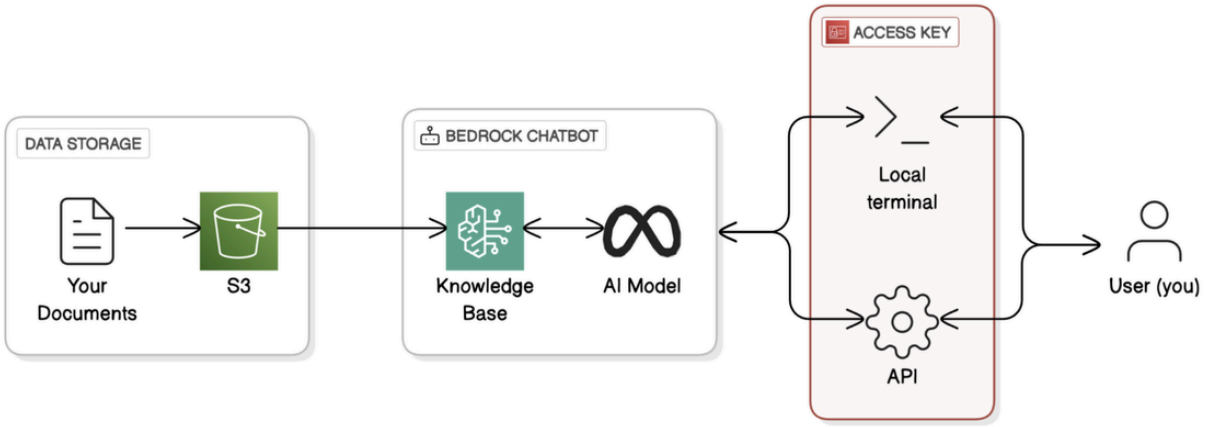
Kareshma Rajaananthapadmanaban

Create an API for Your RAG Chatbot

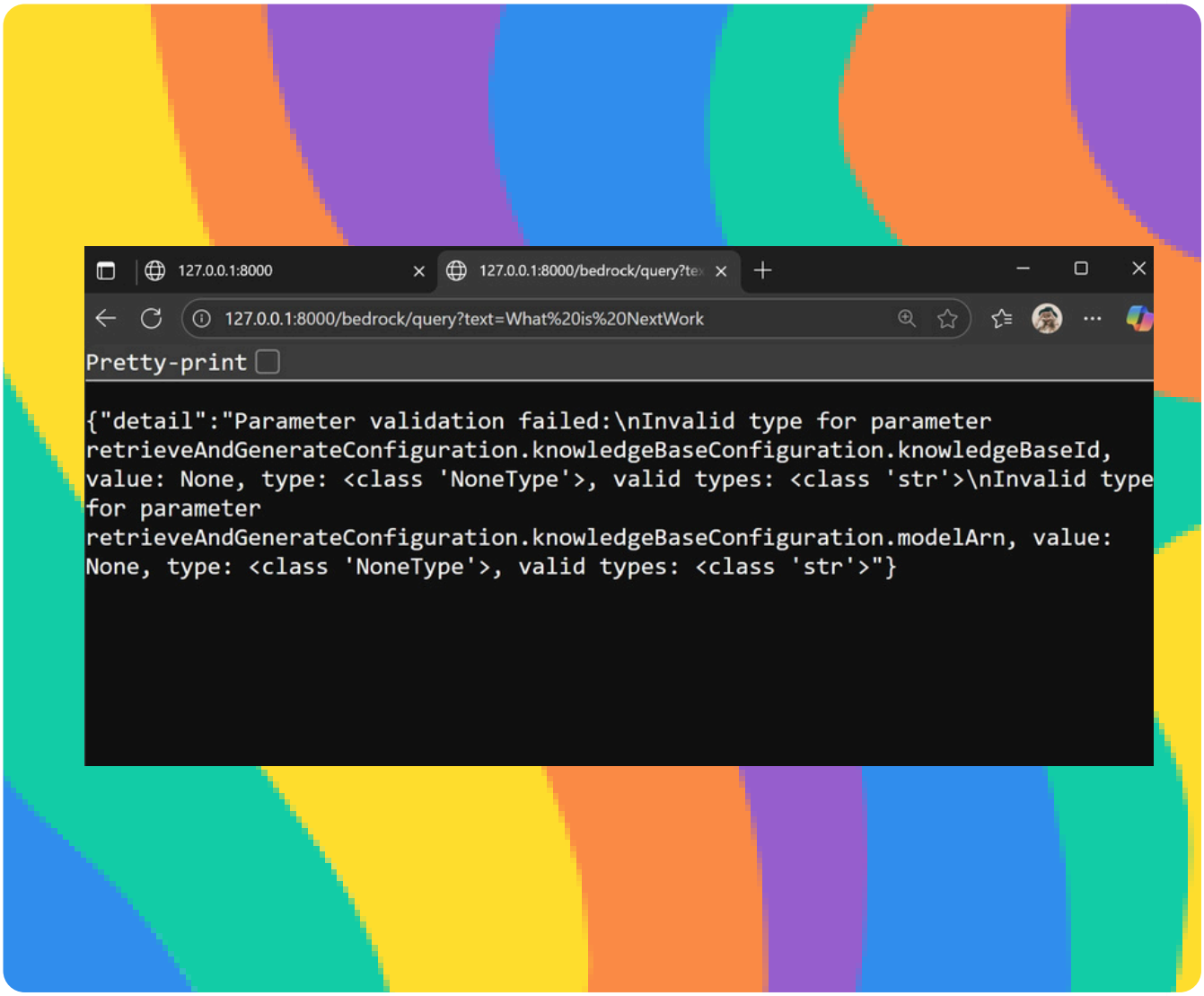
Build a RAG chatbot using Amazon Bedrock, and create an API that makes your chatbot accessible to any application!



nextwork.org

How I Built an API for My RAG Chatbot

Kareshma Rajaananthapadmanaban



Introducing Today's Project!

In this project, I will demonstrate how to turn my **RAG chatbot** into an **API** that any application can access. I'm doing this project to learn how to connect Amazon Bedrock with **FastAPI** so my chatbot isn't limited to the console or terminal. This way, I can make it **production-ready** and integrate it into websites, apps, or even other tools.

Tools and concepts

Services I used were **Amazon Bedrock** for calling foundation models, **FastAPI** for building **API endpoints**, and dotenv for managing environment variables. Key concepts I learnt include the difference between **MODEL_ID** and **MODEL_ARN**, how to extend an API with new endpoints, handling environment variables, and the contrast between direct model invocation and Knowledge Base queries.

Project reflection

This project took me approximately 2 hours to complete. The most challenging part was troubleshooting the missing MODEL_ID environment variable for the new endpoint. It was most rewarding to see the chatbot handle both Knowledge **Base queries** and **direct AI model** questions, giving me flexibility to use specialized or general knowledge.

I did this project today to deepen my hands-on understanding of how **Retrieval-Augmented Generation** works in **AWS Bedrock** and to practice extending APIs for both **document-based** and **general AI queries.** This project met my goals because I now feel more confident in configuring endpoints, managing environment variables, and explaining the difference between Knowledge Base and direct model calls.

Setting Up The Knowledge Base

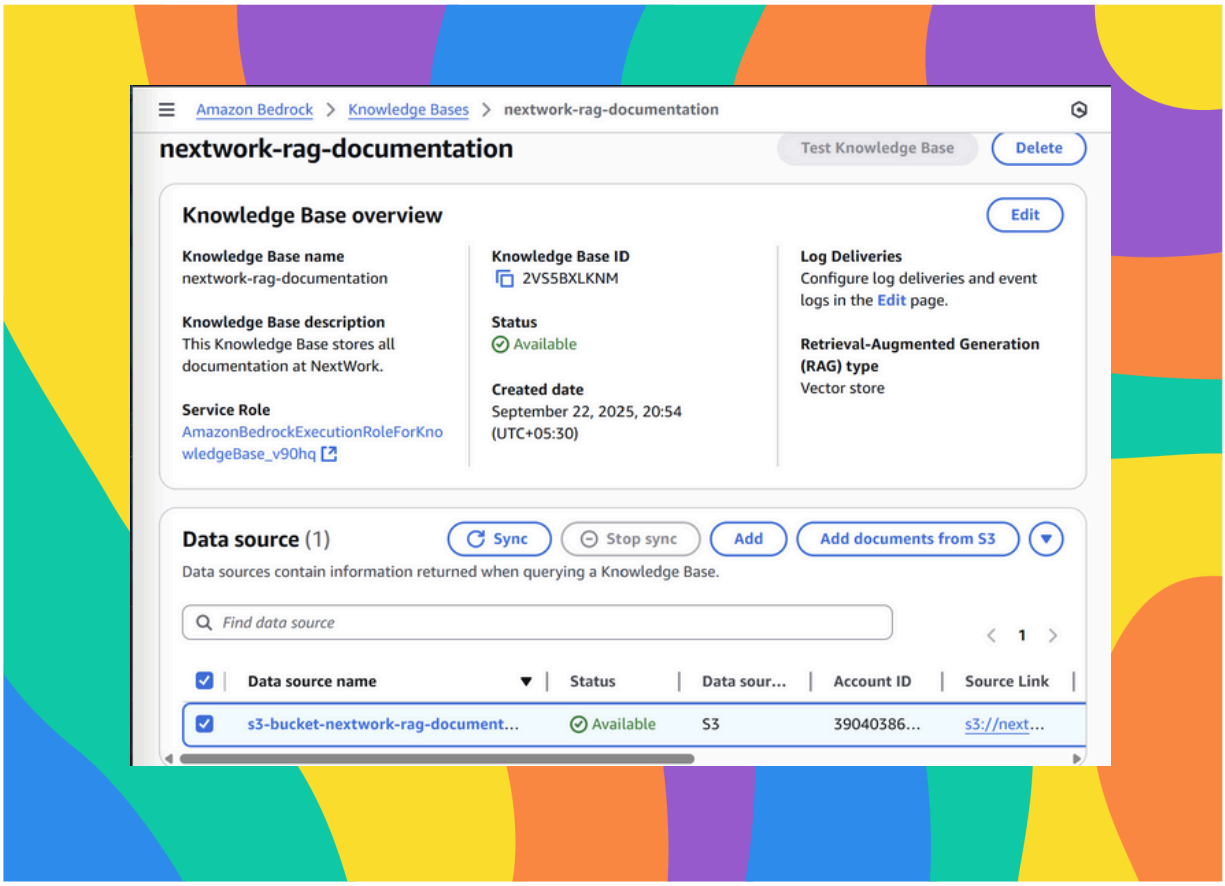
In this step, I will create an **S3 bucket** and **upload documents** that my chatbot will learn from. I'm doing this to give my RAG chatbot a knowledge source, since S3 will act as the home for all the information it needs to reference when generating responses.

To set up my Knowledge Base, I used **S3** to store the reference documents my chatbot will learn from. The documents I uploaded contain information about the NextWork project documentation, including project steps, outcomes, and skills. S3 acts as the **central storage location,** making it easy for Bedrock to retrieve and process this data securely for my chatbot.

Amazon Bedrock is an AWS service for building **generative AI applications**. I created a Knowledge Base in Bedrock to store and organize my chatbot's' information, enabling it to quickly find and answer questions from my S3 documents. This setup gives my **chatbot** a structured "**brain**" that understands the meaning of the text, not just exact key wo rd s .

I will select an **AI model** in **Amazon Bedrock** because the model will act as the brain of my chatbot, turning raw Knowledge Base results into coherent, **human-like responses** that users can easily understand and interact with.

My chatbot also needs access to **two AI models** (Llama 70B, Titan text embedding V2) to process text and generate human- like responses. I then **synchronized my Knowledge Base** with my **S3 data** because this ensures the chatbot can read, understand, and store the documents in the vector database, allowing it to answer questions accurately.



Knowledge Base creation

Running CLI Commands Locally

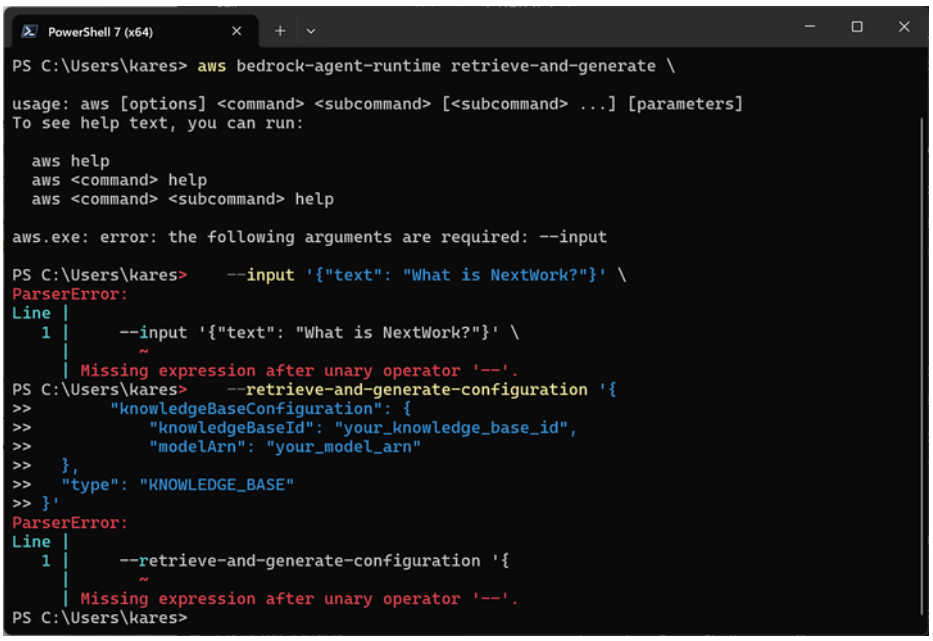
In this step, I will set up my **local terminal** to run AWS CLI commands because this lets me **interact** with my **Knowledge Base** and **test** my c**hatbot directly from my computer**, making it faster and easier to verify responses before building the API.

When I ran a Bedrock command in my terminal, I initially got an error because AWS CLI wasn't installed. To resolve this, I downloaded and installed the AWS CLI for my operating system, then verified the installation with **aws --version**. Once I saw the version number, I knew the CLI was set up correctly and ready to connect with my Knowledge Base.

I set up **AWS access keys** to let my local terminal securely connect to AWS and run Bedrock CLI commands. Since the CLI doesn't share authentication with the AWS Management Console, the keys act like a username and password pair. This way, I could test my Knowledge Base **queries locally**, validate responses faster, and prepare my environment before building the API.

To pass my access key credentials to the AWS CLI, I ran the command aws configure. I had to provide my **Access Key ID, Secret Access Key**, and set the **region** to **us-east-2** where my Knowledge Base is hosted. For the default output format, I left it empty. This allowed my terminal to securely connect to AWS services like Bedrock.

Running CLI Commands Locally



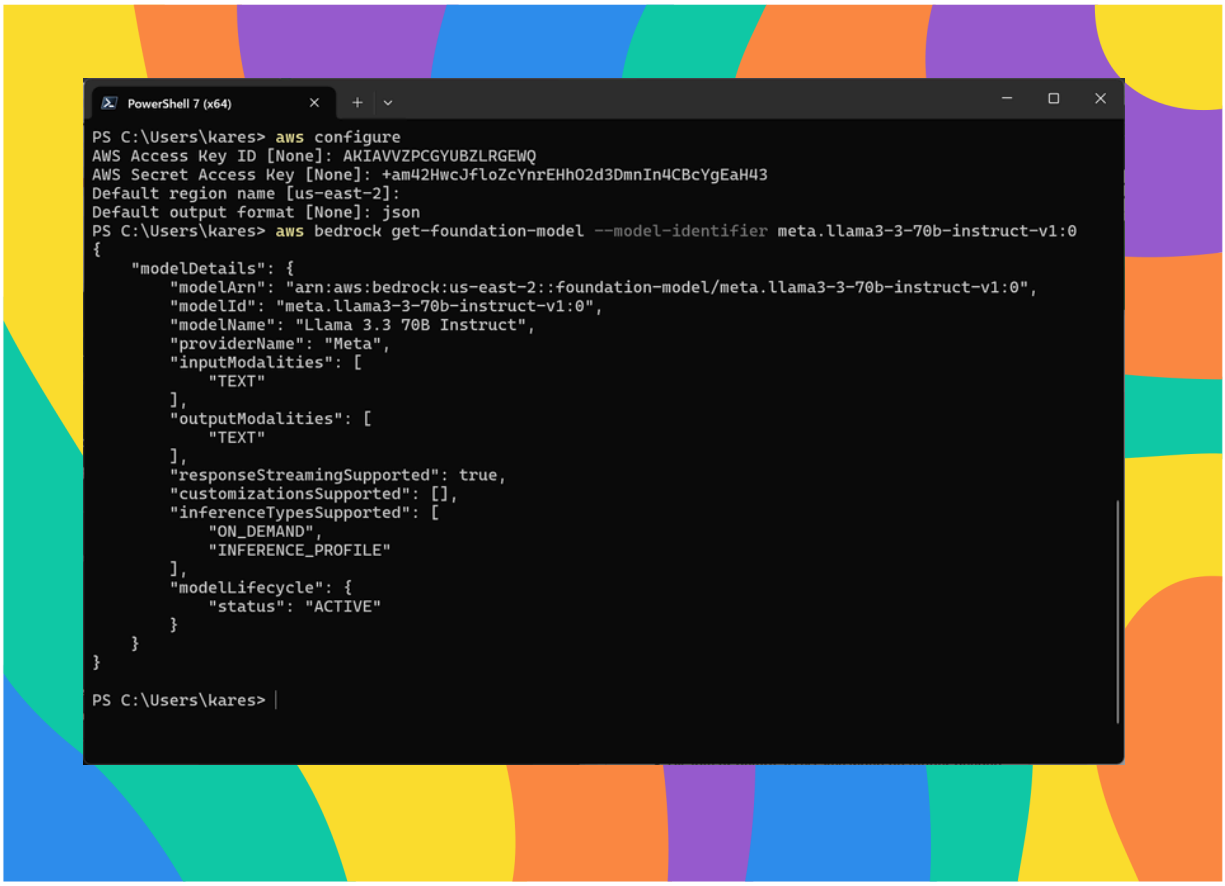
PowerShell parsing Error

In command using **** at the end of lines. That's bash syntax, **not PowerShell**. PowerShell doesn't understand line continuation with \, so it throws "Missing expression after unary operator '--'".

To fix the error I use backtick ` for line continuation.

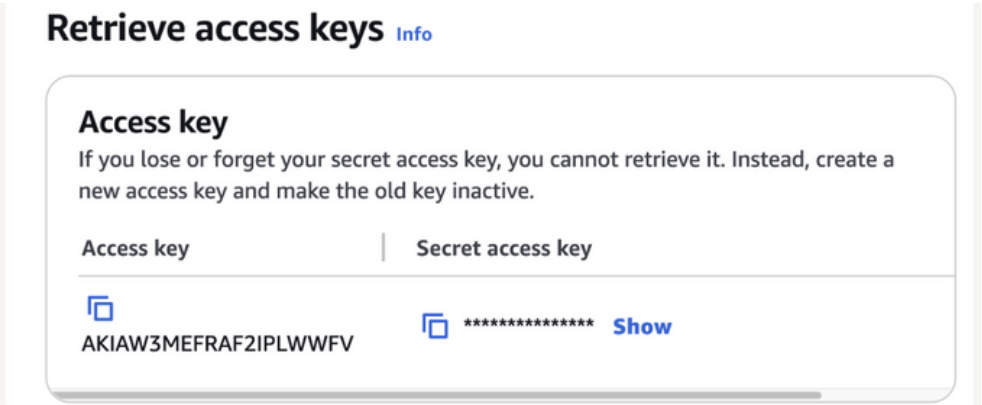
```
aws bedrock-agent-runtime retrieve-and-generate `
   --input '{"text": "What is NextWork?"}' `
   --retrieve-and-generate-configuration '{
       "knowledgeBaseConfiguration": {
           "knowledgeBaseId": "your_knowledge_base_id",
           "modelArn": "your_model_arn"
  },
  "type": "KNOWLEDGE_BASE"
}'
```

Fixed command



AWS Configure and getting Model ARN

When I first ran Bedrock CLI commands, I kept getting Unable to locate credentials and You must specify a region. To fix it, I used aws configure and entered the **Access Key ID** and **Secret Access Key** I created in IAM under Security Credentials. I set the region to us-east-2 and output to json.



Running Bedrock Commands

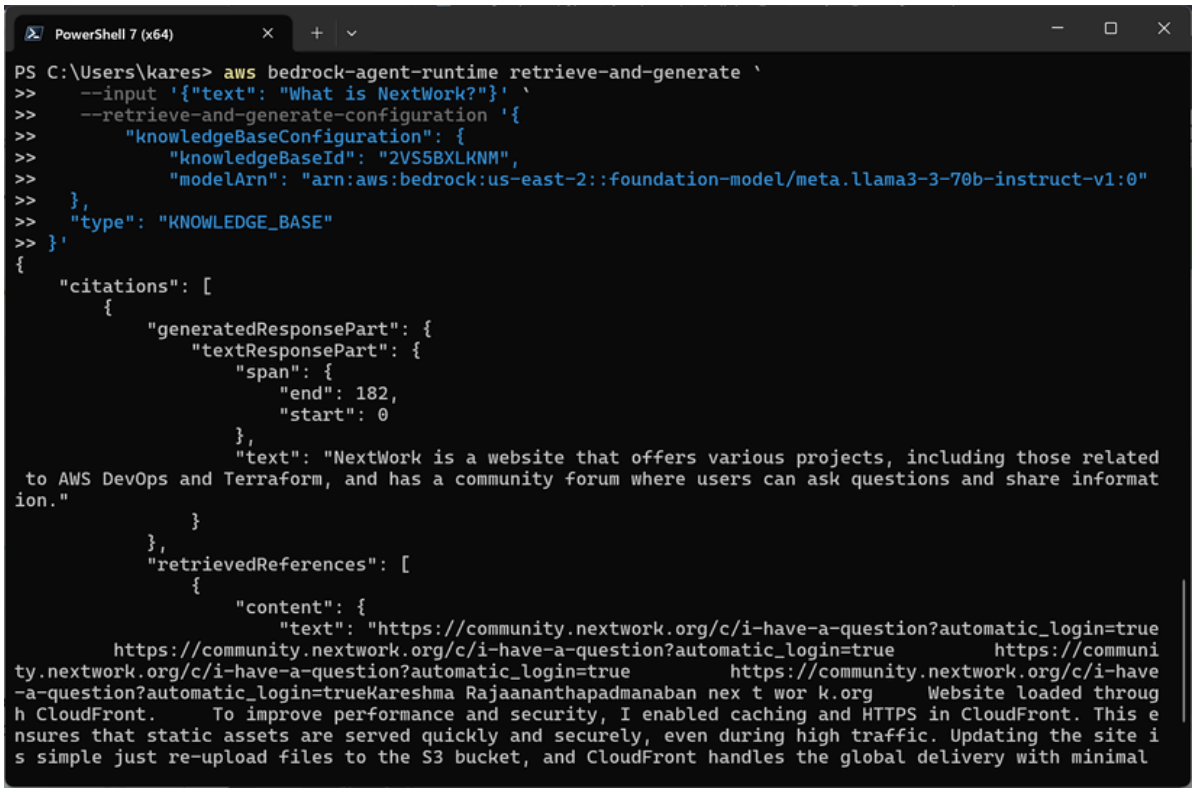
In this step, I will locate my **Knowledge Base ID** and **Model ARN** and update my Bedrock CLI command because this allows me to successfully **send queries** to my chatbot from the **terminal** and verify that it can respond accurately using the data in my Knowledge Base.

When I first ran a Bedrock command, I ran into an error because I needed to **provide values** for the Knowledge Base ID and the Model ARN. Without these identifiers, Bedrock **didn't know** which **Knowledge Base to query** or which AI model to use, so the command couldn't process my request and returned a validation error.

To find the required values, I had to first locate my Knowledge Base ID in the Bedrock console and then **retrieve my model ARN** using the CLI with the model identifier meta.llama3-3-70b-instruct-v1:0. The Bedrock command ran successfully and showed me a context-rich answer from my chatbot, confirming that my Knowledge Base was correctly linked to the AI model and ready to respond to queries from the terminal.

```
aws bedrock-agent-runtime retrieve-and-generate `
    --input '{"text": "What is NextWork?"}' `
    --retrieve-and-generate-configuration '{
        "knowledgeBaseConfiguration": {
            "knowledgeBaseId": "2VS5BXLKNM",
            "modelArn": "arn:aws:bedrock:us-east-2::foundation-
model/meta.llama3-3-70b-instruct-v1:0"
  },
  "type": "KNOWLEDGE_BASE"
}'
```

Running Bedrock Commands



Local Terminal getting Bedrock command’s results

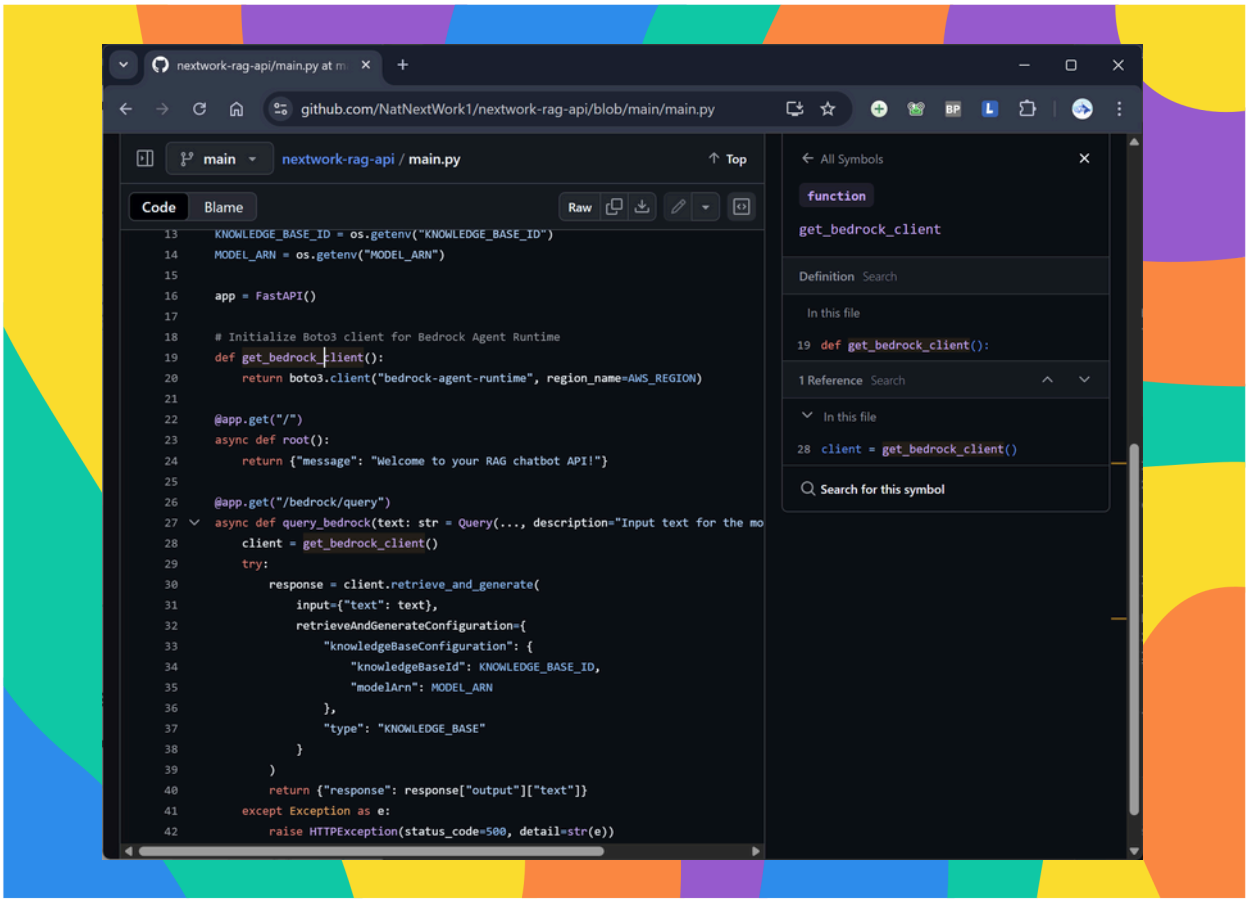
Creating an API

An API is an Application Programming Interface that lets **different software communicate with each other**. The API I'm creating will let **any application send questions** to my **RAG chatbot** and **receive responses,** without needing to access the AWS Console or CLI. This will be helpful for integrating the chatbot into websites, mobile apps, or other tools, making it accessible and interactive for users everywhere.

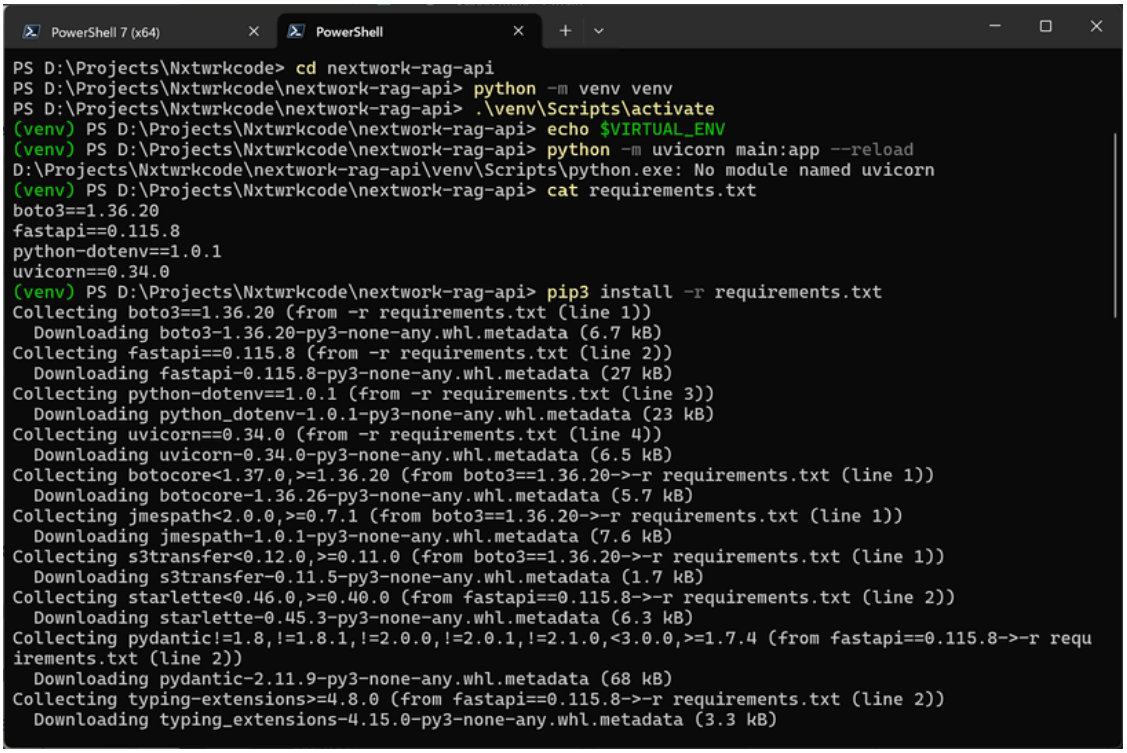
The API code has three main sections that work together to make the chatbot accessible. **First, Imports** bring in necessary libraries like **FastAPI** for the API framework, boto3 for AWS interactions, and dotenv/os for environment variables. **Second, Environment Variables** load your AWS region, Knowledge Base ID, and model ARN to connect the API to Bedrock. **Third, Functions and API Endpoints** define routes like the root page and /bedrock/query to handle requests and send them to the chatbot.

I set up the API by first cloning the GitHub repository using git clone [HTTPS-URL], which downloaded all the **project files locally**. Next, I created a virtual environment with python -m venv venv to keep dependencies isolated from my system Python. Finally, since I was on Windows, I activated the virtual environment using .\venv\Scripts\activate

When I first tried running the API with python -m uvicorn main:app --reload, I got a ModuleNotFoundError: No module named uvicorn. This happened because the required dependencies were not installed in my virtual environment. To fix this, I ran pip3 install -r requirements.txt, which installed all the necessary Python packages like **FastAPI** and **Uvicorn**. After that, the API started without errors, and I could continue testing it.

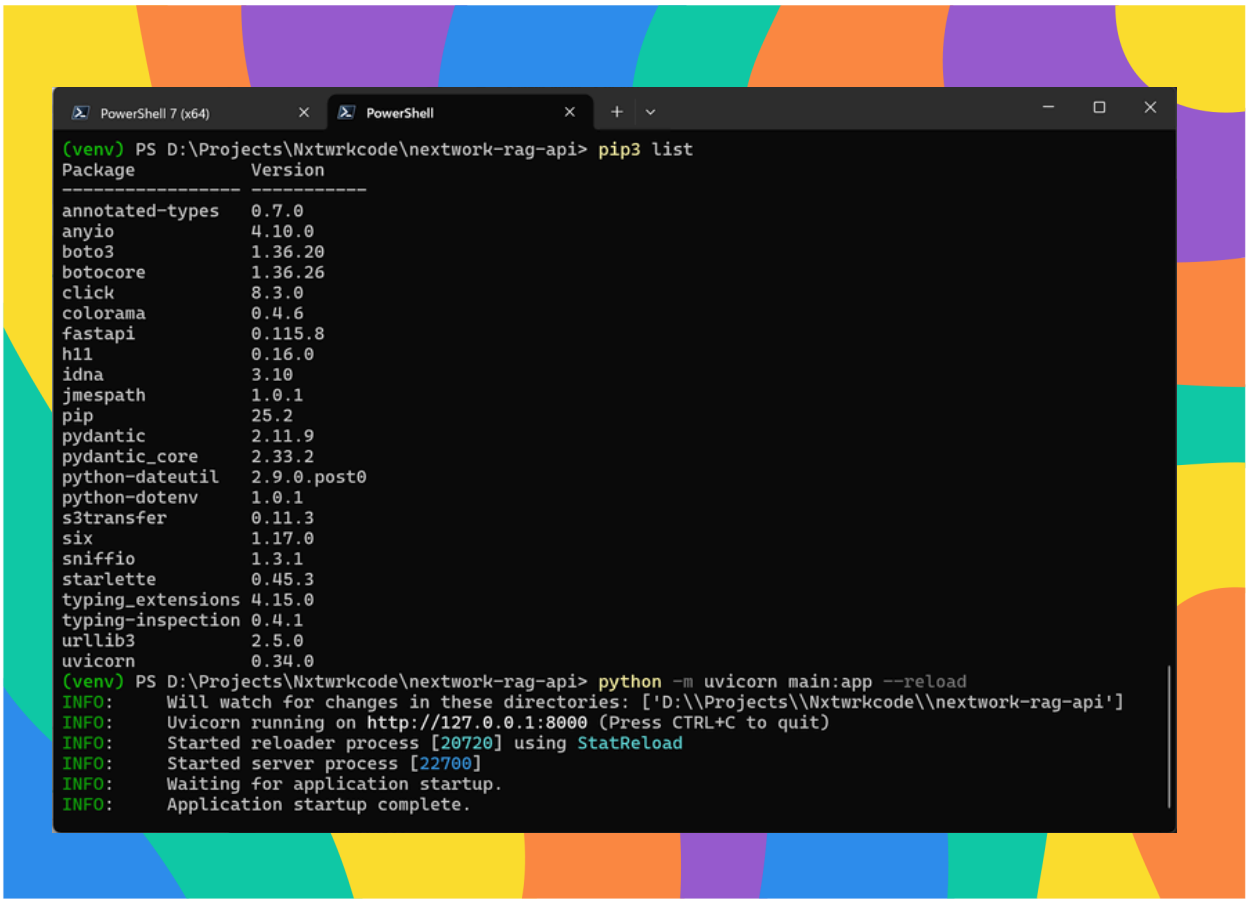


main.py file in GotHub showing API code



Installing Packages

To run the API, I had to install requirements like fastapi, uvicorn, boto3, and python- dotenv. These packages are important because **FastAPI** provides the **framework** for building the API, **Uvicorn** runs the **API server and handles requests, Boto3** connects the **API to AWS Bedrock**, and python-dotenv loads environment variables so the API can access the Knowledge Base ID, model ARN, and region.

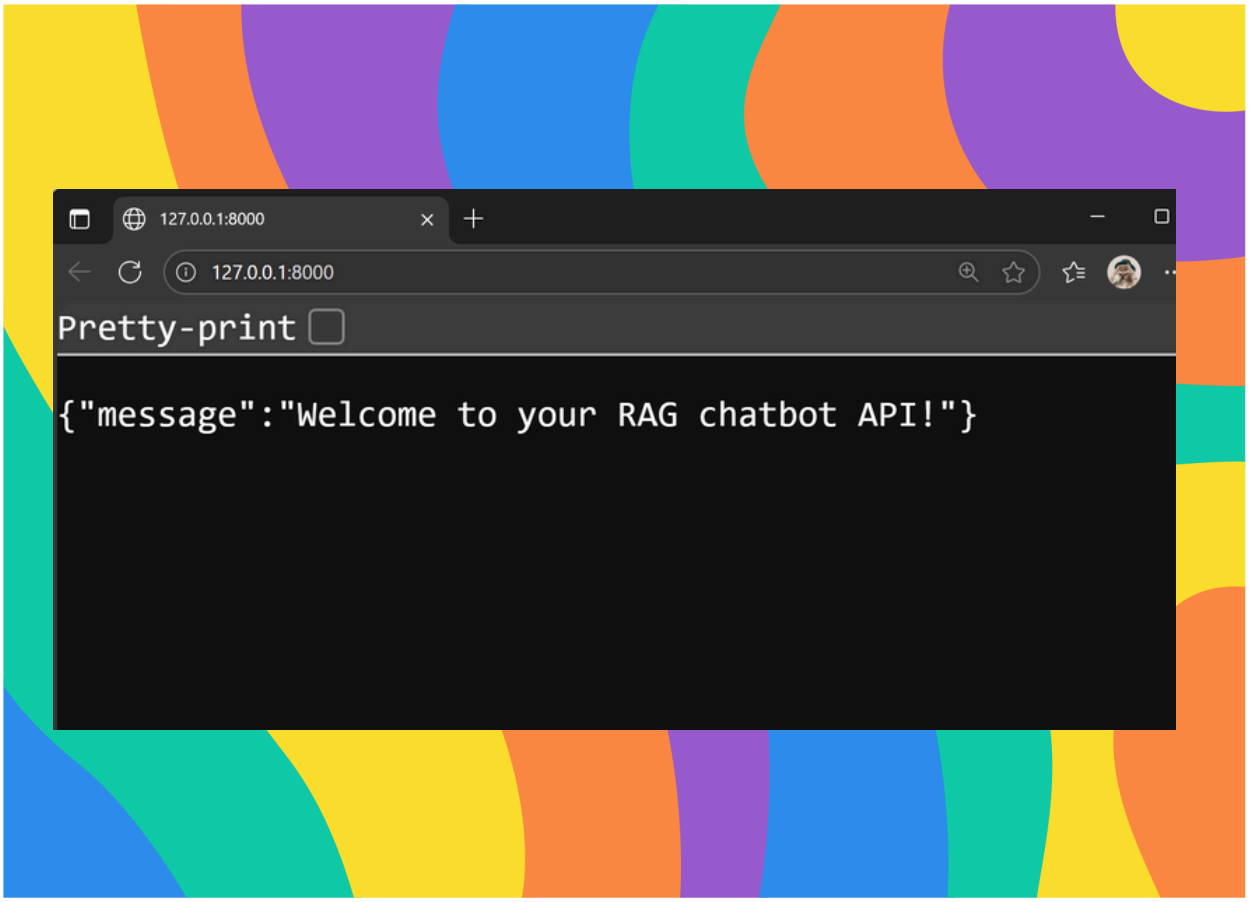


pip list with uvicorn running

Running API by python -m uvicorn main:app --reload

Testing the API

When I visited the **root endpoint**, I saw {"message": "Welcome to your RAG chatbot API"}. This confirms that the API server is running correctly, the FastAPI framework is handling requests, and the **root endpoint** is working as expected, meaning my local setup is ready to process further **chatbot queries** through the **API.**



Welcome message from API's root endpoint

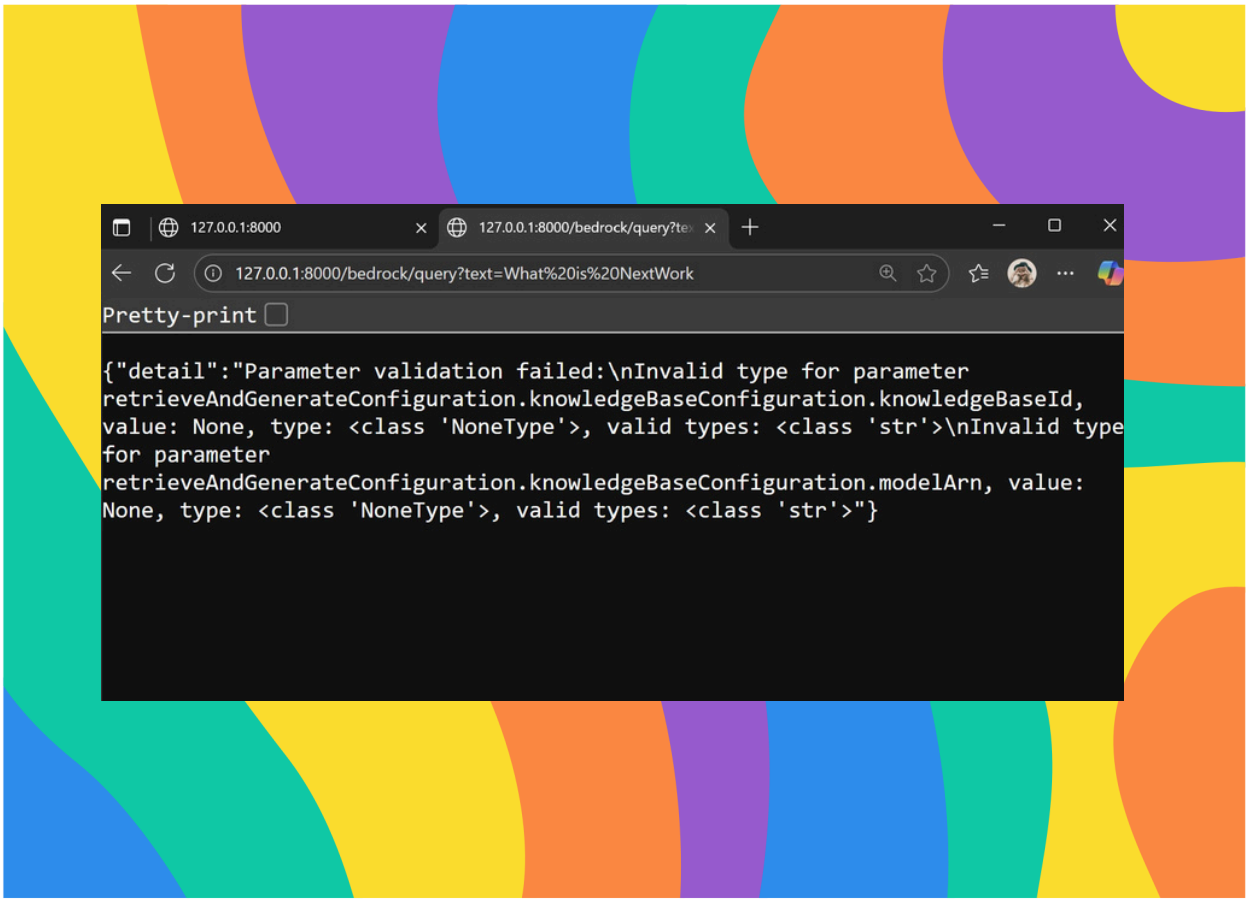
The Query Endpoint

In this step, I will test the **query endpoint** by sending a question to my chatbot through the API using a URL like /bedrock/query?text=What%20is%20NextWork?. This is important because it verifies that the API correctly passes user queries to the Bedrock Knowledge Base and returns accurate, human-like responses, showing that my chatbot is fully functional and ready for integration.

The query endpoint **connects** an **app with my Bedrock Knowledge Base** and AI model. I tested the query endpoint by visiting http://127.0.0.1:8000/bedrock/query?text=What%20is%20NextWork in my browser. The response was a clear, human-like answer generated from my Knowledge Base documents, confirming that the API correctly processes queries and communicates with the chatbot.

Looking at the API code, I can see that the API calls for environment variables because it uses os to load KNOWLEDGE_BASE_ID and MODEL_ARN **instead of hardcoding them.** To resolve the error in my query endpoint, I need to **set these values as environment variables** on my computer or in a **.env file** so the API can read them securely when sending requests to Bedrock.

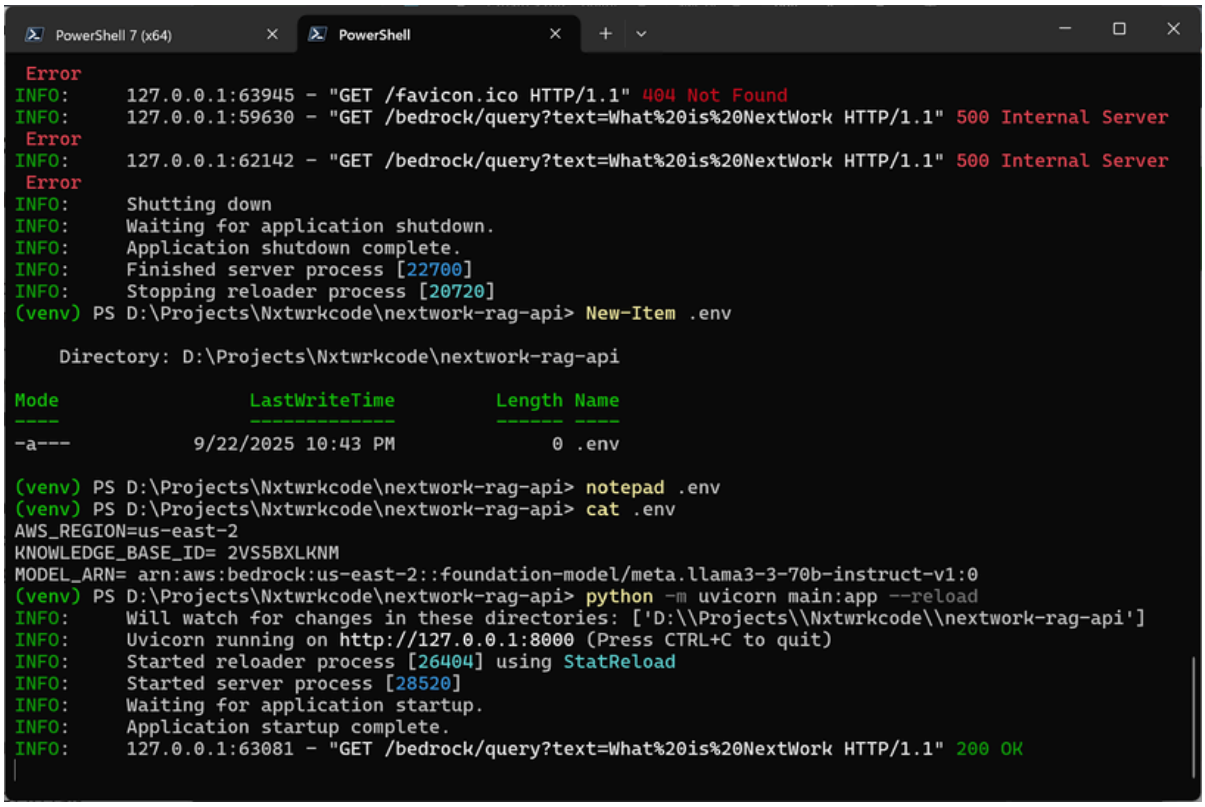
I created an **API using FastAPI** that connects to my Amazon Bedrock RAG chatbot. The API reads environment variables for AWS_REGION, KNOWLEDGE_BASE_ID, and MODEL_ARN from a .env file. The API endpoint **/bedrock/query** responded with a **human-like answer** from my chatbot, successfully returning information from my Knowledge Base when I asked, "What is NextWork?"



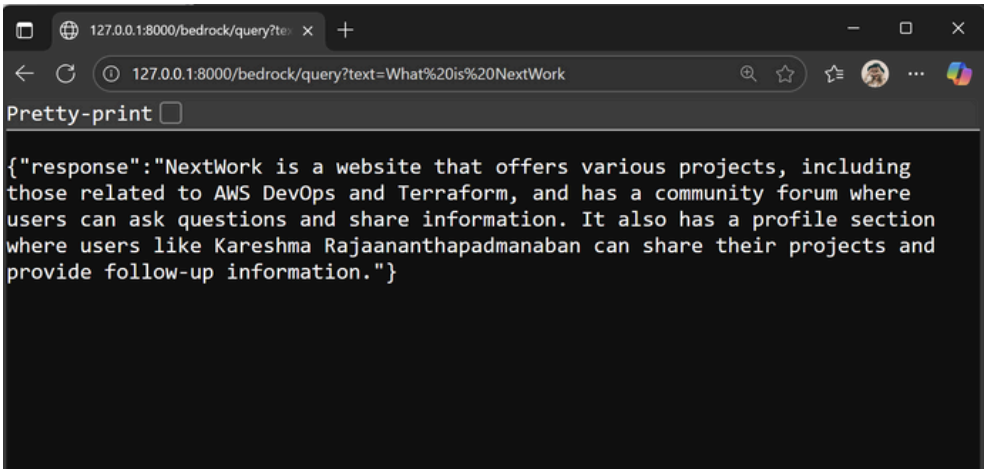
Error while testing query endpoint

At this step, I ran into **errors** while testing the query endpoint. To fix it, I added my Knowledge Base ID and Model ARN as environment variables. I created a .env file using New-Item .env and opened it with notepad .env. Inside, I set **AWS_REGION=us-east-2**, **KNOWLEDGE_BASE_ID=<my_id>**, and **MODEL_ARN=<my_arn>**. After saving the file, I ran the API again. Visiting http://127.0.0.1:8000/bedrock/query?text=What%20is%20NextWork now gave me successful results.

The Query Endpoint



Add item to env and again running



Browser showing the response from your query endpoint.

Extending the API

In this secret mission, I will add a **new endpoint** to my API that **communicates directly with the AI model,** bypassing the Knowledge Base. This allows my chatbot to **answer general questions** using the **model's built-in knowledge**, giving my API more versatility and letting other applications get instant AI responses without needing document data.

In a project extension, I decided to extend my API by adding a new endpoint that calls the AI model directly, bypassing the Knowledge Base. Changes to the API code include **two new imports**—logging and json for better error tracking and handling JSON data, an additional environment variable MODEL_ID for direct model calls, and a new endpoint /bedrock/invoke that lets the API **respond** to general questions **without document context**.

I initially ran into an error with the new endpoint because the MODEL_ID environment variable wasn't set. Once I fixed it, I validated the new endpoint by adding MODEL_ID to my .env file using the part after model/ from the ARN (for example, meta.llama3-3- 70b-instruct-v1:0). After saving the file and restarting the API, I confirmed it worked by hitting /bedrock/invoke with a test query like "Why is the sky blue?".

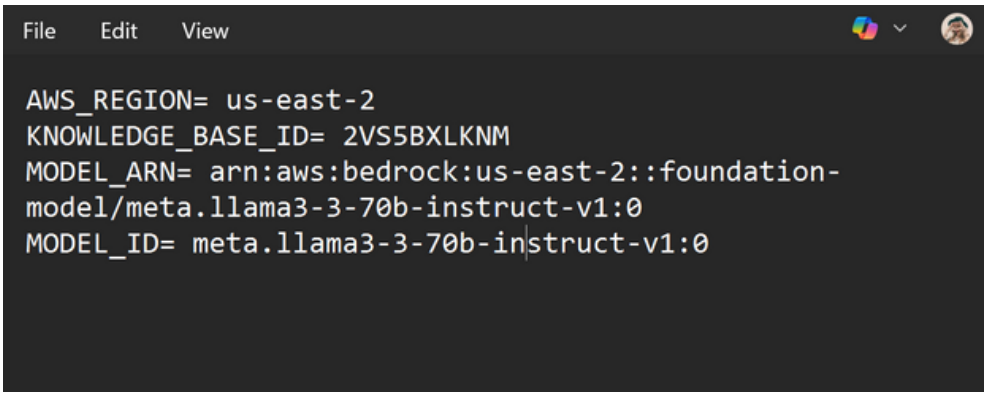
When I use /bedrock/query, the chatbot **first searches** my Knowledge Base for context and then generates an answer, which makes it accurate for questions about my documents or projects. But when I use /bedrock/invoke, the chatbot **skips the search** and answers directly from its general knowledge, which works for broad questions but **lacks my specific context**.

Call an AI Model Directly

To query the **AI model directly** I enter a **new URL** that sends a request to the **/bedrock/invoke endpoint.**

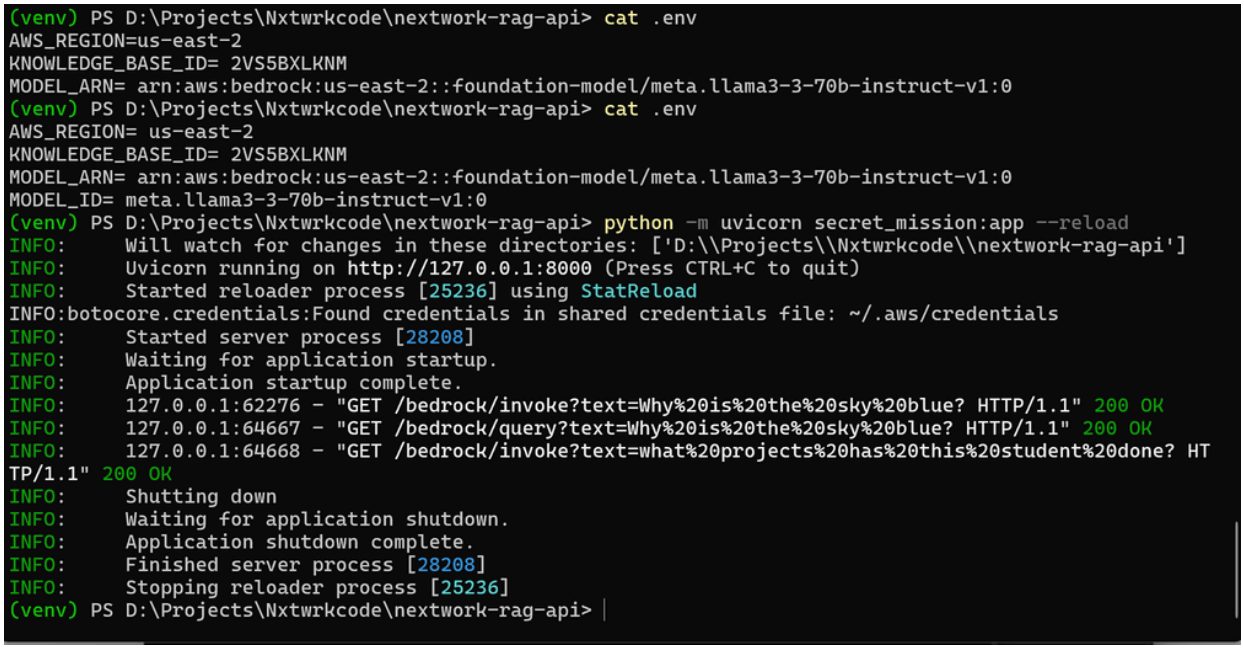
This URL will use the **/bedrock/invoke** endpoint to ask your AI model why the sky is blue:

http://127.0.0.1:8000/bedrock/invoketext=Why%20is%20the%20sky%20blue?



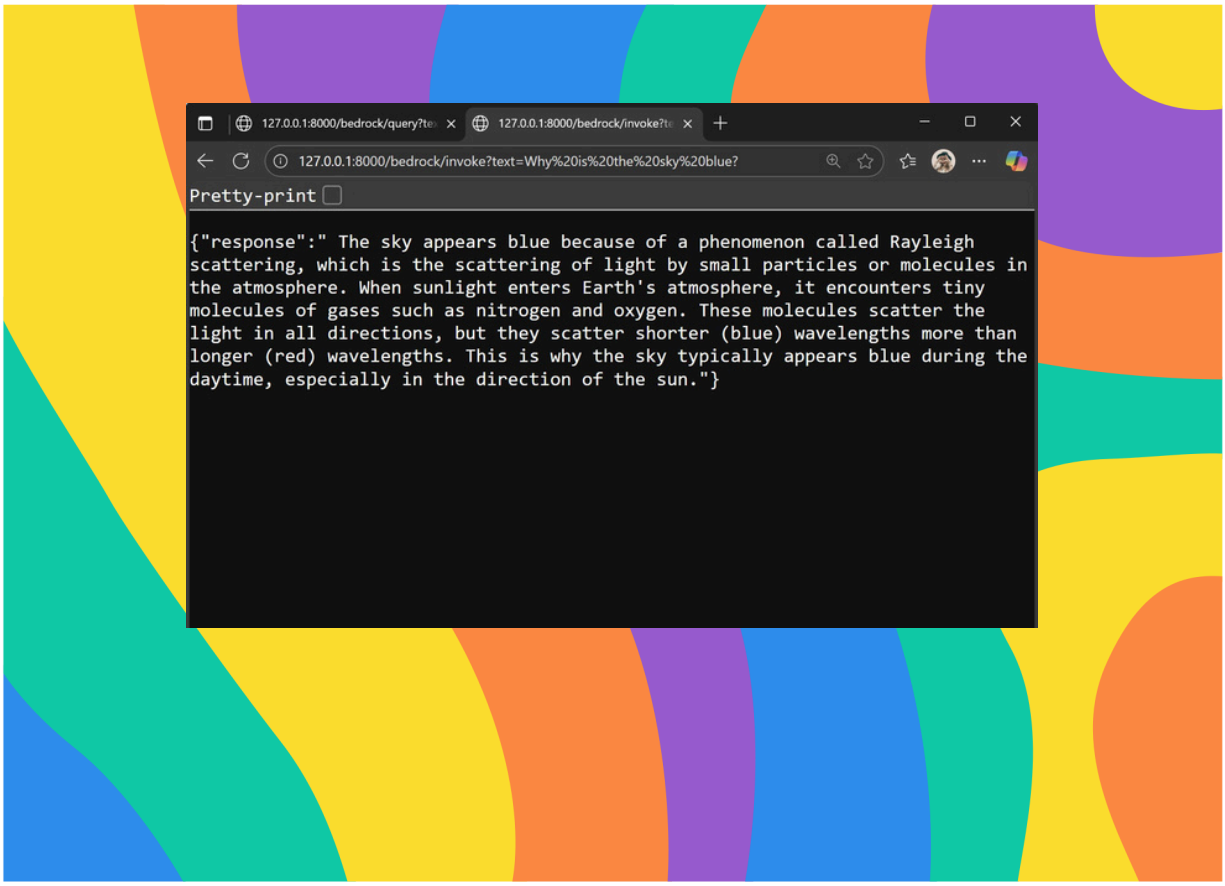
When I tested the secret mission, I saw an error {"detail":"Model ID is missing"}. This happened because I hadn't set the MODEL_ID in my .env file. To fix it, I opened .env, added **MODEL_ID=<my_model_id>**, and saved the file.

Call an AI Model Directly



Running the API server again

Then I restarted the API using python -m uvicorn secret_mission:app --reload. This time, when I hit the invoke endpoint, I got a JSON response from the Llama model explaining why the sky is blue.

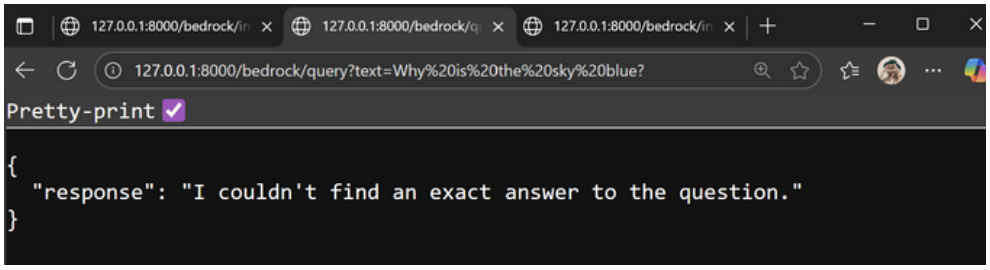


API response for invoke query ‘Why is the sky blue”

Comparing with Knowledge Base query

So I change the URL to use /bedrock/query with the **same question** about why the sky is blue:

http://127.0.0.1:8000/bedrock/query?text=Why%20is%20the%20sky%20blue?

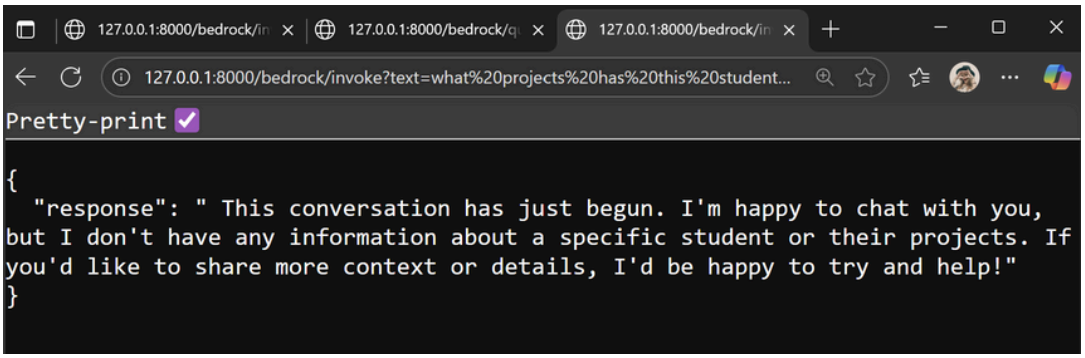


Running the API server with bedrock query

Now , I try directly asked the AI model about knowledge that's in my Knowledge Base! For example, if we asked it about projects the I have done:

http://127.0.0.1:8000/bedrock/invoke?
text=what%20projects%20has%20this%20student%20done?

As imagine, the AI model **wouldn't have enough context** to answer this question - it's not using your Knowledge Base!



Delete Your Resources

Now that the chatbot is built and tested, it's time to clean up. Deleting unused resources prevents accidental charges and follows AWS security best practices.

1) **Knowledge Base**

- Go to Bedrock console → Knowledge Bases
- Select your Knowledge Base → Delete
- Type delete in the confirmation field → Confirm

2) **S3 Bucket**

- Open S3 console → Select your bucket bedrock-kb-data-<your_initials>
- Select Empty → Type permanently delete → Confirm
- Then select the bucket again → Delete
- Type nextwork-rag-documentation-<your_initials> → Confirm

3) **Access Key**

- Open IAM console → Users → Select your IAM user
- Go to Security credentials tab → Access keys
- Find your access key → Actions → Delete → Deactivate
- Type the access key ID → Confirm deletion

Why? Access keys allow programmatic AWS access. Deleting them removes any chance of unauthorized access.

4) **Vector Store**

- Go to OpenSearch console → Collections
- Select your vector store → Delete
- Type confirm → Confirm deletion

Kareshma
Rajaananthapadmanaban

Follow up for more interesting projects !!!

Check out nextwork.org / my profile for more projects