

使用 LlamaIndex 和 IBM Granite 构建 AI 冰破机器人

预计所需时间： 45 分钟

想象一下，你正在参加一个大型的社交活动，周围都是潜在雇主和行业领袖。你想给人留下深刻的第一印象，但却苦于想不出比“你是做什么的？”更好的开场白。

现在，想象一下拥有一个由 AI 驱动的工具为你做研究。你输入一个名字，几秒钟内，借助 **LlamaIndex** 和 **IBM watsonx** 的机器人会搜索 LinkedIn，生成基于某人职业亮点、兴趣甚至趣事的个性化冰破话题。你不再是问一些普通的问题，而是以独特而有意义的内容开始对话。

这个 AI 冰破机器人利用 **自然语言处理 (NLP)** 创建量身定制的对话引导，帮助你脱颖而出。在这个项目结束时，你将构建一个工具，使得在社交、求职面试或任何社交场合的介绍更加顺畅、个性化和难忘。

完成此实验后，你将能够：

- 理解如何使用 **LlamaIndex** 和 **IBM watsonx** 进行个性化信息检索。
- 学会如何从 **LinkedIn** 搜索和提取相关数据以生成冰破话题。
- 发展使用 **AI** 和 **自然语言处理 (NLP)** 生成基于个人在线形象的定制对话引导的技能。
- 获得将 **AI 驱动工具** 应用于自动化社交互动的实践经验，使介绍更加引人入胜和难忘。

AI 冰破机器人的简要介绍

AI 冰破机器人能做什么？

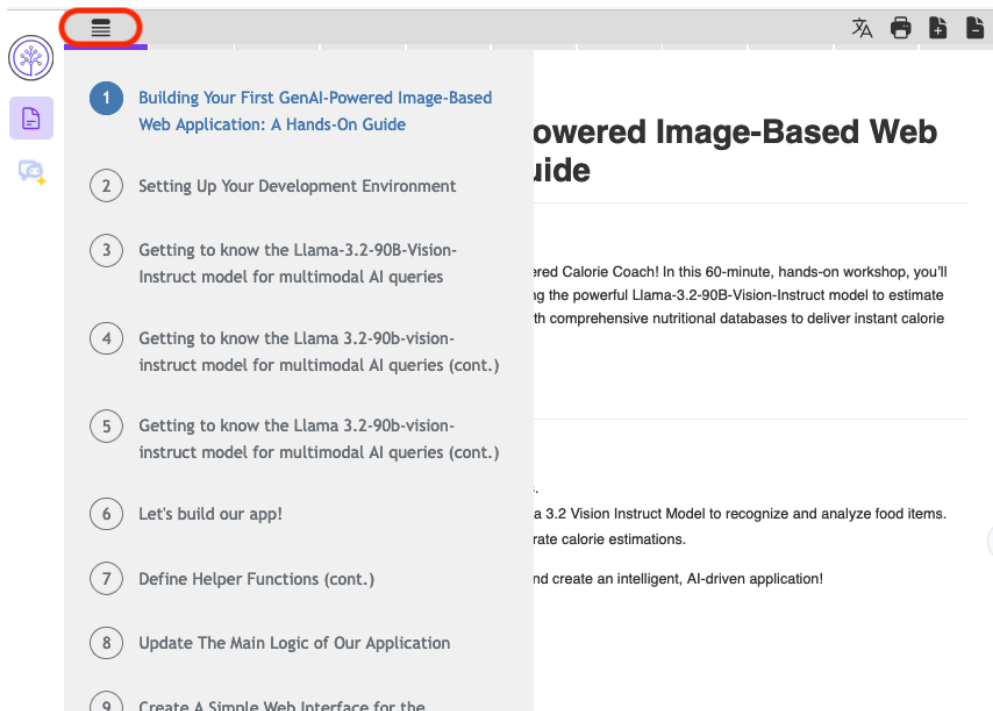
LinkedIn 冰破机器人是一个 AI 驱动的工具，可以从 LinkedIn 个人资料中生成个性化的对话引导。它使用 **IBM watsonx** 和 **LlamaIndex** 提取个人资料数据（通过 **ProxyCurl API** 或内置的模拟数据），通过 **RAG 流水线** 处理，并生成关于某人职业的量身定制的见解。通过方便的模拟数据选项，你可以在没有 API 密钥的情况下演示机器人的功能——非常适合测试或演示。只需在界面中选择“使用模拟数据”，即可立即分析预加载的专业资料。

该机器人可作为命令行工具和网络界面使用，帮助你通过用与某人实际经验和成就相关的个性化对话引导替代普通的寒暄，从而建立有意义的职业联系。

最佳体验提示 – 请阅读！

请随时参考以下提示，以应对教程中的任何困惑。如果现在看起来与您无关，请不要担心。我们会逐步讲解所有内容。

- 在项目的任何时候，如果您感到迷失，请点击页面左上角的 **目录** 图标，导航到您想要的内容。



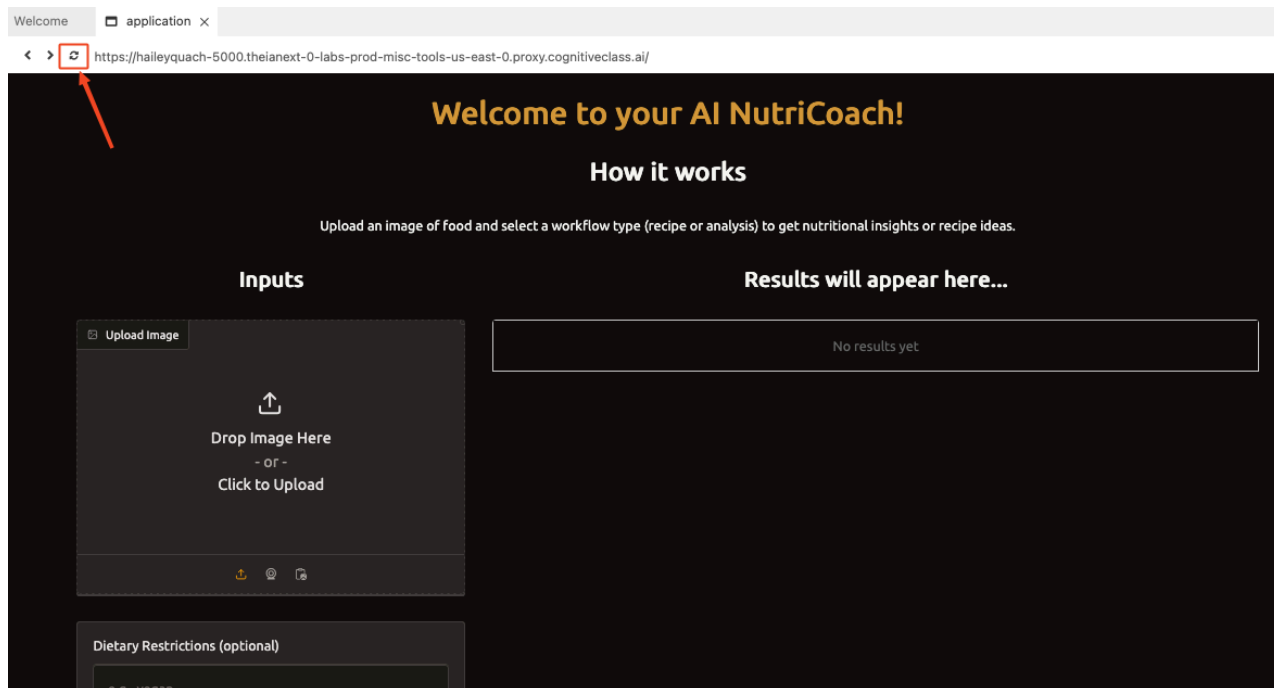
- 每当您对文件进行更改时，请确保保存您的工作。云IDE会立即自动保存您对文件所做的任何更改。您也可以通过工具栏进行保存。
- 在每个部分的末尾，您将获得该部分的完整更新脚本。在项目结束时，您还将被提示拉取项目的完整代码库。
- 在运行应用程序之前，请确保 `app.py` 在后台运行，然后再打开Web应用程序。
- 您可以通过点击右下角的 `>_` 按钮来运行代码块。

```
python app.py
```

- 始终确保您当前的目录是 `/home/project/icebreaker`。如果您不在 `icebreaker` 文件夹中，某些代码文件可能无法运行。使用 `cd` 命令导航到正确的位置。
- 确保您通过端口 `5000` 访问应用程序。点击紫色的 Web 应用程序按钮将自动通过端口 `5000` 运行应用。

Web Application

- 如果您收到无法访问其他端口（例如，`8888`）的错误，只需通过点击小刷新图标来 `refresh` 应用。如果遇到与服务器相关的其他错误，也只需刷新页面。



- 要停止 `app.py` 的执行，除了关闭应用程序标签页外，还需要在终端中按 `Ctrl+C`。
- 如果在运行应用程序时遇到错误，或者在输入所需关键字后遇到错误，请尝试使用应用程序页面顶部的按钮刷新应用。您也可以尝试输入不同的查询。
- 通常，使用 `Watsonx.ai` 提供的模型需要 `Watsonx` 凭据，包括 API 密钥和项目 ID。然而，在本实验中，这些凭据并不需要。

设置您的开发环境

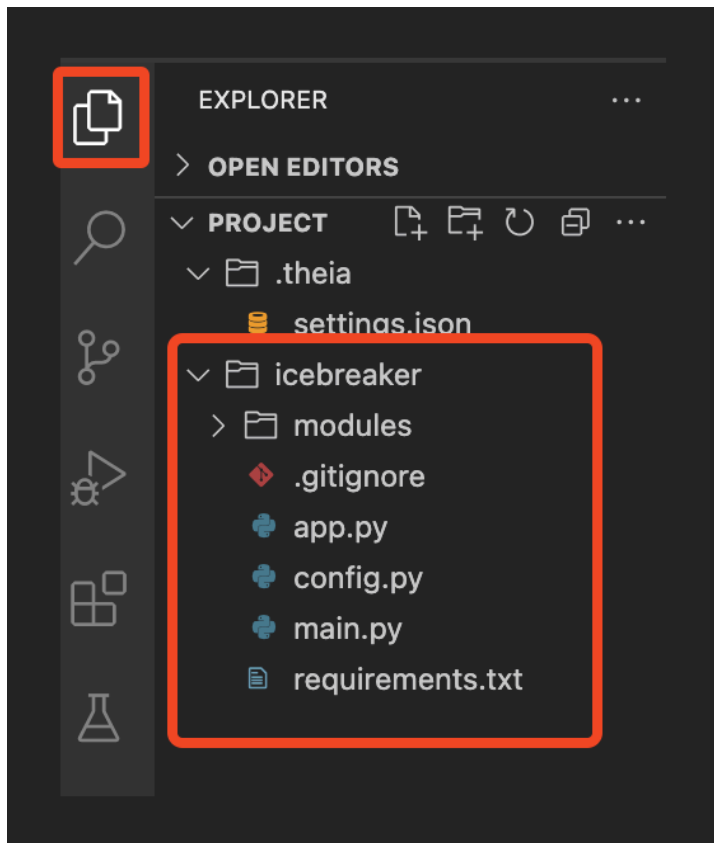
在我们开始开发之前，让我们在云 IDE 中设置您的项目环境。该环境基于 Ubuntu 22.04，并提供构建您的 AI 驱动 Gradio 应用所需的所有工具。

第一步：创建项目目录


1. 在云 IDE 中打开终端并运行：

```
git clone --no-checkout https://github.com/HaileyTQuach/icebreaker.git
cd icebreaker
git checkout 1-start
```

一旦你设置完成，选择文件资源管理器，然后选择 `icebreaker` 文件夹。你应该有以下结构的所有文件。



2. 接下来，我们将为项目设置一个虚拟环境并安装所有必需的库。

请注意，您可以通过点击运行按钮  直接运行这个代码片段。安装所有必需的库大约需要 **3-5分钟**。在等待的同时，随意去喝杯咖啡吧！

这一系列命令安装了构建基于 IBM watsonx 的检索增强生成（RAG）系统所需的重要软件包。ibm-watsonx-ai 提供对托管在 watsonx 上的基础模型的无缝访问，而 llama-index 及其模块组件——例如 llama-index-core、llama-index-llms-ibm 和 llama-index-embeddings-ibm——则支持结构化文档索引、与 IBM 模型的集成和嵌入生成。llama-index-readers-web 增加了从网站直接摄取和处理内容的能力，而 llama-hub 提供了多种数据源的预构建连接器和加载器的集合。最后，requests 用于发起 HTTP 请求，支持 API 通信和自定义集成。这些库共同创建了一个灵活且适合生产的智能信息检索和交互管道。

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

现在您的环境已设置好，您可以开始构建您的 AI Icebreaker 机器人！

第 1 部分：设置您的配置

在此步骤中，我们将为我们的 Icebreaker Bot 设置配置文件。config.py 文件将所有设置集中在一个地方，使管理应用程序参数变得更加容易。

步骤 1：打开 config.py 文件

首先，点击下面的紫色按钮打开 config.py 文件。您会看到它已经包含了一些样板代码和占位符：

Open config.py in IDE

步骤 2：添加您的 ProxyCurl API 密钥（可选）

如果您打算提取真实的 LinkedIn 个人资料（而不是使用模拟数据），请添加您的 ProxyCurl API 密钥。如果您只是使用模拟数据进行本教程，请跳过此步骤。

什么是 ProxyCurl?

ProxyCurl 是一个强大的 API，允许开发者从各种网站提取信息，包括像 LinkedIn 这样的社交媒体平台。

虽然 LlamaIndex 提供了内置的 **网页阅读器** 来阅读网站，但它无法提取 LinkedIn 数据。为了解决这个问题，我们使用 ProxyCurl API，它提供了一种可靠且高效的方式来提取 LinkedIn 个人资料数据。

注册 ProxyCurl

要在 ProxyCurl 上注册账户并获取 API 密钥：

1. 访问 ProxyCurl 的网站：[ProxyCurl](#)
2. 创建账户：点击 **注册**，填写所需信息以创建账户。
3. 选择您的电子邮件类型：
 1. 个人邮箱：注册后您将获得 20 个免费积分。
 2. 工作邮箱：如果您使用工作邮箱，注册后将获得 100 个免费积分。
4. 注册并获得免费积分后，将会提供给您一个 API 密钥。该密钥用于验证您对 ProxyCurl 的请求。获取密钥后，请将其放入下面的代码块中。

```
# TODO: Add your ProxyCurl API key here
PROXYCURL_API_KEY = "" # Replace with your API key
```

注意：对于这个项目，20个积分就足够了。**每次API调用消耗2个积分**，因此可以从10个个人资料中提取数据。如果您需要更多积分来处理额外的个人资料，您可以直接从ProxyCurl购买，但这完全由您自行决定和承担责任。

第3步：定义提示模板

此步骤最重要的是定义提示模板，这些模板将指导LLM生成响应。这些模板使用占位符，如{context_str}和{query_str}，在执行过程中将被实际内容替换。**启动文件中已经包含了提示模板。**

第4步：理解提示模板

让我们分解每个提示模板的作用：

初始事实模板：

- 目的：生成关于一个人职业或教育的3个有趣事实
- 上下文：使用{context_str}占位符，在此插入LinkedIn个人资料数据
- 指令：告诉LLM仅根据提供的上下文进行回答
- 输出格式：请求关于该人的详细回答

用户问题模板：

- 目的：回答有关LinkedIn个人资料的具体问题
- 上下文：使用{context_str}占位符用于LinkedIn数据
- 查询：使用{query_str}占位符用于用户的问题
- 指令：告诉LLM仅根据上下文进行回答，如果信息不可用，则说“我不知道”

第5步：其他配置选项（可选）

您可能想调整一些其他设置：

- CHUNK_SIZE：控制在拆分LinkedIn数据时每个文本块的大小。如果您想要更细粒度的检索，可以减少这个值（例如，设置为300）。
- SIMILARITY_TOP_K：确定在回答查询时要检索多少个相似块。增加这个值可能会提供更全面但潜在更嘈杂的响应。

```
# 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
```

第6步：保存您的配置

在进行这些更改后，保存 `config.py` 文件。您的配置现在已完成，可以供应用程序中的其他模块使用。

测试您的配置

要验证您的配置是否正确，您可以创建一个简单的测试脚本。点击下面的紫色按钮以打开 `test_config.py` 文件。

Open `test_config.py` in IDE

然后，将下面的代码复制粘贴到 `test_config.py` 中并保存该文件。

```
import config
print("ProxyCurl API Key:", "Set" if config.PROXYCURL_API_KEY else "Not set")
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)
```

运行此脚本以确保您的所有设置已正确定义。您应该在终端中看到配置打印。如果任何配置不正确，请返回并修改 `config.py`。

```
python test_config.py
```

我们的成就

在这一步中，我们已经：

- 设置了我们的 API 密钥和服务连接
- 定义了将指导我们 LLM 响应的提示模板
- 配置了我们 RAG 工作流的检索参数
- 创建了所有模块都可以使用的集中配置

接下来，我们将实现数据提取模块，以使用这些设置获取 LinkedIn 个人资料数据。

第二部分：理解核心概念

1. 大型语言模型与RAG（可选 – 如果您已经了解这些概念，可以跳过）

什么是大型语言模型（LLM）？

[大型语言模型](#)是一种人工智能（AI）模型，经过大量文本数据的训练。LLM旨在生成类似人类的文本响应，适用于广泛的问题。它们基于Transformer架构，并在多种语言任务上进行预训练，以提高其性能。

什么是IBM watsonx？

[IBM watsonx](#)是一套AI工具和服务，旨在帮助开发人员构建和部署以AI驱动的应用程序。watsonx提供一系列API和工具，使将AI能力集成到应用程序中变得简单，包括自然语言处理、计算机视觉和语音识别。在本项目中，我们使用了两种不同的LLM基础模型，均由watsonx.ai支持：`ibm/granite-3-2-8b-instruct`和`meta-llama/llama-3-3-70b-instruct`。您可以自由切换这些模型并比较它们的性能。

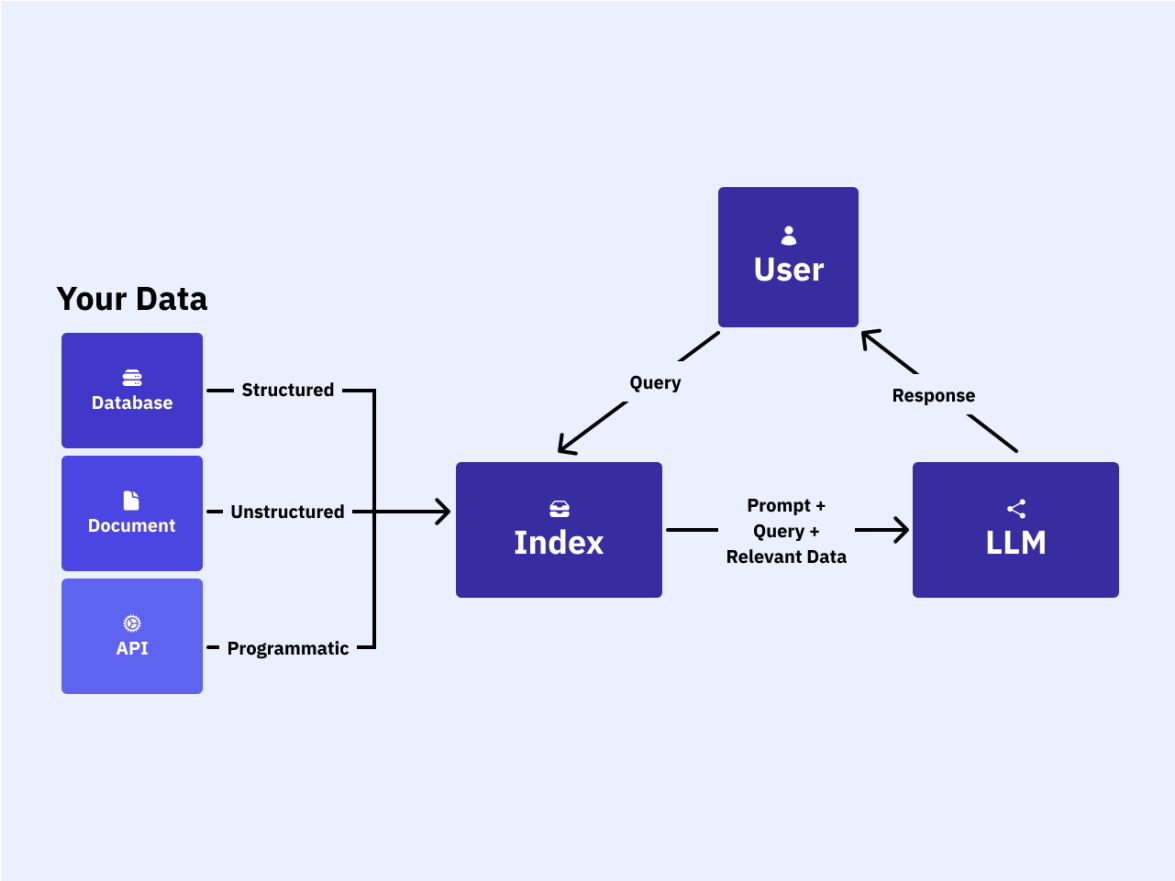
什么是LlamaIndex?

[LlamaIndex](#)是一个开源数据编排平台，用于创建大型语言模型（LLM）应用程序。LlamaIndex可以在Python和TypeScript中访问，并使用一套工具和功能来简化通过检索增强（RAG）管道进行[生成AI \(genAI\)](#)用例的上下文增强过程。

什么是RAG?

虽然LLM是使用大量数据集构建的，但它们自然不包含您的特定数据。[检索增强生成 \(RAG\)](#) 通过将您的数据与LLM的现有知识结合起来解决了这个问题。在本指南中，您会经常看到RAG的提及，因为它是查询和聊天引擎以及代理中用于提升性能的关键技术。

在RAG设置中，您的数据首先被加载、处理并索引，以便快速检索。当用户提交查询时，系统会搜索索引以找到您数据中最相关的信息。然后，这些上下文信息与用户的查询结合，并发送到LLM，LLM根据这些精炼的数据生成响应。

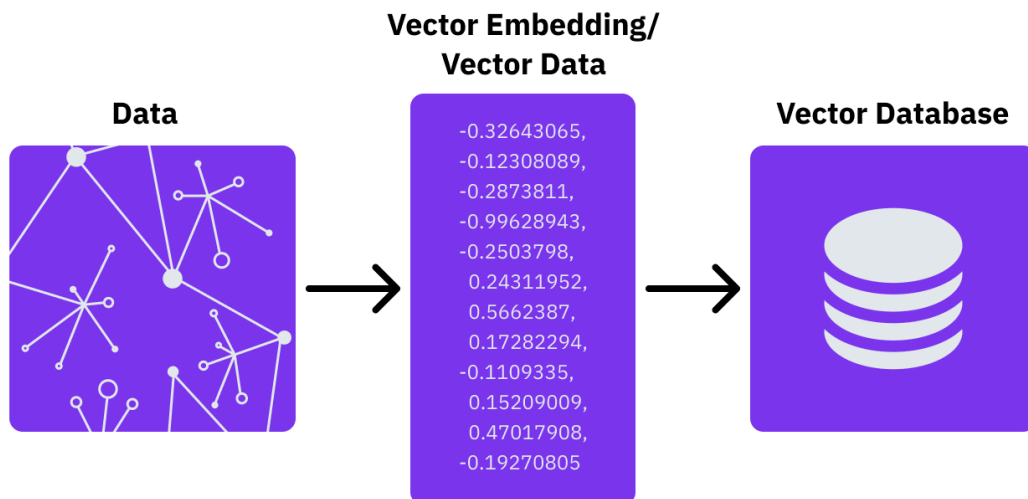


[source](#)

RAG阶段是什么?

在RAG框架中，有五个关键阶段，尽管本实验将集中于前四个。这些阶段对于您可能开发的大多数大规模应用程序至关重要。它们包括：

- **加载**：此步骤涉及将您的数据导入工作流程，无论数据来源如何——例如文本文件、PDF、网站、数据库或API。LlamaHub提供各种连接器以简化此过程。
- **拆分**：加载后，数据被分成更小、更易于管理的块，以提高检索的相关性和LLM生成响应的质量。拆分策略因数据类型和用例而异——例如，使用固定大小的窗口、语义分割或递归字符拆分文本文件。
- **索引**：在此阶段，构建一个数据结构以便于高效查询。对于LLM，这通常意味着创建向量嵌入，向量嵌入是捕捉数据含义的数值表示，结合元数据策略以确保精确和上下文感知的数据检索。
 - Index是LlamaIndex中的一个关键数据结构，允许我们在响应用户查询时检索相关上下文。索引至关重要，因为它能够快速高效地访问相关数据块，使问答变得更快、更准确。
 - 从高层次看，Indexes是由前一拆分阶段创建的Nodes构建的。这些索引随后用于构建Retrievers和Query Engines，为基于您数据的问答交互和对话体验提供动力。
 - **什么是向量嵌入？** 在我们能够索引节点之前，需要将它们转换为向量表示。这一过程称为embedding，使模型能够更好地理解文本数据，并将其语义相似性与用户查询进行比较。向量使得高效的搜索和检索操作成为可能，因为相似的文本片段将具有相似的向量表示。
 - 为了嵌入节点，我们将使用IBM watsonx Embedding model。一旦数据被嵌入，我们将在向量数据库中存储这些向量，以便在查询处理期间进行检索。



[source](#)

- **存储：**索引完成后，保存索引及其相关元数据以避免后续重新索引数据是重要的。
- **查询：**根据索引策略，有多种方式可以使用LLM和LlamaIndex结构执行查询。这可以包括子查询、多步骤查询或混合技术。
- **评估：**任何工作流程中的关键步骤是评估您方法的有效性。评估提供客观指标，以测量在比较不同方法或进行调整时查询结果的准确性、相关性和速度。



[source](#)

在我们的Icebreaker Bot中，RAG工作流程如下：

- 我们加载LinkedIn个人资料数据
- 我们将其转换为块并生成嵌入
- 我们将这些嵌入存储在向量数据库中
- 当您提出问题时，我们检索最相关的个人资料部分
- 我们使用LLM根据这个特定上下文生成个性化答案

这种方法确保我们的响应基于实际的LinkedIn个人资料数据，而不是LLM的一般知识。

什么是提示工程？

提示工程是设计和优化输入（提示）给语言模型（如GPT）以获得期望输出的过程。它涉及以一种指导模型生成准确、相关或创造性回应的方式来构建问题、指令或上下文。良好的提示工程可以提高生成文本的质量、特异性和实用性，使其成为在各种应用中利用大型语言模型的关键技能，如聊天机器人、内容生成或数据分析。

2. LinkedIn数据提取

数据提取过程

在提取LinkedIn个人资料数据时，我们使用REST API获取有关用户专业背景的结构化信息。该过程如下：

1. 准备请求：

- 格式化LinkedIn个人资料URL
- 使用您的API密钥设置身份验证头
- 配置参数（例如要包含哪些数据）

2. 发送API请求：

- 向ProxyCurl的端点发出HTTP GET请求

- 传递个人资料URL和其他参数

3. 处理响应：

- 检查请求是否成功（HTTP 200状态）
- 解析JSON响应数据
- 通过删除空值和不需要的字段来清理数据

4. 结构化数据：

- 结果JSON包含以下部分：
 - 基本信息（姓名、标题、位置）
 - 工作经验
 - 教育历史
 - 技能
 - 证书
 - 成就

数据提取是我们RAG管道的第一步，提供了后续将要拆分、索引和查询的原材料。

模拟数据与真实API调用的选项

我们的应用程序提供两种处理LinkedIn个人资料数据的选项：

使用模拟数据：

- 优势：
 - 不需要API密钥
 - 不消耗信用
 - 测试时结果一致
 - 离线工作
- 工作原理：
 - 从URL加载预生成的LinkedIn个人资料JSON
 - 将这些数据视为来自API的相同数据
 - 所有RAG处理步骤保持不变

使用真实API调用：

- 优势：
 - 访问真实的、最新的LinkedIn个人资料
 - 能够分析任何公开的LinkedIn个人资料
 - 结果更为多样化和个性化
- 工作原理：
 - 您的ProxyCurl API密钥验证请求
 - API从LinkedIn获取实时数据
 - 每个请求消耗您账户中的2个信用

选择合适的方法：

使用模拟数据的场合：

- 学习和开发
- 测试您的实现
- 无API费用的演示

使用真实API调用的场合：

- 分析特定感兴趣的个人资料
- 生产应用程序

- 数据更新至关重要的情况

在本教程的下一部分中，我们将实现这两种方法，允许您根据需要轻松切换。

第三部分：实现 LinkedIn 个人资料数据提取

在这一步中，我们将实现数据提取模块，通过 ProxyCurl API 或使用模拟数据来获取 LinkedIn 个人资料数据。

第一步：检查 data_extraction.py 启动文件

首先，让我们看看启动文件的结构。点击下面的紫色按钮以打开 data_extraction.py。

Open data_extraction.py in IDE

在这个文件中，我们需要实现 extract_linkedin_profile 函数，该函数用于检索和处理 LinkedIn 个人资料数据，既可以通过提供的 URL 和 API 密钥从 ProxyCurl API 获取数据，也可以在启用模拟选项时加载预配置的模拟数据。

练习：花点时间思考一下您将如何实现这个函数，并在继续下一步之前为其创建伪代码。

第二步：实现 extract_linkedin_profile 函数

现在，让我们实现这个函数，以处理模拟数据和真实的 API 调用。将下面的代码复制粘贴到 extract_linkedin_profile 函数中，并保存您的文件。

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

    Args:
        linkedin_profile_url: The LinkedIn profile URL to extract 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.

    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("ProxyCurl 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 {}
```

这个实现是如何工作的：

让我们来分解一下我们的实现所做的事情：

处理模拟数据与API调用

如果 `mock=True`，我们从配置中的预定义URL加载数据

如果 `mock=False`，我们准备一个带有提供的API密钥的ProxyCurl API请求

API请求配置

- 我们在 <https://nubela.co/proxycurl/api/v2/linkedin> 设置API端点
- 我们使用API密钥配置授权头
- 我们指定参数，例如：
 - LinkedIn个人资料URL
 - 缓存设置以减少API使用
 - 需要包含的其他数据（技能、薪资信息等）

响应处理

- 我们检查成功的状态码（200）
- 我们解析JSON响应
- 我们通过以下方式清理数据：
 - 移除空值（空列表、空字符串、None）
 - 排除不需要的字段，如“people_also_viewed”
 - 从组中移除个人资料图片URL

错误处理

- 我们将所有内容包装在try/except块中，以优雅地处理错误
- 我们记录详细的错误信息以帮助调试
- 如果出现问题，我们返回一个空字典

点击下面的按钮查看完整更新的 `data_extraction.py`。

► 点击查看解决方案

我们的成就

在这一步中，我们已经：

- 实现了LinkedIn个人资料数据提取功能
- 添加了对模拟数据和真实API调用的支持
- 增强了错误处理和日志记录
- 清理了响应数据以便于处理

这个功能构成了我们RAG管道的基础，提供了我们将在后续步骤中处理、索引和查询的原始数据。

第4部分：实现RAG的数据处理

在这一步中，我们将实现数据处理模块，将原始的LinkedIn个人资料数据转换为适合检索和问答的格式。这涉及将数据拆分为块、创建向量嵌入，并将其存储在可搜索的数据库中。

第一步：检查data_processing.py入门文件

让我们首先检查入门文件结构。点击下面的紫色按钮打开data_processing.py。

Open data_processing.py in IDE

我们需要实现三个函数：

1. split_profile_data: 将个人资料数据划分为可管理的块
2. create_vector_database: 从块中创建向量索引
3. verify_embeddings: 确保所有嵌入都已正确创建

练习：花点时间思考一下你将如何实现这些函数，并为它们创建伪代码，然后再继续下一步。

第二步：实现split_profile_data函数

在传统的RAG框架中，加载阶段之后的下一步是拆分。这涉及将数据划分为更小、可管理的块，以便能够有效地嵌入到向量数据库中，并在后续检索用户查询时使用。



[来源](#)

首先，让我们实现将个人资料数据拆分为节点的函数。将下面的代码复制粘贴到split_profile_data函数中并保存你的文件。

```
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 []
```

此功能：

- 将个人资料数据字典转换为 JSON 字符串
- 从字符串创建一个 Document 对象
- 使用 SentenceSplitter 将文本分割成配置中指定大小的块
- 返回结果节点

在我们的项目中，我们需要将从 ProxyCurl 抓取的 LinkedIn 个人资料数据拆分成更小的段落或节点。这些节点代表可以后续查询的逻辑信息块。

由于 LinkedIn 个人资料数据是一个 JSON 文件，我们首先将其转换为文本格式，然后将文本拆分成每个约 500 个字符的小节点。这个拆分过程确保我们的数据以可管理的块进行索引，使得模型能够根据用户查询更容易地检索相关信息。

步骤 3: 实现 create_vector_database 函数

在这一步中，我们进入 **索引** 和 **存储** 这些节点以便高效检索。这一步对于构建 RAG 的基础至关重要，在此过程中我们检索相关数据以回答用户查询。



[source](#)

接下来，让我们实现创建向量数据库的功能。将下面的代码复制粘贴到 create_vector_database 函数中，并保存您的文件。

```
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
```

这个函数：

- 从 llm_interface 模块获取嵌入模型
- 使用节点和嵌入模型创建 VectorStoreIndex
- 返回索引，如果发生错误则返回 None

第 4 步: 实现 verify_embeddings 函数

最后，让我们实现验证嵌入的函数。将下面的代码复制粘贴到 verify_embeddings 函数中并保存您的文件。

```
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.")
    except Exception as e:
        logger.error(f"Error in verify_embeddings: {e}")
        return False
    return not missing_embeddings
```

```
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
```

这个函数：

- 从索引中获取向量存储
- 获取节点 ID 列表
- 检查每个节点是否具有有效的嵌入
- 如果所有嵌入都是有效的，则返回 True，否则返回 False

点击下面的按钮查看完整更新的 `data_processing.py`。

► 点击查看解决方案

我们的成就

在这一步，我们已经：

- 实现了将LinkedIn个人资料数据拆分为可管理块的功能
- 实现了从这些块创建向量数据库的功能
- 实现了验证嵌入的功能
- 创建了一个测试脚本来验证我们的实现

这些功能构成了我们RAG管道索引阶段的核心。现在数据已经正确处理和索引，我们可以继续实现查询引擎，以根据这些索引数据生成响应。

第 5 部分：实现与 IBM watsonx.ai 的语言和嵌入模型的接口

在本节中，我们将实现与 IBM watsonx.ai 的语言和嵌入模型的接口。这是我们 Icebreaker Bot 的一个关键组件，因为它处理与驱动我们应用程序的 AI 模型的连接。

步骤 1：检查 `llm_interface.py` 启动文件

让我们首先检查启动文件的结构。点击下面的紫色按钮打开 `llm_interface.py`。

Open `llm_interface.py` in IDE

在这个文件中，我们需要实现 3 个函数：

- `create_watsonx_embedding()`: 该函数创建将文本转换为向量表示的嵌入模型。它应该返回一个配置了正确模型 ID、URL 和项目 ID 的 `WatsonxEmbeddings` 实例，这些信息来自我们的配置文件。这个嵌入模型将用于将 LinkedIn 个人资料数据块转换为用于语义搜索的向量。
- `create_watsonx_llm()`: 该函数创建生成用户查询响应的语言模型。它应该返回一个配置了控制生成过程的参数的 `WatsonxLLM` 实例，例如温度（用于随机性）、令牌限制和解码方法。这个 LLM 将负责生成有趣的事实并回答有关 LinkedIn 个人资料的问题。
- `change_llm_model()`: 这个实用函数允许我们在运行时动态切换不同的语言模型。它应该更新我们配置中的 LLM 模型 ID，并记录更改。这种灵活性使得可以尝试不同的模型，以比较它们在冰破生成任务中的性能。

练习：花点时间思考一下你将如何实现这些函数，并为它们创建伪代码，然后再继续进行下一步。

步骤 2：实现嵌入模型函数

接下来，我们将实现创建嵌入模型的函数。将下面的代码复制粘贴到 `create_watsonx_embedding` 函数中并保存你的文件。

```
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

```

此函数创建一个 WatsonxEmbeddings 实例，我们将用它将 LinkedIn 个人资料中的文本块转换为向量表示。这些向量捕捉文本的语义含义，使我们能够在回答用户查询时找到最相关的信息。

关键参数：

- model_id: 指定要使用的嵌入模型（在 config.py 中定义）
- url: watsonx.ai 服务的端点 URL
- project_id: watsonx.ai 的项目 ID
- truncate_input_tokens: 控制模型如何处理超出模型上下文窗口的令牌

第 3 步：实现语言模型函数

现在，让我们实现创建语言模型的函数。将下面的代码复制粘贴到 create_watsonx_llm 函数中并保存您的文件。

```

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

```

此函数创建一个 WatsonxLLM 的实例，我们将用它来生成对用户查询的响应。它连接到 IBM watsonx.ai，并根据我们的用例配置语言模型的适当参数。

关键参数：

- temperature: 控制生成文本的随机性。较低的值（如 0.0）使响应更具确定性和集中性，而较高的值则引入更多创造性和变化。
- max_new_tokens: 限制生成响应的长度。
- decoding_method: 确定模型如何选择下一个标记。选项包括：
 - “sample”: 从概率分布中随机抽样（更具创造性）

- “greedy”: 始终选择最可能的下一个标记（更具确定性）
- additional_params: 模型的额外配置：
 - min_new_tokens: 生成的最小标记数
 - top_k: 将标记选择限制为前 k 个最可能的标记
 - top_p: 使用核采样从构成前 p 概率质量的标记中选择

第 4 步：实现模型切换功能

最后，我们将实现一个函数，允许我们在不同的 LLM 模型之间切换。将下面的代码复制粘贴到 `change_llm_model` 函数中并保存您的文件。

```
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}")
```

此功能允许用户尝试不同的语言模型。它在配置模块中更新 `LLM_MODEL_ID` 以使用不同的模型来生成响应。在这个项目中，我们使用了两个模型： `ibm/granite-3-2-8b-instruct` 和 `meta-llama/llama-3-3-70b-instruct`。如果您想探索 `watsonx.ai` 上的更多模型，请随时根据您的喜好更新列表。请参阅 [此页面](#) 以获取当前支持的基础模型列表。

点击下面的按钮查看完整更新的 `llm_interface.py`。

► 点击查看解决方案

我们的成就

在这一步中，我们：

- 设置了与 IBM watsonx 嵌入和语言模型的接口
- 配置了关键生成参数，包括温度和解码方法
- 实现了模型切换功能以便于实验
- 创建了一个干净的抽象层，隐藏了连接 AI 服务的复杂性

这个 LLM 接口作为我们 Icebreaker Bot 的智能层，使其能够通过嵌入理解 LinkedIn 个人资料的语义，并通过语言模型生成个性化的破冰话题。

第6部分：创建查询引擎

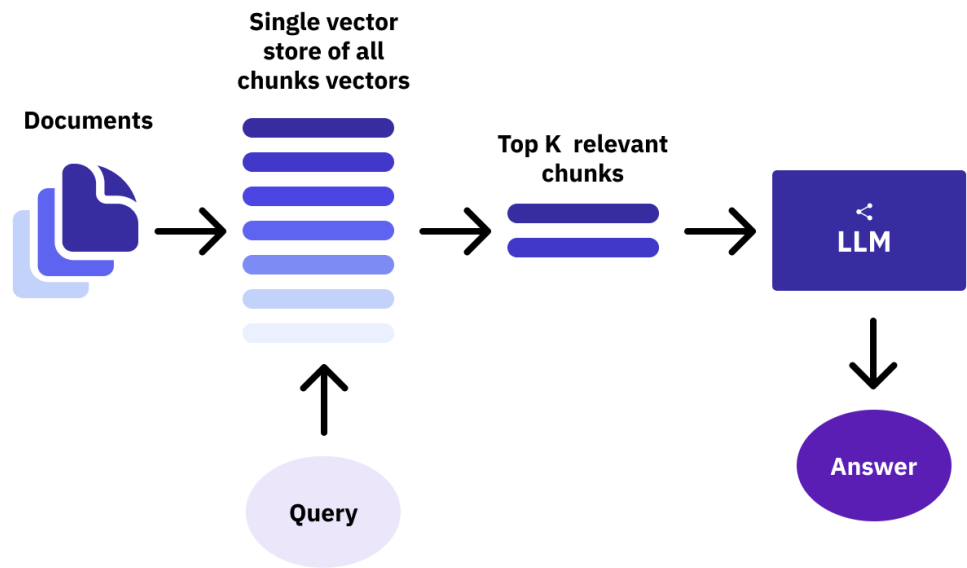
现在我们已经对 LinkedIn 数据进行了索引，并配置了我们的 LLM 接口，我们需要创建查询引擎，以支持我们的 Icebreaker Bot 生成洞察和回答问题的能力。该模块负责从我们的向量索引中检索相关信息，并将其格式化为引人入胜的响应。

什么是查询引擎？

在我们的 RAG（检索增强生成）系统中，查询引擎负责：

- 根据查询从我们的向量索引中检索相关信息
- 将检索到的信息格式化为 LLM 的提示
- 基于检索到的上下文生成连贯且准确的响应

Basic Index Retrieval



[来源](#)

让我们逐步实现我们的查询引擎。

第一步：检查query_engine.py启动文件

让我们首先检查启动文件的结构。点击下面的紫色按钮以打开query_engine.py。

Open query_engine.py in IDE

在这个文件中，我们需要实现两个函数：

- generate_initial_facts(index)：该函数根据一个人的LinkedIn资料生成引人入胜的对话开场白。它应该创建一个具有适当参数的 watsonx LLM用于事实生成，使用我们在配置中预定义的模板构建提示模板，构建一个将索引与LLM和提示结合的查询引擎，然后执行查询以生成关于此人的三个有趣事实。
- answer_user_query(index, user_query)：该函数为我们的机器人提供互动问答能力。它应该创建一个优化用于问答的watsonx LLM（使用与事实生成不同的参数），根据用户的查询从我们的向量索引中检索最相关的节点，从这些节点构建上下文字符串，使用我们的问答提示模板创建查询引擎，并执行查询以生成精确的答案。该函数实现了完整的RAG（检索增强生成）工作流程，使得能够准确、上下文相关地响应关于LinkedIn资料的问题。

练习：花一点时间思考您将如何实现这些函数，并为它们创建伪代码，然后再继续进行下一步。

第二步：实现事实生成函数

首先，让我们实现生成关于一个人的LinkedIn资料的有趣事实的函数。将下面的代码复制粘贴到generate_initial_facts函数中并保存您的文件。

```
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."

```

这个功能是如何工作的：

1. **LLM 配置：** 我们创建一个具有特定参数的语言模型实例：

- temperature=0.0：使输出更具确定性，专注于事实信息
- max_new_tokens=500：允许详细的响应
- decoding_method="sample"：在保持事实性的同时引入一些变化

2. **提示模板：** 我们使用一个预定义的模板（来自 config.py），指示 LLM 如何根据个人资料数据生成事实。

3. **查询引擎创建：** 我们使用以下参数从索引创建一个查询引擎：

- streaming=False：等待完整响应，而不是流式传输
- similarity_top_k=config.SIMILARITY_TOP_K：从个人资料中检索最相关的片段
- llm=watsonx_llm：使用我们配置的语言模型
- text_qa_template=facts_prompt：使用我们的事实生成提示模板

4. **查询执行：** 我们向引擎发送一个简单的查询并返回响应

5. **错误处理：** 我们捕获任何异常并返回友好的错误消息

第 3 步：实现问答功能

现在，让我们实现一个回答用户关于 LinkedIn 个人资料的特定问题的功能。将下面的代码复制粘贴到 answer_user_query 函数中并保存文件。

```

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."
```

这个功能是如何工作的：

1. **LLM 配置：** 我们创建一个具有特定参数的语言模型实例：

- `temperature=0.0`：使输出更加确定，专注于事实信息
- `max_new_tokens=250`：保持答案简洁
- `decoding_method="greedy"`：始终选择最可能的下一个标记，以确保最大准确性

2. **提示模板：** 我们使用一个问答模板，指示 LLM 根据提供的上下文进行回答。

3. **相关节点检索：** 这是一个关键步骤，我们：

- 从我们的索引中创建一个检索器
- 根据用户的查询检索最相关的节点
- 从这些节点构建上下文字符串

4. **查询引擎创建：** 类似于事实生成，但针对问答进行了优化

5. **查询执行：** 我们将用户的查询发送到引擎，并返回完整的响应对象

6. **错误处理：** 我们捕获任何异常并返回友好的错误信息

点击下面的按钮查看完整更新的 `query_engine.py`。

► 点击查看解决方案

我们的成就

在这一步中，我们已经：

- 实现了我们的 Icebreaker Bot 的核心查询引擎
- 创建了生成有趣事实和回答特定问题的函数
- 设置了 RAG 管道，以在生成响应之前检索相关上下文
- 添加了强大的错误处理和日志记录
- 为不同类型的查询配置了不同的 LLM 参数

这个查询引擎是我们应用的核心，将索引的 LinkedIn 数据与 LLM 连接起来，以生成个性化、准确的响应。在下一部分中，我们将把这些组件整合成一个完整的用户界面应用。

第7部分：构建命令行界面（CLI）应用程序

现在我们已经构建了 Icebreaker Bot 的所有关键组件，是时候将它们整合到一个命令行界面（CLI）应用程序中。这将允许用户直接从终端与机器人互动，提供 LinkedIn URL 并询问关于该个人资料的问题。

第一步：检查 `main.py` 启动文件

首先，让我们检查启动文件的结构。点击下面的紫色按钮以打开 `main.py`。

Open `main.py` in IDE

在 `main.py` 模块中，我们需要实现两个核心函数，以协调整个 Icebreaker Bot 的工作流程并提供用户界面：

- `process_linkedin(linkedin_url, api_key, mock)`：这个函数作为我们应用程序的核心协调者。它应该通过首先提取 LinkedIn 个人资料数据（可以通过 API 或模拟数据），然后将这些数据拆分成可管理的节点，创建一个基于这些节点的向量数据库，验证嵌入是否正确创建，生成关于该个人的初始对话引导，最后启动互动聊天机器人界面来协调整个 RAG 管道。这个函数处理从原始 LinkedIn URL 到完全功能的对话助手的端到端工作流程。

- `chatbot_interface(index)`: 这个函数为与我们的机器人互动创建一个用户友好的命令行界面。它应该向用户显示清晰的指示，创建一个循环来处理用户关于LinkedIn个人资料的问题，使用我们的查询引擎检索答案，以对话格式显示这些答案，并提供一个干净的退出机制。

`main.py` 文件还包括一个现成的 `main()` 函数，用于处理命令行参数，提供灵活的使用方式。一旦我们实现了缺失的函数，取消注释 `process_linkedin` 调用将激活完整的应用程序。

练习：花一点时间思考一下你将如何实现这些函数，并为它们创建伪代码，然后再继续下一步。

第二步：实现 Process LinkedIn 函数

这是协调整个工作流程的核心函数。将下面的代码复制粘贴到 `process_linkedin` 函数中并保存文件。

```
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)}")
```

该功能的工作原理：

1. **提取：** 它调用我们的 `extract_linkedin_profile` 函数以获取个人资料数据
2. **处理：** 它将数据拆分为节点并创建一个向量数据库
3. **初始事实：** 它生成 3 个有趣的事实作为起点
4. **用户界面：** 它打印这些事实并启动聊天机器人界面
5. **日志记录和计时：** 它记录每一步并跟踪每个过程所花费的时间
6. **错误处理：** 它捕获并报告处理过程中发生的任何错误

第 3 步：实现聊天机器人界面

接下来，让我们与机器人交互创建一个简单的基于文本的界面。将下面的代码复制粘贴到 `chatbot_interface` 函数中并保存您的文件。

```
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")

```

这个功能是如何工作的：

1. 用户输入： 它提示用户提出关于个人资料的问题
2. 查询处理： 它将用户查询发送到我们的 `answer_user_query` 函数
3. 响应显示： 它以对话格式显示机器人的响应
4. 退出命令： 它允许用户通过简单命令结束对话
5. 错误处理： 它处理中断和错误

第4步：创建主入口点

最后，让我们创建一个主函数，将所有内容联系在一起。将下面的代码复制粘贴到 `main` 函数中并保存您的文件。

```

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()

```

这个函数是如何工作的：

1. 参数解析： 它处理命令行参数
2. 交互式提示： 如果缺少必要的信息，它会提示用户
3. 个人资料处理： 它使用提供的参数调用我们的 `process_linkedin` 函数

点击下面的按钮查看完整更新的 `main.py`。

► 点击查看解决方案

测试完整的 CLI 工作流

要测试我们的 CLI 应用程序，我们可以使用各种参数组合运行它：

1. 使用 LinkedIn URL 和 API 密钥：（用真实的 LinkedIn URL 和您的 API 密钥替换）：

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

2. 使用模拟数据：

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

3. 互动式（带提示）：

```
python main.py
```

我们的成就

在这一步中，我们已经：

- 实现了一个全面的主应用程序，协调我们所有的模块
- 创建了一个从头到尾处理 LinkedIn 个人资料的函数
- 构建了一个简单但有效的聊天机器人界面供用户互动
- 添加了命令行参数解析，以实现灵活使用
- 包含了计时和日志记录，以便更好的调试和监控
- 在整个应用程序中添加了强大的错误处理

main.py 文件作为连接我们所有组件的粘合剂，提供了从个人资料 URL 到个性化破冰者和对话的无缝用户体验。

接下来的步骤

随着我们的 CLI 应用程序完成，用户现在可以直接从终端与 Icebreaker Bot 互动。然而，基于网页的界面将使机器人更加易于访问和用户友好。在下一部分中，我们将构建一个 Gradio 网页界面，提供所有相同的功能，并带有图形用户界面。

第8部分：使用Gradio构建用户友好的网页界面

现在我们已经构建了命令行界面，让我们使用Gradio创建一个更用户友好的网页界面。这将使我们的Icebreaker Bot对喜欢图形界面的用户可用，并提供更精致的体验。

第一步：检查app.py启动文件

让我们首先检查启动文件的结构。点击下面的紫色按钮打开 app.py。

Open **app.py** in IDE

app.py模块使用Gradio创建了一个基于网页的Icebreaker Bot界面，Gradio是一个用于构建机器学习模型网页界面的Python库。这个界面将会：

- 允许用户输入LinkedIn个人资料URL
- 处理个人资料并显示有趣的事实
- 提供一个聊天界面以询问有关个人资料的问题
- 支持在不同的LLM模型之间切换

让我们一步一步实现这一点。

第二步：导入必要的库和我们的模块

```
"""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 = {}
```

这些导入使我们能够访问：

- 用于唯一 ID 和日志记录的标准 Python 库
- 用于构建网页界面的 Gradio
- 我们的自定义模块以实现核心功能
- 一个字典用于存储跨会话的活跃对话

第 3 步：实现个人资料处理功能

接下来，让我们实现处理 LinkedIn 个人资料的函数。将下面的代码复制粘贴到 process_profile 函数中并保存您的文件。

```
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

```

这个功能是如何工作的：

- **模型选择：** 如果用户选择不同的 LLM 模型，它会更改模型
- **数据提取：** 它提取 LinkedIn 个人资料数据，如果指定则使用模拟数据
- **数据处理：** 它将个人资料拆分成节点并创建向量数据库
- **事实生成：** 它生成关于个人资料的有趣事实
- **会话管理：** 它创建一个唯一的会话 ID 并存储向量索引以备后用
- **错误处理：** 它捕获任何异常并返回用户友好的错误消息

第 4 步：实现聊天功能

接下来，让我们实现处理与个人资料聊天的功能。将下面的代码复制粘贴到 `chat_with_profile` 函数中并保存你的文件。

```

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)}"]]
```

这个功能是如何工作的：

- **会话验证：** 检查配置文件是否已处理，且会话是否仍然有效
- **查询处理：** 检索会话的向量索引，并利用它来回答查询
- **聊天记录：** 更新聊天记录，包括用户的问题和机器人的回答
- **错误处理：** 捕获任何异常并将错误消息添加到聊天记录中

第5步：创建Gradio界面

最后，让我们创建Gradio界面。将下面的代码复制粘贴到 `create_gradio_interface` 函数中并保存您的文件。

```
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
```

这个功能是如何工作的：

- **界面结构：** 它创建了一个带标签的界面，分别用于处理个人资料和聊天
- **个人资料处理标签：** 提供了LinkedIn URL、API密钥、模拟数据选项和模型选择的字段
- **聊天标签：** 提供了一个聊天界面，用于询问有关个人资料的问题
- **事件处理程序：** 将UI元素连接到我们的处理和聊天功能

第6步： 启动界面

最后，让我们添加启动界面的代码：

```
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
    )
```

这段代码在脚本直接运行时创建并启动 Gradio 界面，使其可以在网页浏览器中访问。

点击下面的按钮查看完整更新的 `app.py`。

► 点击查看解决方案

启动应用程序

返回终端，确认venv虚拟环境的标签出现在行的开头。这意味着您已进入刚刚创建的venv环境。然后，您可以通过在终端运行以下命令来启动Gradio应用程序。

```
python app.py
```

注意： 成功运行后，您将在终端中看到类似以下示例的消息：

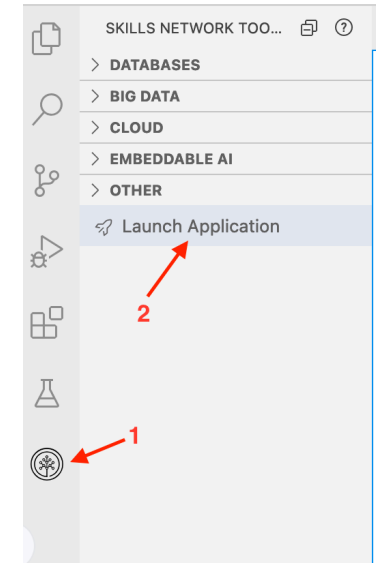
```
(venv) C:\Users\haseyquela\OneDrive\Documents> python app.py
INFO:root:Extracting ingredients from image...
* Running on local URL:  http://127.0.0.1:5000
INFO:httpx:HTTP Request: GET http://127.0.0.1:5000/gradio_api/startup-events "HTTP/1.1 200 OK"
INFO:httpx:HTTP Request: HEAD http://127.0.0.1:5000/ "HTTP/1.1 200 OK"

To create a public link, set `share=True` in `launch()`.
INFO:httpx:HTTP Request: GET https://api.gradio.app/pkg-version "HTTP/1.1 200 OK"
□
```

由于该网络应用程序托管在本地端口 5000，请点击以下按钮查看我们开发的应用程序。

Web Application

注意：如果这个“Web 应用程序”按钮无法工作，请按照以下图片说明启动应用程序。



Launch Your Application

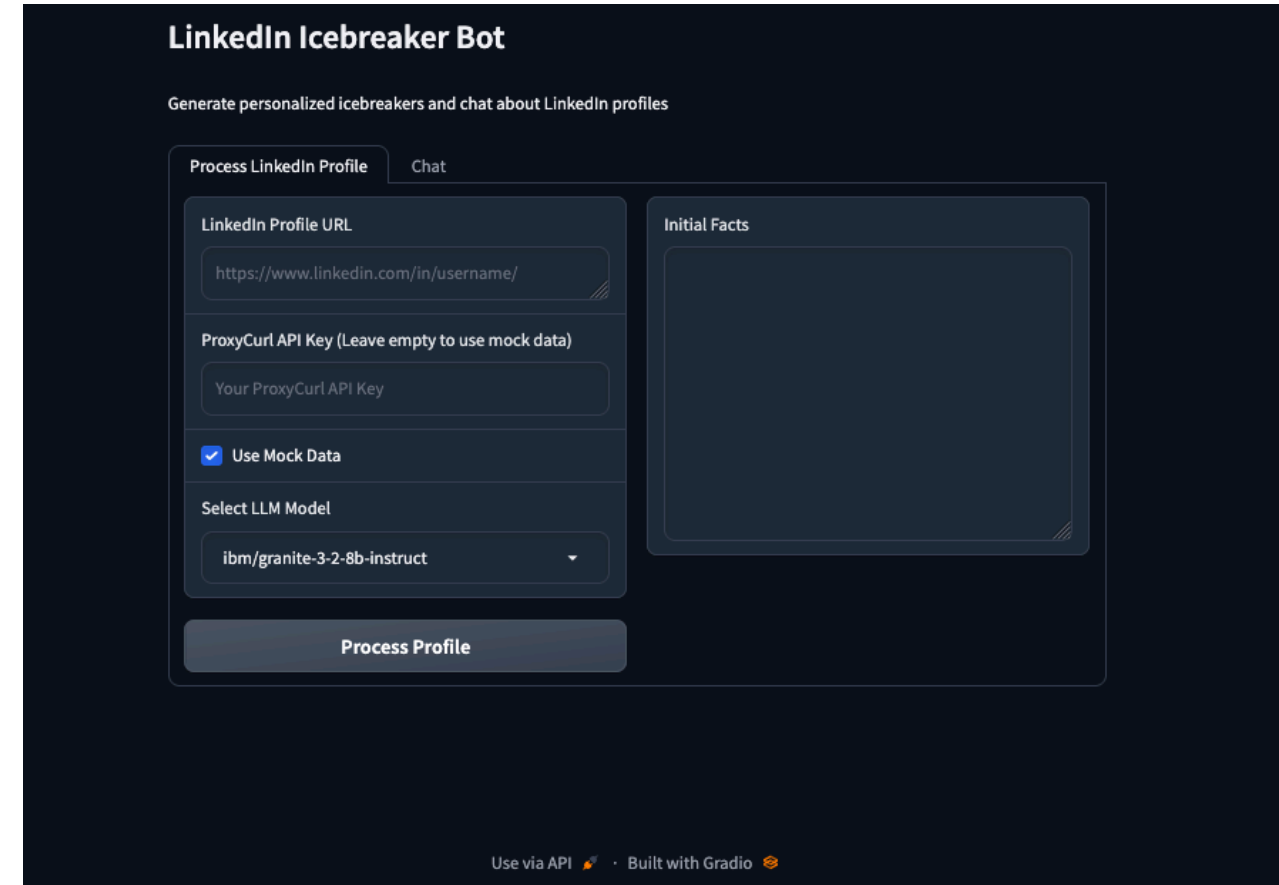
To open any application in the browser, please select or enter the port number below.

Application Port

5000

Your Application

一旦您启动应用程序，将会打开一个窗口，您应该能够看到类似以下示例的应用程序视图：



要停止 app.py 的执行，除了关闭应用程序标签外，还请在终端中按 Ctrl+C。

结论与下一步

结论

恭喜你完成了 AI Icebreaker Bot 项目！你成功构建了一个强大的应用程序，利用增强检索生成（RAG）和大型语言模型来改变网络和专业连接。通过将 IBM watsonx.ai 的能力与 LlamaIndex 框架相结合，你创建了一个可以分析 LinkedIn 个人资料并生成个性化对话开场白的工具——弥合数据与有意义的人际互动之间的鸿沟。

这个项目带你经历了完整的 RAG 工作流程——从数据提取和处理到索引、检索和生成。你学会了如何将复杂的 JSON 数据分割成可管理的块，创建向量嵌入，构建向量数据库，并为各种任务构建有效的提示。通过命令行界面和 Gradio 网络应用程序，你使这个创新工具对不同技术背景的用户都可访问。

作为最后一步，欢迎你：

- 使用不同的 LinkedIn 个人资料测试机器人，看看它如何适应各种职业路径
- 尝试不同的 LLM 模型和配置设置，以比较它们在冰破生成方面的表现
- 精炼提示，以生成更具吸引力和个性化的对话开场白
- 将你的应用程序部署到云平台，使其对你网络中的其他人可访问

快乐学习！

Author(s)

[Hailey Quach](#)

Other Contributor(s)

[Wojciech "Victor" Fulmyk](#) 是 IBM 的数据科学家，也是卡尔加里大学的博士候选人。