries using LLMs and LlamaIndex structures. This can include sub-queries, multi-step queries, or hybrid techniques.
- **Evaluation**: A crucial step in any workflow is evaluating the effectiveness of your approach. Evaluation provides objective metrics to measure the accuracy, relevance, and speed of query results when comparing different methods or making adjustments.

  source

In our Icebreaker Bot, the RAG workflow looks like this:

- We load LinkedIn profile data
- We convert it into chunks and generate embeddings
- We store these embeddings in a vector database
- When you ask a question, we retrieve the most relevant pieces of the profile
- We use an LLM to generate personalized answers based on this specific context

This approach ensures that our responses are grounded in the actual LinkedIn profile data rather than the LLM's general knowledge.

What is prompt engineering?

Prompt Engineering is the process of designing and refining the inputs (prompts) given to language models such as GPT to get desired outputs. It involves crafting questions, instructions, or context in a way that guides the model to generate accurate, relevant, or creative responses. Good prompt engineering can improve the quality, specificity, and usefulness of the generated text, making it a critical skill for leveraging large language models in various applications like chatbots, content generation, or data analysis.

2. LinkedIn Data Extraction

> Please note that while we present both options for extracting profile data in this step, only the mock data option will work in the current app. Nevertheless, we include the API integration example in case you wish to adapt the code for your own development purposes.

The Data Extraction Process

When extracting LinkedIn profile data, we're using a REST API to fetch structured information about a user's professional background. The process works as follows:

1. Prepare the request:

- Format the LinkedIn profile URL
- Set up authentication headers with your API key
- Configure parameters (like what data to include)

2. Send the API request:

- Make an HTTP GET request to an API's endpoint
- Pass the profile URL and other parameters

3. Process the response:

- Check if the request was successful (HTTP 200 status)
- Parse the JSON response data
- Clean the data by removing empty values and unwanted fields

4. Structure the data:

- The resulting JSON contains sections like:
  - Basic information (name, headline, location)
  - Work experience
  - Education history
  - Skills
  - Certifications
  - Accomplishments

The data extraction is the first step in our RAG pipeline, providing the raw material that we'll later split, index, and query.

Option for Mock Data vs. Real API Calls

Our application provides two options for working with LinkedIn profile data:

Using Mock Data:

- **Advantages:**
  - No API key required
  - No credit consumption
  - Consistent results for testing
  - Works offline
- **How it works:**
  - A pre-generated LinkedIn profile JSON is loaded from a URL
  - This data is treated the same as if it came from the API
  - All RAG processing steps remain the same

Using Real API Calls:

- **Advantages:**
  - Access to real, up-to-date LinkedIn profiles
  - Ability to analyze any public LinkedIn profile
  - More diverse and personalized results
- **How it works:**
  - Your API key authenticates requests
  - The API fetches real-time data from LinkedIn
  - Each request consumes certain amount of credits from your account

Choosing the Right Approach:

Use mock data for:

- Learning and development
- Testing your implementation
- Demonstrations without API costs

Use real API calls for:

- Analyzing specific profiles of interest
- Production applications
- Situations where up-to-date data is crucial

In the next parts of this tutorial, we'll implement both approaches, allowing you to switch between them easily based on your needs.

Part 3: Implementing LinkedIn Profile Data Extraction

> Please note that while we present both options for extracting profile data in this step, as mentione dpreviously, only the mock data option will work in the current app. Nevertheless, we include the API integration example in case you wish to adapt the code for your own development purposes.

In this step, we'll implement the data extraction module that fetches LinkedIn profile data either through API or by using mock data.

Step 1: Examining the data_extraction.py Starter File

Let's first look at the starter file structure. Click on the purple button below to open `data_extraction.py`.

Open **data_extraction.py** in IDE

In this file, we have to implement the `extract_linkedin_profile` function which retrieves and processes LinkedIn profile data, either by fetching it from the API using a provided URL and API key, or by loading pre-configured mock data when the mock option is enabled.

Exercise: Take a moment to think about how you would implement this function and create the pseudocode for it before moving on to the next step.

Step 2: Implement the extract_linkedin_profile Function

Now, let's implement the function to handle both mock data and real API calls. Copy-paste the code below into `extract_linkedin_profile` function and save your files.

```
def extract_linkedin_profile(
    linkedin_profile_url: str,
    api_key: Optional[str] = None,
    mock: bool = False
) -> Dict[str, Any]:
    """Extract LinkedIn profile data using API or loads a premade JSON file.

    Args:
        linkedin_profile_url: The LinkedIn profile URL to extract data from.
        api_key: API key. Required if mock is False.
        mock: If True, loads mock data from a premade JSON file instead of using the API.

    Returns:
        Dictionary containing the LinkedIn profile data.
    """
    start_time = time.time()

    try:
        if mock:
            logger.info("Using mock data from a premade JSON file...")
            mock_url = config.MOCK_DATA_URL
            response = requests.get(mock_url, timeout=30)
        else:
            # Ensure API key is provided when mock is False
            if not api_key:
                raise ValueError("API key is required when mock is set to False.")

            logger.info("Starting to extract the LinkedIn profile...")
            # Set up the API endpoint and headers
            api_endpoint = "https://nubela.co/proxycurl/api/v2/linkedin"
            headers = {
                "Authorization": f"Bearer {api_key}"
            }
            # Prepare parameters for the request
            params = {
                "url": linkedin_profile_url,
                "fallback_to_cache": "on-error",
                "use_cache": "if-present",
                "skills": "include",
                "inferred_salary": "include",
                "personal_email": "include",
                "personal_contact_number": "include"
            }
            logger.info(f"Sending API request to ProxyCurl at {time.time() - start_time:.2f} seconds...")
            # Send API request
            response = requests.get(api_endpoint, headers=headers, params=params, timeout=10)

        logger.info(f"Received response at {time.time() - start_time:.2f} seconds...")
        # Check if response is successful
        if response.status_code == 200:
            try:
                # Parse the JSON response
                data = response.json()

                # Clean the data, remove empty values and unwanted fields
                data = {
                    k: v
                    for k, v in data.items()
                    if v not in ([], "", None) and k not in ["people_also_viewed", "certifications"]
                }
                # Remove profile picture URLs from groups to clean the data
                if data.get("groups"):
                    for group_dict in data.get("groups"):
                        group_dict.pop("profile_pic_url", None)
                return data
            except ValueError as e:
                logger.error(f"Error parsing JSON response: {e}")
                logger.error(f"Response content: {response.text[:200]}...")  # Print first 200 chars
                return {}
        else:
            logger.error(f"Failed to retrieve data. Status code: {response.status_code}")
            logger.error(f"Response: {response.text}")
            return {}

    except Exception as e:
        logger.error(f"Error in extract_linkedin_profile: {e}")
        return {}
```

How this implementation works:

Let's break down what our implementation does:

Handling Mock Data vs. API Calls

If `mock=True`, we load data from a predefined URL in the config
If `mock=False`, we prepare an API request to the API provider with the provided API key

API Request Configuration

- We set up the API endpoint
- We configure the authorization header with the API key
- We specify parameters like:
  - The LinkedIn profile URL
  - Cache settings to reduce API usage
  - Additional data to include (skills, salary info, etc.)

Response Processing

- We check for a successful status code (200)
- We parse the JSON response
- We clean the data by:
  - Removing empty values (empty lists, empty strings, None)
  - Excluding unwanted fields like "people_also_viewed"
  - Removing profile picture URLs from groups

Error Handling

- We wrap everything in try/except blocks to gracefully handle errors
- We log detailed error messages to help debugging
- We return an empty dictionary if anything goes wrong

Click the button below to see the fully updated `data_extraction.py`.

▶ Click to see the solution

What We've Accomplished

In this step, we've:

- Implemented the LinkedIn profile data extraction function
- Added support for both mock data and real API calls
- Added robust error handling and logging
- Cleaned the response data for easier processing

This function forms the foundation of our RAG pipeline, providing the raw data that we'll process, index, and query in the subsequent steps.

Part 4: Implementing Data Processing for RAG

In this step, we'll implement the data processing module that transforms raw LinkedIn profile data into a format suitable for retrieval and question answering. This involves splitting the data into chunks, creating vector embeddings, and storing them in a searchable database.

Step 1: Examining the data_processing.py Starter File

Let's first examine the starter file structure. Click on the purple button below to open `data_processing.py`.

Open **data_processing.py** in IDE

We need to implement three functions:

1. `split_profile_data`: Divides the profile data into manageable chunks
2. `create_vector_database`: Creates a vector index from the chunks
3. `verify_embeddings`: Ensures all embeddings were created properly

Exercise: Take a moment to think about how you would implement these functions and create the pseudocode for them before moving on to the next step.

Step 2: Implement the split_profile_data Function

In the traditional RAG framework, after the Loading stage, the next step is **Splitting**. This involves dividing the data into smaller, manageable chunks that can be efficiently embedded into a vector database and later retrieved for answering user queries.

source

First, let's implement the function to split the profile data into nodes. Copy-paste the code below into `split_profile_data` function and save your files.

```
def split_profile_data(profile_data: Dict[str, Any]) -> List:
    """Splits the LinkedIn profile JSON data into nodes.

    Args:
        profile_data: LinkedIn profile data dictionary.
    Returns:
        List of document nodes.
    """
    try:
        # Convert the profile data to a JSON string
        profile_json = json.dumps(profile_data)
        # Create a Document object from the JSON string
        document = Document(text=profile_json)
        # Split the document into nodes using SentenceSplitter
        splitter = SentenceSplitter(chunk_size=config.CHUNK_SIZE)
        nodes = splitter.get_nodes_from_documents([document])

        logger.info(f"Created {len(nodes)} nodes from profile data")
        return nodes
    except Exception as e:
        logger.error(f"Error in split_profile_data: {e}")
        return []
```

This function:

- Converts the profile data dictionary to a JSON string
- Creates a Document object from the string
- Uses a SentenceSplitter to divide the text into chunks of size specified in config
- Returns the resulting nodes

In our project, we need to split the LinkedIn profile data (scraped from ProxyCurl) into smaller segments or nodes. These nodes represent logical chunks of information that can later be queried.

Since the LinkedIn profile data is a JSON file, we first convert it into a textual format, and then split the text into smaller nodes of approximately 500 characters each.

This splitting process ensures that our data is indexed in manageable pieces, making it easier for the model to retrieve relevant information based on user queries.

Step 3: Implement the create_vector_database Function

In this step, we move to **Indexing** and **Storing** these nodes for efficient retrieval. This step is critical for building a foundation for RAG, where we retrieve relevant data to answer user queries.

source

Next, let's implement the function to create a vector database. Copy-paste the code below into `create_vector_database` function and save your files.

```
    def create_vector_database(nodes: List) -> Optional[VectorStoreIndex]:
        """Stores the document chunks (nodes) in a vector database.

        Args:
            nodes: List of document nodes to be indexed.

        Returns:
            VectorStoreIndex or None if indexing fails.
        """
        try:
            # Get the embedding model
            embedding_model = create_watsonx_embedding()
            # Create a VectorStoreIndex from the nodes
            index = VectorStoreIndex(
                nodes=nodes,
                embed_model=embedding_model,
                show_progress=True
            )

            logger.info("Vector database created successfully")
            return index
        except Exception as e:
            logger.error(f"Error in create_vector_database: {e}")
            return None
```

This function:

- Gets an embedding model from the `llm_interface` module
- Creates a `VectorStoreIndex` with the nodes and embedding model
- Returns the index or None if an error occurs

Step 4: Implement the verify_embeddings Function

Finally, let's implement the function to verify the embeddings. Copy-paste the code below into `verify_embeddings` function and save your files.

```
def verify_embeddings(index: VectorStoreIndex) -> bool:
    """Verify that all nodes have been properly embedded.

    Args:
        index: VectorStoreIndex to verify.

    Returns:
        True if all embeddings are valid, False otherwise.
    """
    try:
        vector_store = index._storage_context.vector_store
        node_ids = list(index.index_struct.nodes_dict.keys())
        missing_embeddings = False
        for node_id in node_ids:
            embedding = vector_store.get(node_id)
            if embedding is None:
                logger.warning(f"Node ID {node_id} has a None embedding.")
                missing_embeddings = True
            else:
                logger.debug(f"Node ID {node_id} has a valid embedding.")

        if missing_embeddings:
            logger.warning("Some node embeddings are missing")
            return False
        else:
            logger.info("All node embeddings are valid")
            return True
    except Exception as e:
        logger.error(f"Error in verify_embeddings: {e}")
        return False
```

This function:

- Gets the vector store from the index
- Gets the list of node IDs
- Checks if each node has a valid embedding
- Returns True if all embeddings are valid, False otherwise

Click the below button to see the fully updated `data_processing.py`.

▶ Click to see the solution

What We've Accomplished

In this step, we've:

- Implemented the function to split LinkedIn profile data into manageable chunks
- Implemented the function to create a vector database from these chunks
- Implemented the function to verify the embeddings
- Created a test script to validate our implementation

These functions form the core of the indexing phase of our RAG pipeline. With the data now properly processed and indexed, we can move on to implementing the query engine that will generate responses based on this indexed data.

Part 5: Implementing the interface to IBM watsonx.ai's language and embedding models

In this section, we'll implement the interface to IBM watsonx.ai's language and embedding models. This is a crucial component of our Icebreaker Bot as it handles the connection to the AI models that power our application.

Step 1: Examining the llm_interface.py Starter File

Let's first examine the starter file structure. Click on the purple button below to open `llm_interface.py`.

Open **llm_interface.py** in IDE

In this file, we have to implement 3 functions:

- `create_watsonx_embedding()`: This function creates the embedding model that converts text into vector representations. It should return a WatsonxEmbeddings instance configured with the correct model ID, URL, and project ID from our config file. This embedding model will be used to transform LinkedIn profile data chunks into vectors for semantic search.

- `create_watsonx_llm()`: This function creates the language model that generates responses to user queries. It should return a WatsonxLLM instance configured with parameters that control the generation process, such as temperature (for randomness), token limits, and decoding methods. This LLM will be responsible for generating interesting facts and answering questions about LinkedIn profiles.
- `change_llm_model()`: This utility function allows us to dynamically switch between different language models at runtime. It should update the LLM model ID in our config and log the change. This flexibility enables experimenting with different models to compare their performance on icebreaker generation tasks.

Exercise: Take a moment to think about how you would implement these functions and create the pseudocode for them before moving on to the next step.

Step 2: Implement the Embedding Model Function

Next, we'll implement the function that creates our embedding model. Copy-paste the code below into `create_watsonx_embedding` function and save your files.

```
def create_watsonx_embedding() -> WatsonxEmbeddings:
    """Creates an IBM Watsonx Embedding model for vector representation.

    Returns:
        WatsonxEmbeddings model.
    """
    watsonx_embedding = WatsonxEmbeddings(
        model_id=config.EMBEDDING_MODEL_ID,
        url=config.WATSONX_URL,
        project_id=config.WATSONX_PROJECT_ID,
        truncate_input_tokens=3,
    )
    logger.info(f"Created Watsonx Embedding model: {config.EMBEDDING_MODEL_ID}")
    return watsonx_embedding
```

This function creates an instance of WatsonxEmbeddings which we'll use to convert text chunks from LinkedIn profiles into vector representations. These vectors capture the semantic meaning of the text, allowing us to find the most relevant information when answering user queries.

Key Parameters:

- `model_id`: Specifies which embedding model to use (defined in config.py)
- `url`: The endpoint URL for watsonx.ai services
- `project_id`: The project ID for watsonx.ai
- `truncate_input_tokens`: Controls how the model handles tokens that exceed the model's context window

Step 3: Implement the Language Model Function

Now, let's implement the function that creates our language model. Copy-paste the code below into `create_watsonx_llm` function and save your files.

```
def create_watsonx_llm(
    temperature: float = config.TEMPERATURE,
    max_new_tokens: int = config.MAX_NEW_TOKENS,
    decoding_method: str = "sample"
) -> WatsonxLLM:
    """Creates an IBM Watsonx LLM for generating responses.

    Args:
        temperature: Temperature for controlling randomness in generation (0.0 to 1.0).
        max_new_tokens: Maximum number of new tokens to generate.
        decoding_method: Decoding method to use (sample, greedy).

    Returns:
        WatsonxLLM model.
    """
    additional_params = {
        "decoding_method": decoding_method,
        "min_new_tokens": config.MIN_NEW_TOKENS,
        "top_k": config.TOP_K,
        "top_p": config.TOP_P,
    }

    watsonx_llm = WatsonxLLM(
        model_id=config.LLM_MODEL_ID,
        url=config.WATSONX_URL,
        project_id=config.WATSONX_PROJECT_ID,
        temperature=temperature,
        max_new_tokens=max_new_tokens,
        additional_params=additional_params,
    )
    logger.info(f"Created Watsonx LLM model: {config.LLM_MODEL_ID}")
    return watsonx_llm
```

This function creates an instance of WatsonxLLM that we'll use to generate responses to user queries. It connects to IBM watsonx.ai and configures the language model with the appropriate parameters for our use case.

Key Parameters:

- `temperature`: Controls the randomness of the generated text. Lower values (like 0.0) make responses more deterministic and focused, while higher values introduce more creativity and variation.
- `max_new_tokens`: Limits the length of the generated response.
- `decoding_method`: Determines how the model selects the next token. Options include:
    - "sample": Randomly samples from the probability distribution (more creative)
    - "greedy": Always selects the most likely next token (more deterministic)
- `additional_params`: Additional configuration for the model:
    - `min_new_tokens`: Minimum number of tokens to generate
    - `top_k`: Limits token selection to the top k most probable tokens
    - `top_p`: Uses nucleus sampling to select from tokens that comprise the top p probability mass

Step 4: Implement Model Switching Functionality

Finally, we'll implement a function that allows us to switch between different LLM models. Copy-paste the code below into `change_llm_model` function and save your files.

```
def change_llm_model(new_model_id: str) -> None:
    """Change the LLM model to use.

    Args:
        new_model_id: New LLM model ID to use.
    """
    global config
    config.LLM_MODEL_ID = new_model_id
    logger.info(f"Changed LLM model to: {new_model_id}")
```

This function allows users to experiment with different language models. It updates the `LLM_MODEL_ID` in the config module to use a different model for generating responses. In this project, we are using two models: `ibm/granite-3-2-8b-instruct` and `meta-llama/llama-3-3-70b-instruct`. If there are more models on watsonx.ai you want to explore, feel free to update the list to your preferences. See this page for currently supported foundational models on watsonx.ai.

Click the below button to see the fully updated `llm_interface.py`.

▶ Click to see the solution

What We've Accomplished

In this step, we've:

- Set up the interface to both IBM watsonx embedding and language models
- Configured key generation parameters including temperature and decoding methods
- Implemented a model switching function for easy experimentation
- Created a clean abstraction layer that hides the complexity of connecting to AI services

This LLM interface serves as the intelligence layer of our Icebreaker Bot, enabling both the semantic understanding of LinkedIn profiles through embeddings and the generation of personalized icebreakers through the language model.

Part 6: Creating the query engine

Now that we have our LinkedIn data indexed and our LLM interface configured, we need to create the query engine that will power our Icebreaker Bot's ability to generate insights and answer questions. This module is responsible for retrieving relevant information from our vector index and formatting it into engaging responses.

What is a Query Engine?

In the context of our RAG (Retrieval-Augmented Generation) system, a query engine is responsible for:

- Retrieving relevant information from our vector index based on a query
- Formatting the retrieved information into a prompt for the LLM
- Generating coherent and accurate responses based on the retrieved context

  source

Let's implement our query engine step by step.

Step 1: Examining the query_engine.py Starter File

Let's first examine the starter file structure. Click on the purple button below to open `query_engine.py`.

Open **query_engine.py** in IDE

In this file, we have to implement 2 functions:

- `generate_initial_facts(index)`: This function creates engaging conversation starters based on a person's LinkedIn profile. It should create a watsonx LLM with appropriate parameters for fact generation, construct a prompt template using our predefined template from the config, build a query engine that combines the index with the LLM and prompt, and then execute a query to generate three interesting facts about the person.
- `answer_user_query(index, user_query)`: This function powers the interactive Q&A capability of our bot. It should create a watsonx LLM optimized for question answering (using different parameters than fact generation), retrieve the most relevant nodes from our vector index based on the user's query, build a context string from these nodes, create a query engine with our question-answering prompt template, and execute the query to generate a precise answer. This function implements the complete RAG (Retrieval-Augmented Generation) workflow that enables accurate, context-aware responses about the LinkedIn profile.

Exercise: Take a moment to think about how you would implement these functions and create the pseudocode for them before moving on to the next step.

Step 2: Implement the Fact Generation Function

First, let's implement the function that generates interesting facts about a person based on their LinkedIn profile. Copy-paste the code below into `generate_initial_facts` function and save your files.

```
def generate_initial_facts(index: VectorStoreIndex) -> str:
    """Generates interesting facts about the person\'s career or education.

    Args:
        index: VectorStoreIndex containing the LinkedIn profile data.

    Returns:
        String containing interesting facts about the person.
    """
    try:
        # Create LLM for generating facts
        watsonx_llm = create_watsonx_llm(
            temperature=0.0,
            max_new_tokens=500,
            decoding_method="sample"
        )

        # Create prompt template
        facts_prompt = PromptTemplate(template=config.INITIAL_FACTS_TEMPLATE)

        # Create query engine
        query_engine = index.as_query_engine(
            streaming=False,
            similarity_top_k=config.SIMILARITY_TOP_K,
            llm=watsonx_llm,
            text_qa_template=facts_prompt
        )

        # Execute the query
        query = "Provide three interesting facts about this person\'s career or education."
        response = query_engine.query(query)

        # Return the facts
        return response.response
    except Exception as e:
        logger.error(f"Error in generate_initial_facts: {e}")
        return "Failed to generate initial facts."
```

How This Function Works:

1. LLM Configuration: We create a language model instance with specific parameters:

- `temperature=0.0`: Makes the output more deterministic and focused on factual information
- `max_new_tokens=500`: Allows for a detailed response
- `decoding_method="sample"`: Introduces some variation while maintaining factuality

2. Prompt Template: We use a predefined template (from config.py) that instructs the LLM how to generate facts based on the profile data.

3. Query Engine Creation: We create a query engine from our index with these parameters:

- `streaming=False`: Wait for the full response rather than streaming it
- `similarity_top_k=config.SIMILARITY_TOP_K`: Retrieve the most relevant chunks from the profile
- `llm=watsonx_llm`: Use our configured language model
- `text_qa_template=facts_prompt`: Use our fact generation prompt template

4. Query Execution: We send a simple query to the engine and return the response

5. Error Handling: We catch any exceptions and return a friendly error message

Step 3: Implement the Question Answering Function

Now, let's implement the function that answers specific user questions about the LinkedIn profile. Copy-paste the code below into `answer_user_query` function and save your files.

```
def answer_user_query(index: VectorStoreIndex, user_query: str) -> Any:
    """Answers the user's question using the vector database and the LLM.

    Args:
        index: VectorStoreIndex containing the LinkedIn profile data.
        user_query: The user's question.

    Returns:
        Response object containing the answer to the user's question.
    """
    try:
        # Create LLM for answering questions
        watsonx_llm = create_watsonx_llm(
            temperature=0.0,
            max_new_tokens=250,
            decoding_method="greedy"
        )

        # Create prompt template
        question_prompt = PromptTemplate(template=config.USER_QUESTION_TEMPLATE)

        # Retrieve relevant nodes
        base_retriever = index.as_retriever(similarity_top_k=config.SIMILARITY_TOP_K)
        source_nodes = base_retriever.retrieve(user_query)

        # Build context string
        context_str = "\n\n".join([node.node.get_text() for node in source_nodes])

        # Create query engine
        query_engine = index.as_query_engine(
            streaming=False,
            similarity_top_k=config.SIMILARITY_TOP_K,
            llm=watsonx_llm,
            text_qa_template=question_prompt
        )

        # Execute the query
        answer = query_engine.query(user_query)
        return answer
    except Exception as e:
        logger.error(f"Error in answer_user_query: {e}")
        return "Failed to get an answer."
```

How This Function Works:

1. LLM Configuration: We create a language model instance with specific parameters:

- `temperature=0.0`: Makes the output more deterministic and focused on factual information
- `max_new_tokens=250`: Keeps answers concise
- `decoding_method="greedy"`: Always selects the most likely next token for maximum accuracy

2. Prompt Template: We use a question-answering template that instructs the LLM to answer based on the provided context.

3. Relevant Node Retrieval: This is a key step where we:

- Create a retriever from our index
- Retrieve the most relevant nodes based on the user's query
- Build a context string from these nodes

4. Query Engine Creation: Similar to the fact generation, but optimized for question answering

5. Query Execution: We send the user's query to the engine and return the full response object

6. Error Handling: We catch any exceptions and return a friendly error message

Click the below button to see the fully updated `query_engine.py`.

▶ Click to see the solution

What We've Accomplished

In this step, we've:

- Implemented the core query engine for our Icebreaker Bot
- Created functions for generating interesting facts and answering specific questions
- Set up the RAG pipeline to retrieve relevant context before generating responses
- Added robust error handling and logging
- Configured different LLM parameters for different types of queries

This query engine is the heart of our application, connecting the indexed LinkedIn data with the LLM to generate personalized, accurate responses. In the next section, we'll integrate these components into a complete application with a user interface.

Part 7: Building command-line interface (CLI) application

> Please note that while we present both options for extracting profile data in this step, as mentione dpreviously, only the mock data option will work in the current app. Nevertheless, we include the API integration example in case you wish to adapt the code for your own development purposes.

Now that we've built all the key components of our Icebreaker Bot, it's time to bring everything together in a command-line interface (CLI) application. This will allow users to interact with the bot directly from the terminal, providing a LinkedIn URL and asking questions about the profile.

Step 1: Examining the main.py Starter File

Let's first examine the starter file structure. Click on the purple button below to open `main.py`.

Open **main.py** in IDE

In the main.py module, we need to implement two central functions that orchestrate the entire Icebreaker Bot workflow and provide the user interface:

- `process_linkedin(linkedin_url, api_key, mock)`: This function serves as the core orchestrator of our application. It should coordinate the entire RAG pipeline by first extracting LinkedIn profile data (either via API or from mock data), then splitting this data into manageable nodes, creating a vector database from these nodes, verifying that embeddings were properly created, generating initial conversation starters about the person, and finally launching the interactive chatbot interface. This function handles the end-to-end workflow from raw LinkedIn URL to a fully functional conversational assistant.
- `chatbot_interface(index)`: This function creates a user-friendly command-line interface for interacting with our bot. It should display clear instructions to the user, create a loop that processes user questions about the LinkedIn profile, retrieve answers using our query engine, display these answers in a conversational format, and provide a clean exit mechanism.

The main.py file also includes a ready-to-use `main()` function that handles command-line arguments, providing flexibility in how the bot is used. Once we implement the missing functions, uncommenting the process_linkedin call will activate the complete application.

Exercise: Take a moment to think about how you would implement these functions and create the pseudocode for them before moving on to the next step.

Step 2: Implement the Process LinkedIn Function

This is the core function that orchestrates the entire workflow. Copy-paste the code below into `process_linkedin` function and save your files.

```python
def process_linkedin(linkedin_url, api_key=None, mock=False):
    """
    Processes a LinkedIn URL, extracts data from the profile, and interacts with the user.
    Args:
        linkedin_url: The LinkedIn profile URL to extract or load mock data from.
        api_key: ProxyCurl API key. Required if mock is False.
        mock: If True, loads mock data from a premade JSON file instead of using the API.
    """
    try:
        # Extract the profile data
        profile_data = extract_linkedin_profile(linkedin_url, api_key, mock=mock)

        if not profile_data:
            logger.error("Failed to retrieve profile data.")
            return

        # Split the data into nodes
        nodes = split_profile_data(profile_data)

        # Store in vector database
        vectordb_index = create_vector_database(nodes)

        if not vectordb_index:
            logger.error("Failed to create vector database.")
            return

        # Verify embeddings
        if not verify_embeddings(vectordb_index):
            logger.warning("Some embeddings may be missing or invalid.")

        # Generate and display the initial facts
        initial_facts = generate_initial_facts(vectordb_index)

        print("\nHere are 3 interesting facts about this person:")
        print(initial_facts)

        # Start the chatbot interface
        chatbot_interface(vectordb_index)

    except Exception as e:
        logger.error(f"Error occurred: {str(e)}")
```

How This Function Works:

1. Extraction: It calls our extract_linkedin_profile function to get the profile data
2. Processing: It splits the data into nodes and creates a vector database
3. Initial Facts: It generates 3 interesting facts as a starting point
4. User Interface: It prints the facts and launches the chatbot interface
5. Logging and Timing: It logs each step and tracks how long each part of the process takes
6. Error Handling: It catches and reports any errors that occur during processing

Step 3: Implement the Chatbot Interface

Next, let's create a simple text-based interface for interacting with the bot. Copy-paste the code below into `chatbot_interface` function and save your files.

```
def chatbot_interface(index):
    """
    Provides a simple chatbot interface for user interaction.

    Args:
        index: VectorStoreIndex containing the LinkedIn profile data.
    """
    print("\nYou can now ask more in-depth questions about this person. Type 'exit', 'quit', or 'bye' to quit.")

    while True:
        user_query = input("You: ")
        if user_query.lower() in ['exit', 'quit', 'bye']:
            print("Bot: Goodbye!")
            break

        print("Bot is typing...", end='')
        sys.stdout.flush()
        time.sleep(1)  # Simulate typing delay
        print('\r', end='')

        response = answer_user_query(index, user_query)
        print(f"Bot: {response.response.strip()}\n")
```

How This Function Works:

1. User Input: It prompts the user for questions about the profile
2. Query Processing: It sends user queries to our answer_user_query function
3. Response Display: It shows the bot's responses in a conversational format
4. Exit Commands: It allows the user to end the conversation with simple commands
5. Error Handling: It handles interruptions and errors

Step 4: Create the Main Entry Point

Finally, let's create the main function that ties everything together. Copy-paste the code below into `main` function and save your files.

```
def main():
    """Main function to run the Icebreaker Bot."""
    parser = argparse.ArgumentParser(description='Icebreaker Bot - LinkedIn Profile Analyzer')
    parser.add_argument('--url', type=str, help='LinkedIn profile URL')
    parser.add_argument('--api-key', type=str, help='ProxyCurl API key')
    parser.add_argument('--mock', action='store_true', help='Use mock data instead of API')
    parser.add_argument('--model', type=str, help='LLM model to use (e.g., "ibm/granite-3-2-8b-instruct")')

    args = parser.parse_args()

    # Use command line arguments or prompt user for input
    linkedin_url = args.url or input("Enter LinkedIn profile URL (or press Enter to use mock data): ")
    use_mock = args.mock or not linkedin_url

    if args.model:
        from modules.llm_interface import change_llm_model
        change_llm_model(args.model)

    api_key = args.api_key or config.PROXYCURL_API_KEY

    if not use_mock and not api_key:
        api_key = input("Enter ProxyCurl API key: ")

    # Use a default URL for mock data if none provided
    if use_mock and not linkedin_url:
        linkedin_url = "https://www.linkedin.com/in/leonkatsnelson/"
        process_linkedin(linkedin_url, api_key, mock=use_mock)
if __name__ == "__main__":
    main()
```

How This Function Works:

1. Argument Parsing: It processes command-line arguments
2. Interactive Prompts: If required information is missing, it prompts the user
3. Profile Processing: It calls our process_linkedin function with the provided arguments

Click the below button to see the fully updated `main.py`.

▶ Click to see the solution

Testing the Complete CLI Workflow

> Please note that while we present both options for extracting profile data in this step, as mentione dpreviously, only the mock data option will work in the current app. Nevertheless, we include the API integration example in case you wish to adapt the code for your own development purposes.

To test our CLI application, we can run it with various combinations of arguments:

1. With a LinkedIn URL and API key: (replace with a real LinkedIn URL and your API key):

```
python main.py --url https://www.linkedin.com/in/johndoe/ --api-key YOUR_API_KEY
```

2. With mock data:

```
python main.py --mock
```

3. Interactively (with prompts):

```
python main.py
```

What We've Accomplished

In this step, we've:

- Implemented a comprehensive main application that orchestrates all our modules
- Created a function that processes LinkedIn profiles from start to finish
- Built a simple but effective chatbot interface for user interaction
- Added command-line argument parsing for flexible usage

- Included timing and logging for better debugging and monitoring
- Added robust error handling throughout the application

The main.py file serves as the glue that connects all our components, providing a seamless user experience from profile URL to personalized icebreakers and conversations.

Next Steps

With our CLI application complete, users can now interact with the Icebreaker Bot directly from the terminal. However, a web-based interface would make the bot even more accessible and user-friendly. In the next section, we'll build a Gradio web interface that provides all the same functionality with a graphical user interface.

Part 8: Building a user-friendly web interface with Gradio

Now that we've built our command-line interface, let's create a more user-friendly web interface using Gradio. This will make our Icebreaker Bot accessible to users who prefer graphical interfaces and provide a more polished experience.

> Please note that while we present both options for extracting profile data in this step, as mentione dpreviously, only the mock data option will work in the current app. Nevertheless, we include the API integration example in case you wish to adapt the code for your own development purposes.

Step 1: Examining the app.py Starter File

Let's first examine the starter file structure. Click on the purple button below to open `app.py`.

Open **app.py** in IDE

The app.py module creates a web-based interface for our Icebreaker Bot using Gradio, a Python library for building web interfaces for machine learning models. This interface will:

- Allow users to input LinkedIn profile URLs
- Process profiles and display interesting facts
- Provide a chat interface for asking questions about the profile
- Support switching between different LLM models

Let's implement this step by step.

Step 2: Import the Necessary Libraries and Our Modules

```
"""Gradio web interface for the Icebreaker Bot."""
import os
import sys
import logging
import uuid
import gradio as gr
from modules.data_extraction import extract_linkedin_profile
from modules.data_processing import split_profile_data, vector_database
from modules.llm_interface import change_llm_model
from modules.query_engine import generate_initial_facts, answer_user_query
import config
# Set up logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.StreamHandler(stream=sys.stdout)
    ]
)
logger = logging.getLogger(__name__)
# Dictionary to store active conversations
active_indices = {}
```

These imports give us access to:

- Standard Python libraries for unique IDs and logging
- Gradio for building the web interface
- Our custom modules for the core functionality
- A dictionary to store active conversations across sessions

Step 3: Implement the Profile Processing Function

Next, let's implement the function that processes LinkedIn profiles. Copy-paste the code below into `process_profile` function and save your files.

```
def process_profile(linkedin_url, api_key, use_mock, selected_model):
    """Process a LinkedIn profile and generate initial facts.

    Args:
        linkedin_url: LinkedIn profile URL to process.
        api_key: ProxyCurl API key.
        use_mock: Whether to use mock data.
        selected_model: LLM model to use.

    Returns:
        Initial facts about the profile and a session ID for this conversation.
    """
    try:
        # Change LLM model if needed
        if selected_model != config.LLM_MODEL_ID:
            change_llm_model(selected_model)

        # Use a default URL for mock data if none provided
        if use_mock and not linkedin_url:
            linkedin_url = "https://www.linkedin.com/in/leonkatsnelson/"

        # Extract profile data
        profile_data = extract_linkedin_profile(
            linkedin_url,
            api_key if not use_mock else None,
            mock=use_mock
        )

        if not profile_data:
            return "Failed to retrieve profile data. Please check the URL or API key.", None

        # Split data into nodes
        nodes = split_profile_data(profile_data)

        if not nodes:
            return "Failed to process profile data into nodes.", None

        # Create vector database
        index = create_vector_database(nodes)

        if not index:
            return "Failed to create vector database.", None

        # Verify embeddings
        if not verify_embeddings(index):
            logger.warning("Some embeddings may be missing or invalid")

        # Generate initial facts
        facts = generate_initial_facts(index)

        # Generate a unique session ID
        session_id = str(uuid.uuid4())

        # Store the index for this session
        active_indices[session_id] = index

        # Return the facts and session ID
        return f"Profile processed successfully!\n\nHere are 3 interesting facts about this person:\n\n{facts}", session_id

    except Exception as e:
        logger.error(f"Error in process_profile: {e}")
        return f"Error: {str(e)}", None
```

How This Function Works:

- **Model Selection:** It changes the LLM model if the user selects a different one
- **Data Extraction:** It extracts LinkedIn profile data, using mock data if specified
- **Data Processing:** It splits the profile into nodes and creates a vector database
- **Facts Generation:** It generates interesting facts about the profile
- **Session Management:** It creates a unique session ID and stores the vector index for later use
- **Error Handling:** It catches any exceptions and returns user-friendly error messages

Step 4: Implement the Chat Function

Next, let's implement the function that handles chatting with the profile. Copy-paste the code below into `chat_with_profile` function and save your files.

```
    def chat_with_profile(session_id, user_query, chat_history):
        """Chat with a processed LinkedIn profile.

        Args:
            session_id: Session ID for this conversation.
            user_query: User's question.
            chat_history: Chat history.

        Returns:
            Updated chat history.
        """
        if not session_id:
            return chat_history + [[user_query, "No profile loaded. Please process a LinkedIn profile first."]]

        if session_id not in active_indices:
            return chat_history + [[user_query, "Session expired. Please process the LinkedIn profile again."]]

        if not user_query.strip():
            return chat_history

        try:
            # Get the index for this session
            index = active_indices[session_id]

            # Answer the user's query
            response = answer_user_query(index, user_query)

            # Update chat history
            return chat_history + [[user_query, response.response]]

        except Exception as e:
            logger.error(f"Error in chat_with_profile: {e}")
            return chat_history + [[user_query, f"Error: {str(e)}"]]
```

How This Function Works:

- **Session Validation**: It checks if a profile has been processed and if the session is still active
- **Query Processing**: It retrieves the vector index for the session and uses it to answer the query
- **Chat History:** It updates the chat history with the user's question and the bot's response
- **Error Handling**: It catches any exceptions and adds error messages to the chat history

Step 5: Create the Gradio Interface

Finally, let's create the Gradio interface. Copy-paste the code below into `create_gradio_interface` function and save your files.

```
def create_gradio_interface():
    """Create the Gradio interface for the Icebreaker Bot."""
    # Define available LLM models
    available_models = [
        "ibm/granite-3-2-8b-instruct",
        "meta-llama/llama-3-3-70b-instruct"
    ]

    with gr.Blocks(title="LinkedIn Icebreaker Bot") as demo:
        gr.Markdown("# LinkedIn Icebreaker Bot")
        gr.Markdown("Generate personalized icebreakers and chat about LinkedIn profiles")

        with gr.Tab("Process LinkedIn Profile"):
            with gr.Row():
                with gr.Column():
                    linkedin_url = gr.Textbox(
                        label="LinkedIn Profile URL",
                        placeholder="https://www.linkedin.com/in/username/"
                    )
                    api_key = gr.Textbox(
                        label="ProxyCurl API Key (Leave empty to use mock data)",
                        placeholder="Your ProxyCurl API Key",
                        type="password",
                        value=config.PROXYCURL_API_KEY
                    )
                    use_mock = gr.Checkbox(label="Use Mock Data", value=True)
                    model_dropdown = gr.Dropdown(
                        choices=available_models,
                        label="Select LLM Model",
                        value=config.LLM_MODEL_ID
                    )
                    process_btn = gr.Button("Process Profile")

                with gr.Column():
                    result_text = gr.Textbox(label="Initial Facts", lines=10)
                    session_id = gr.Textbox(label="Session ID", visible=False)

            process_btn.click(
                fn=process_profile,
                inputs=[linkedin_url, api_key, use_mock, model_dropdown],
                outputs=[result_text, session_id]
            )

        with gr.Tab("Chat"):
            gr.Markdown("Chat with the processed LinkedIn profile")

            chatbot = gr.Chatbot(height=500)
            chat_input = gr.Textbox(
                label="Ask a question about the profile",
                placeholder="What is this person's current job title?"
            )

            chat_btn = gr.Button("Send")

            chat_btn.click(
                fn=chat_with_profile,
                inputs=[session_id, chat_input, chatbot],
                outputs=[chatbot]
            )

            chat_input.submit(
                fn=chat_with_profile,
                inputs=[session_id, chat_input, chatbot],
                outputs=[chatbot]
            )

    return demo
```

How This Function Works:

- **Interface Structure:** It creates a tabbed interface with separate tabs for processing profiles and chatting
- **Profile Processing Tab:** It provides fields for the LinkedIn URL, API key, mock data option, and model selection
- **Chat Tab:** It provides a chat interface for asking questions about the profile
- **Event Handlers:** It connects UI elements to our processing and chat functions

Step 6: Launch the Interface

Lastly, let's add the code to launch the interface:

```
if __name__ == "__main__":
    demo = create_gradio_interface()
    # Launch the Gradio interface
    # You can customize these parameters:
    # - share=True creates a public link you can share with others
    # - server_name and server_port set where the app runs
    demo.launch(
        server_name="127.0.0.1",
        server_port=5000,
        share=True  # Set to False if you don't want to create a public link
    )
```

This code creates and launches the Gradio interface when the script is run directly, making it accessible in a web browser.

Click the below button to see the fully updated `app.py`.

▶ Click to see the solution

Launching the application

Return to the terminal and verify that the virtual environment `venv` label appears at the start of the line. This means that you are in the `venv` environment that you just created. Then you can run the Gradio app by running the below command in the terminal.

```
python app.py
```

Note: After it runs successfully, you will see a message similar to the following example in the terminal:

Since the web application is hosted locally on port 5000, click on the following button to view the application we've developed.

Web Application

Note: If this "Web Application" button does not work, follow the following picture instructions to launch the application.

Once you launch your application. A window opens and you should be able to see the application view similar to the following example:

To stop execution of `app.py` in addition to closing the application tab, hit `Ctrl+C` in terminal.

Conclusion and next steps

Conclusion

Congratulations on completing the AI Icebreaker Bot project! You've successfully built a powerful application that leverages Retrieval-Augmented Generation (RAG) and Large Language Models to transform networking and professional connections. By integrating IBM watsonx.ai's capabilities with the LlamaIndex framework, you've created a tool that can analyze LinkedIn profiles and generate personalized conversation starters — bridging the gap between data and meaningful human interaction.

This project has taken you through the complete RAG workflow — from data extraction and processing to indexing, retrieval, and generation. You've learned how to split complex JSON data into manageable chunks, create vector embeddings, build a vector database, and construct effective prompts for various tasks. With both a command-line interface and a Gradio web application, you've made your innovative tool accessible to users of different technical backgrounds.

As a final step, feel free to:

- Test the bot with different LinkedIn profiles to see how it adapts to various career paths
- Experiment with different LLM models and different configuration settings to compare their performance on icebreaker generation
- Refine the prompts to generate even more engaging and personalized conversation starters
- Deploy your application to a cloud platform to make it accessible to others in your network

HAPPY LEARNING!

Author(s)

Hailey Quach

Other Contributor(s)

Wojciech "Victor" Fulmyk is a Data Scientist at IBM and a PhD candidate at the University of Calgary.