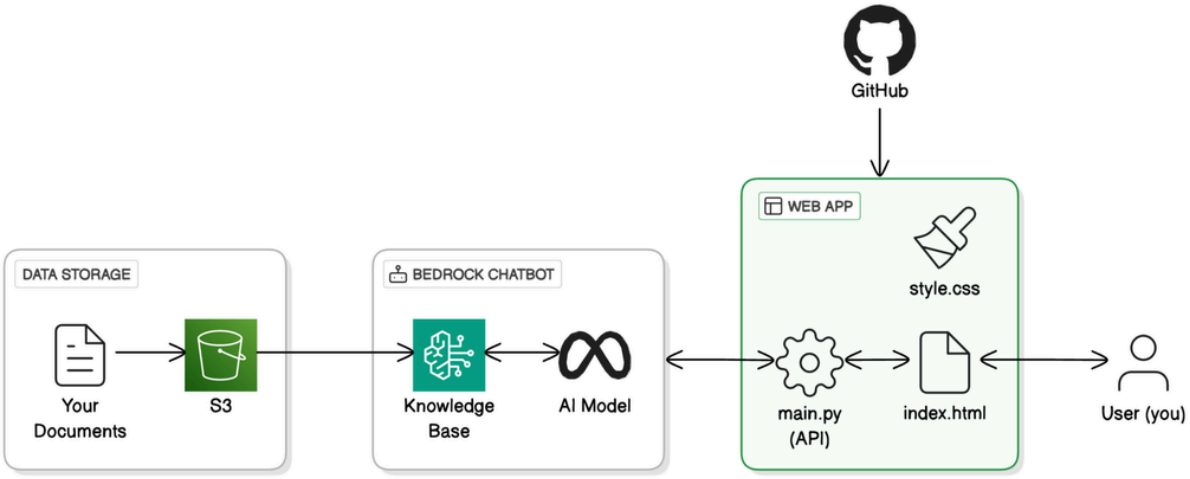
Kareshma Rajaananthapadmanaban

Build a Web App for RAG Chatbot

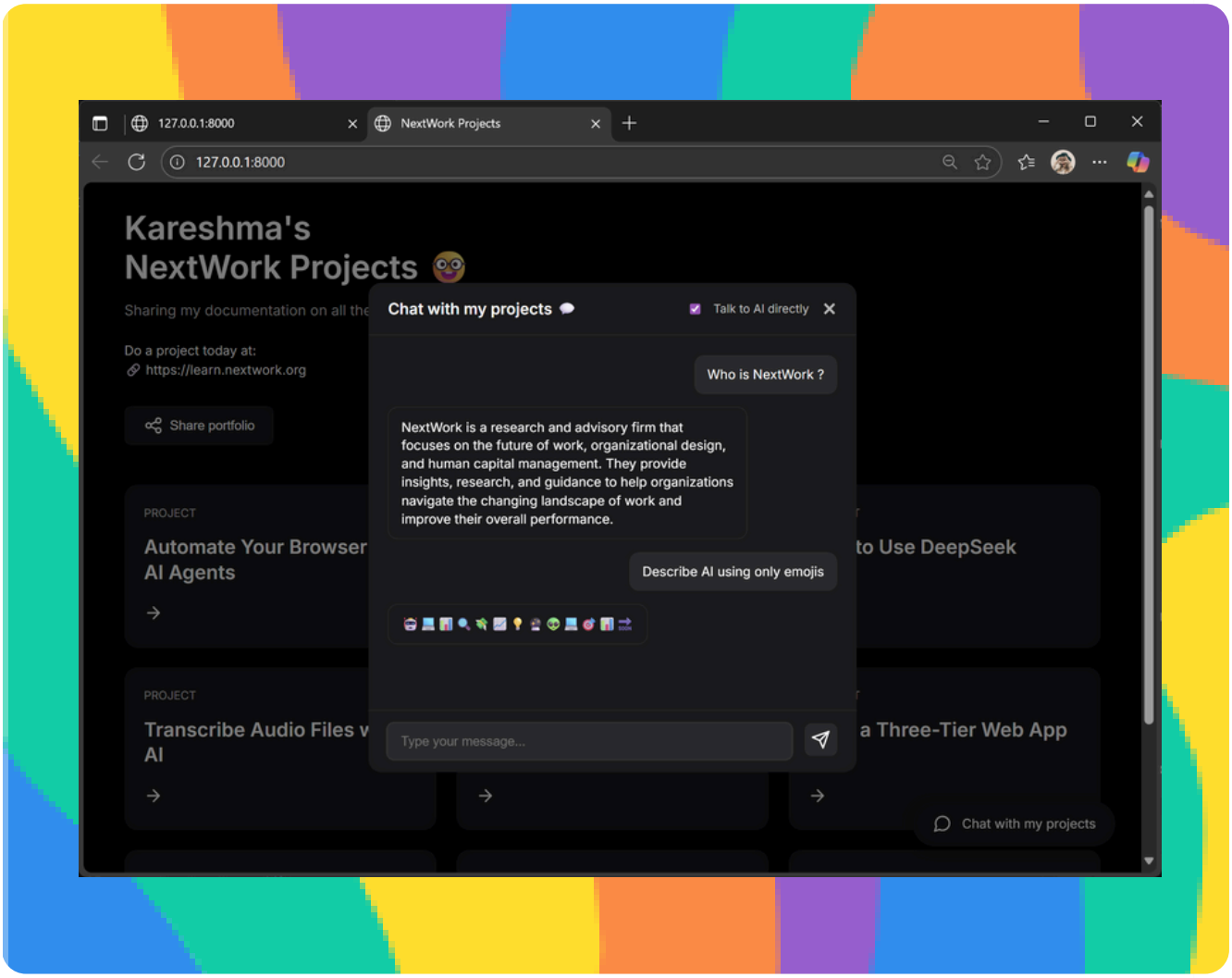
Let's build a web app that makes it easy to connect and chat with my RAG chatbot!



nextwork.org

Building a RAG Chatbot with a Web Interface

Kareshma Rajaananthapadmanaban



Introducing Today's Project!

In this project, I will demonstrate how to build a **fully functional web app** for my **RAG chatbot**. I'm doing this project to learn how to connect a **frontend chat interface** with an **API-powered backend** that uses **Amazon Bedrock** and a **knowledge base** in **S3**. This final step is about making the chatbot accessible, scalable, and user-friendly, while gaining hands-on skills in web development and AI integration.

Tools and concepts

Services I used were Amazon Bedrock for model inference, **AWS IAM** for secure access, and **FastAPI** with **Uvicorn** to build and run the chatbot API. I also worked with **dotenv** to manage environment variables and integrated a web frontend to interact with the backend. Key concepts I learnt include Retrieval-Augmented Generation (RAG), separating frontend and backend layers, secure credential handling, and how **enabling or disabling** RAG changes chatbot behavior.

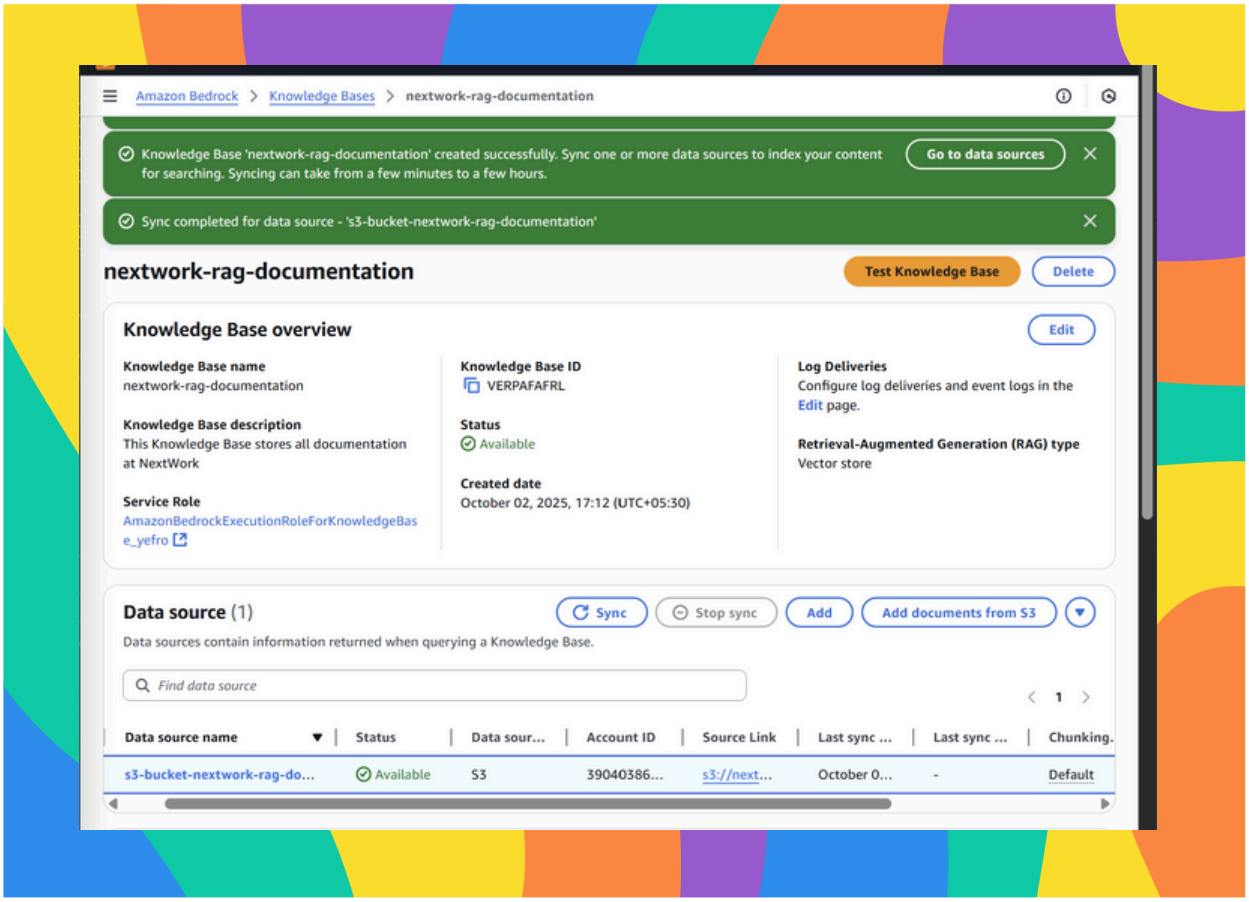
Project reflection

This project took me approximately 2 hours and 30 minutes. The most challenging part was **troubleshooting API integration** and ensuring the chatbot responded correctly with and without RAG enabled. It was most rewarding to see the chatbot pull accurate answers from my knowledge base and experience the difference in responses when switching between **general AI knowledge and RAG-based retrieval.**

I did this project today to practice building with Amazon Bedrock and get hands-on with **Retrieval-Augmented Generation concepts**. My goal was to understand how different AWS services connect to create a working chatbot with a **custom knowledge base**. The project met my goals because I learned how to set up, test, and clean up the entire workflow while strengthening both my **cloud and AI integration skills**.

Chatbot setup

In this step, I will create an **S3 bucket** to store the **documents** my chatbot will use as its knowledge source. I'm doing this to give my RAG chatbot a foundation of information it can reference when generating answers. By uploading my documents to S3 and linking them to a **Bedrock Knowledge Base,** I ensure the chatbot can retrieve accurate, context-aware responses based on my own data.



I created a **Knowledge Base** to give my **RAG chatbot a structured way** to store, organize, and retrieve information from my documents in S3. This is important because without a Knowledge Base, the chatbot wouldn't know how to process or search through the data. By using **embeddings** and a **vector store** in Bedrock, my chatbot can understand context, match questions to relevant content, and deliver accurate, meaningful answers.

Setting Up the API

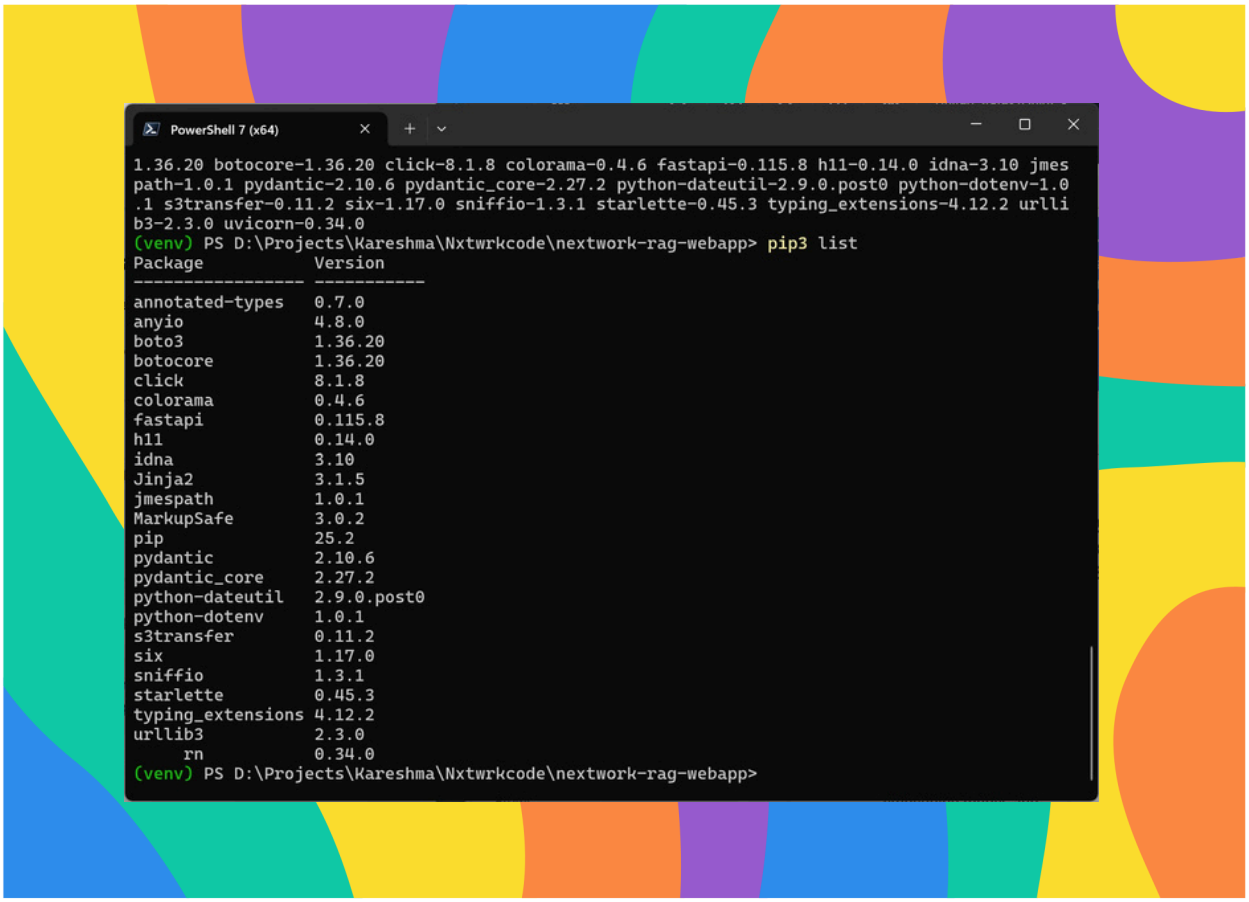
In this step, I will **clone** a **pre-built web app** from GitHub to my local machine. I'm doing this to quickly set up both the frontend and backend code needed for my RAG chatbot without writing everything from scratch. This starter code includes the **chat interface, API handling**, and **integration points** with AWS, so I can focus on connecting it to my Knowledge Base and customizing it.

A **virtual environment** helps me by keeping all the **Python dependencies** for this project **isolated** from my **global setup**. This means any libraries I install for the web app won't affect other projects on my computer.

It also makes the project more portable and reproducible, since I can recreate the same environment anywhere. For this RAG chatbot web app, it ensures clean, conflict-free package management.

I will **test** and **run** the **backend code** of my **RAG web app locally**. I'm doing this to make sure the backend can actually connect to my Bedrock Knowledge Base and handle requests before I combine it with the frontend. Verifying the backend works first helps me catch issues early and ensures my app has a solid foundation before moving forward.

To run the API, I had to install requirements like **fastapi, uvicorn, boto3**, and **python-dotenv**. These packages are important because FastAPI builds the API itself, Uvicorn runs the server so my API is accessible, Boto3 connects my app to AWS Bedrock, and python-dotenv loads environment variables securely. Together, they make my chatbot's backend run smoothly and talk to AWS.



Successful Installation of Required Packages

Breaking down the API

In our web app, we need an **API** because it acts as the **bridge** between the **frontend chat interface** and the **backend logic** that talks to Amazon Bedrock. The frontend alone can't process or fetch answers it just displays them. The API handles requests from the user, connects to Bedrock, processes responses, and then sends them back to the frontend for display.

Digging deeper into the API code, the main tools we need to import are **FastAPI, boto3, os**, and **load_dotenv**. These tools help us by letting us build the API quickly (FastAPI), connect to AWS services like Bedrock (boto3), securely access environment variables (os), and load them from a file (load_dotenv) without hardcoding sensitive information.

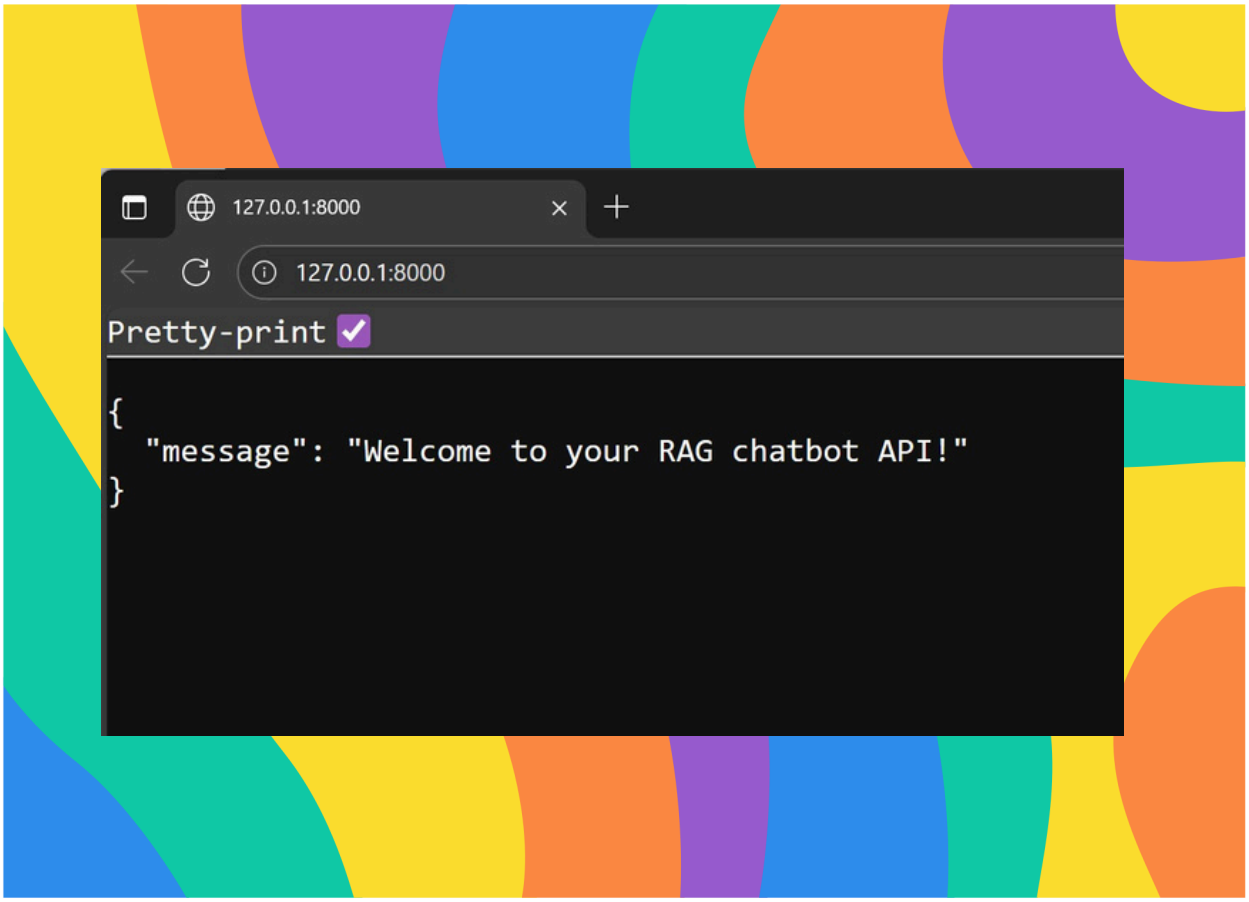
The environment variables we need are AWS_REGION, KNOWLEDGE_BASE_ID, and MODEL_ARN. We store them separately because they contain sensitive information like credentials and IDs. Keeping them in a **separate .env file protects** this information from being exposed in our code or accidentally pushed to GitHub.

My API uses the Python **AWS SDK** to connect to **Bedrock** from within the code. The difference between the AWS CLI and an SDK is that the CLI is a command-line tool for manual tasks, while an SDK is a **library** for building applications that programmatically interact with AWS services. SDKs allow seamless integration into code, unlike the CLI.

Our API has two main routes: / is the **root route** that returns a welcome message, and /bedrock/query is where **users send questions** to the chatbot. In general, routes help us define paths in the API that perform specific functions, making it possible for different endpoints to handle different types of requests.

Testing the API

When I visited the root endpoint, I saw `{"message": "Welcome to your RAG chatbot API"}`. This confirms that my **API server** is **running correctly** and responding to requests from the browser, which means the backend setup is working as expected.



Troubleshooting the API

In this step, I will **test the chatbot** by sending a **question to the API** using the **/bedrock/query endpoint**. This is important because it lets me verify that the API can successfully retrieve answers from our Bedrock Knowledge Base and ensures the backend is fully functional before connecting it to the frontend chat interface.

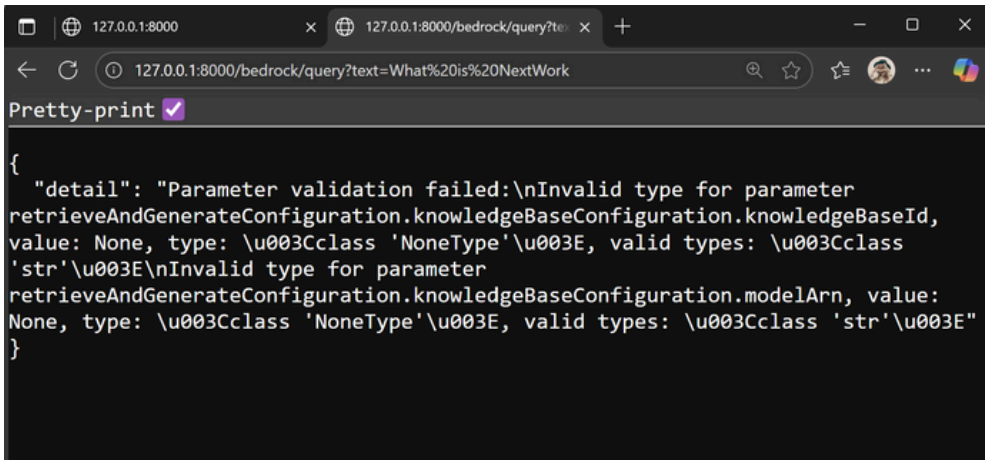
My second endpoint had errors!

To fix the Parameter Failed error, I had to create a **.env** file in my project directory and add the correct environment variables: AWS_REGION, KNOWLEDGE_BASE_ID, and MODEL_ARN. Doing this helps because it ensures the API knows which AWS region, Knowledge Base, and AI model to use, preventing invalid parameter errors when querying Bedrock.

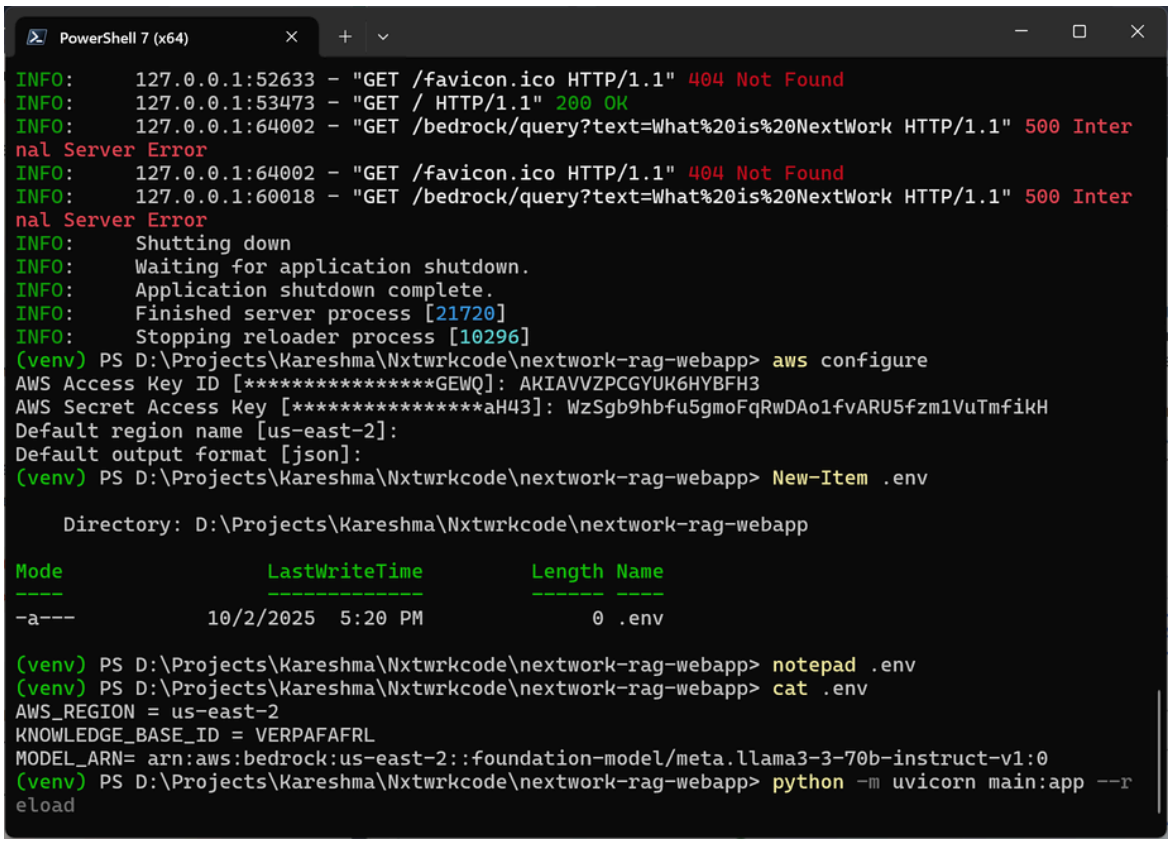
To fix the credentials error, I ran aws configure in my terminal and entered my IAM user's Access Key ID, Secret Access Key, and the correct AWS region (us-east-2). This was necessary because the AWS CLI needs these credentials to authenticate my terminal commands and allow my API to communicate securely with Bedrock.

My chatbot initially failed to generate responses, so I made sure I **had requested access** to the Titan Text Embeddings V2 and Llama 3.3 70B Instruct models in Bedrock, then **synced** my Knowledge Base with the S3 bucket containing my documents. This should help the AI properly process my data and generate accurate, meaningful answers.

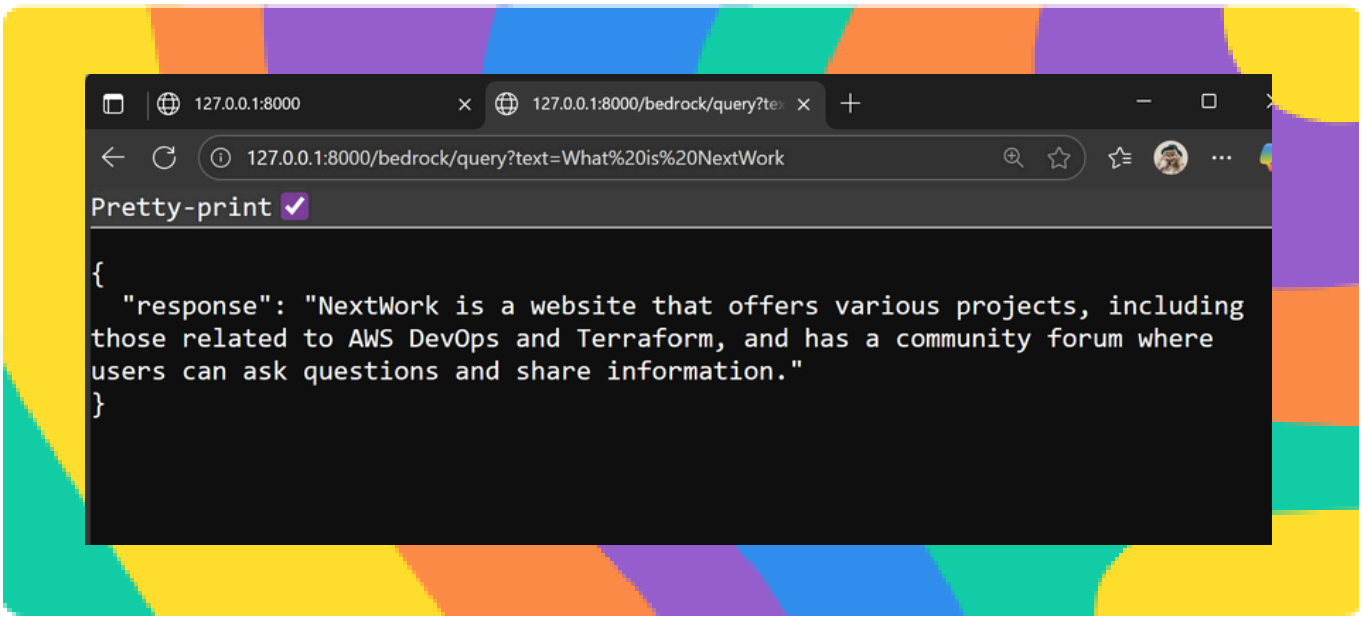
A error message for **Invalid type for parameter** is related to **KnowledgeBaseID** or **modelArn** means API isn't correctly configured with the Knowledge Base ID and Model ARN



Parameter failed error



Parameter failed error set up



API’s successful response to user’s question

Running the Web App

In this step, I will **run** the web app using **web_app.py**, set the **MODEL_ID** environment variable in my **.env file**, and **interact** with the **chatbot** through the browser. I'm doing this to see how the full web interface works, **test responses with and without RAG,** and understand how the frontend, backend, and data layer work together in real time.

The difference between the API and the web app for the user is how they interact with the chatbot. The **API** gives **raw JSON responses** when you hit endpoints like /bedrock/query, which is useful for testing and backend integration.

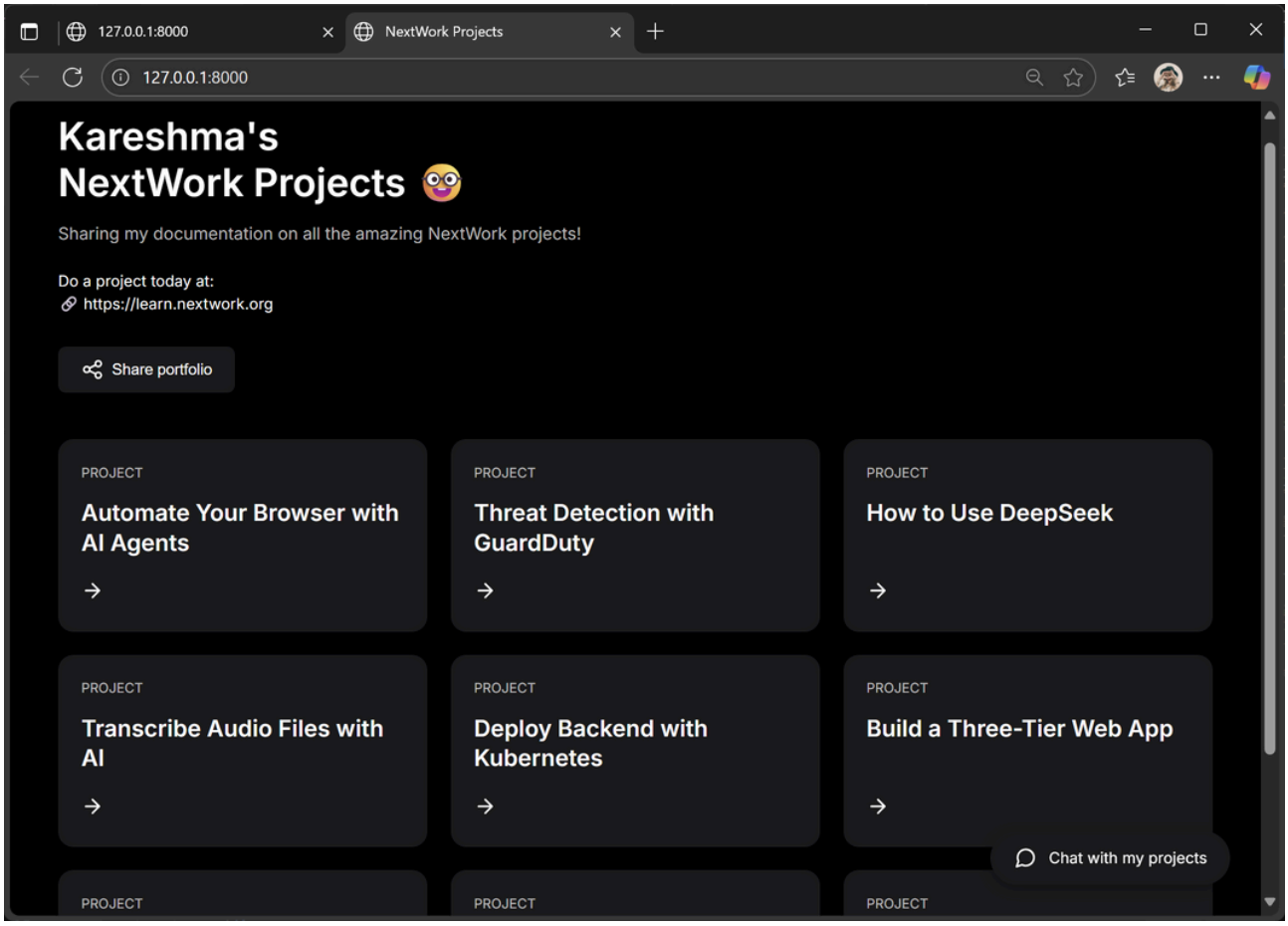
The **web app**, on the other hand, provides a **full chat interface in the browser** where users can type questions and see responses in a **conversational format**.

The **web app** has the **ability to switch** between using **RAG with my Knowledge Base** and **talking directly to the AI model**.

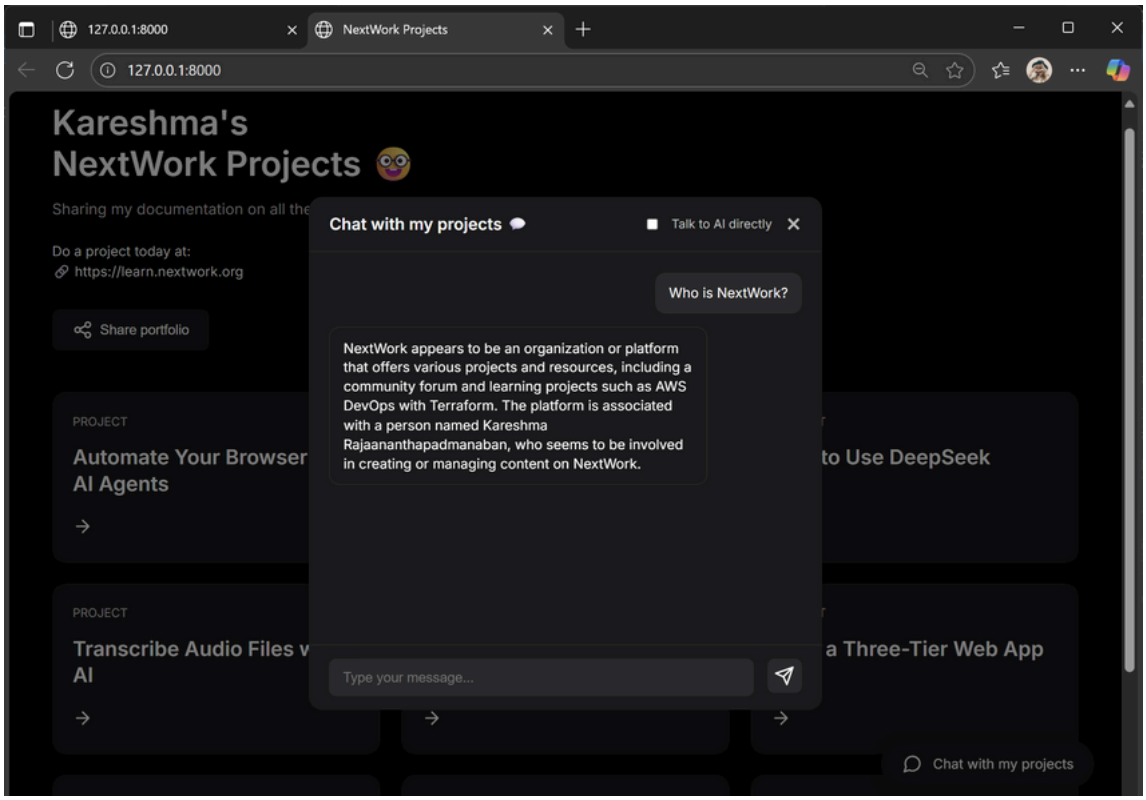
When I asked the same question, “Who is NextWork?”, with **RAG enabled** it gave me a **grounded answer** based on my documents. When I checked “**Talk to AI directly**,” the response came only from the **model’s general knowledge**, which was less accurate.

```
INFO:     Started server process [19408]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     127.0.0.1:60355 - "GET /bedrock/query?text=What%20is%20NextWork HTTP/1.1" 200 OK
INFO:     Shutting down
INFO:     Waiting for application shutdown.
INFO:     Application shutdown complete.
INFO:     Finished server process [19408]
INFO:     Stopping reloader process [3556]
(venv) PS D:\Projects\Kareshma\Nxtwrkcode\nextwork-rag-webapp> python -m uvicorn web_app:app
--reload
INFO:     Will watch for changes in these directories: ['D:\\Projects\\Kareshma\\Nxtwrkcode\\
nextwork-rag-webapp']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [16392] using StatReload
INFO:botocore.credentials:Found credentials in shared credentials file: ~/.aws/credentials
INFO:     Started server process [1460]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Web app running on Terminal



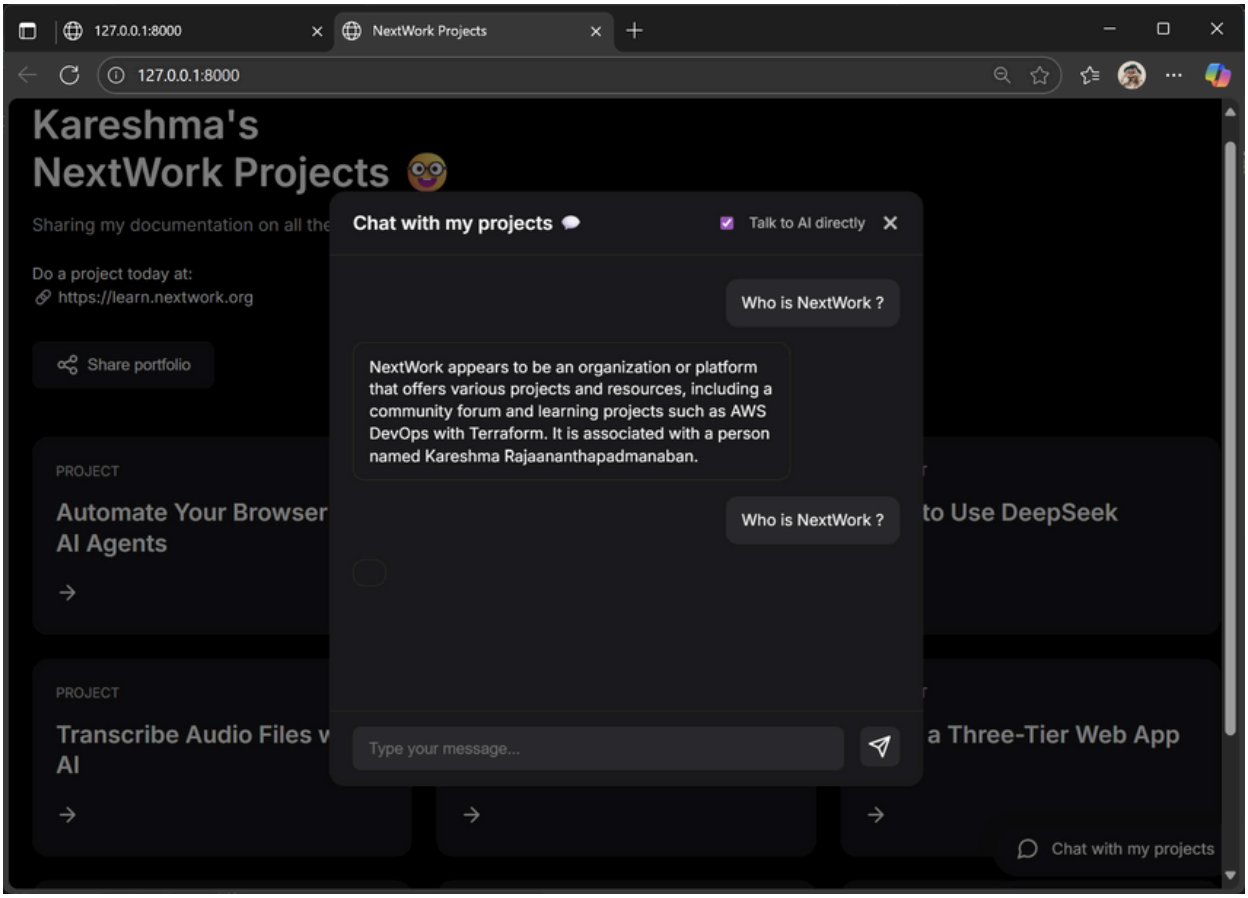
Web app running on browser



Web app chat with projects RAG Enabled

```
(venv) PS D:\Projects\Kareshma\Nxtwrkcode\nextwork-rag-webapp> python -m uvicorn web_app
:app --reload
INFO:     Will watch for changes in these directories: ['D:\\Projects\\Kareshma\\Nxtwrkc
ode\\nextwork-rag-webapp']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [16392] using StatReload
INFO:botocore.credentials:Found credentials in shared credentials file: ~/.aws/credentia
ls
INFO:     Started server process [1460]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     127.0.0.1:61717 - "GET / HTTP/1.1" 200 OK
INFO:     127.0.0.1:61717 - "GET /static/style.css HTTP/1.1" 200 OK
INFO:     127.0.0.1:61718 - "GET / HTTP/1.1" 200 OK
INFO:     127.0.0.1:58472 - "GET /bedrock/query?text=Who%20is%20NextWork%20%3F HTTP/1.1"
 200 OK
INFO:     127.0.0.1:58477 - "GET /bedrock/invoke?text=Who%20is%20NextWork%20%3F HTTP/1.1
" 500 Internal Server Error
```

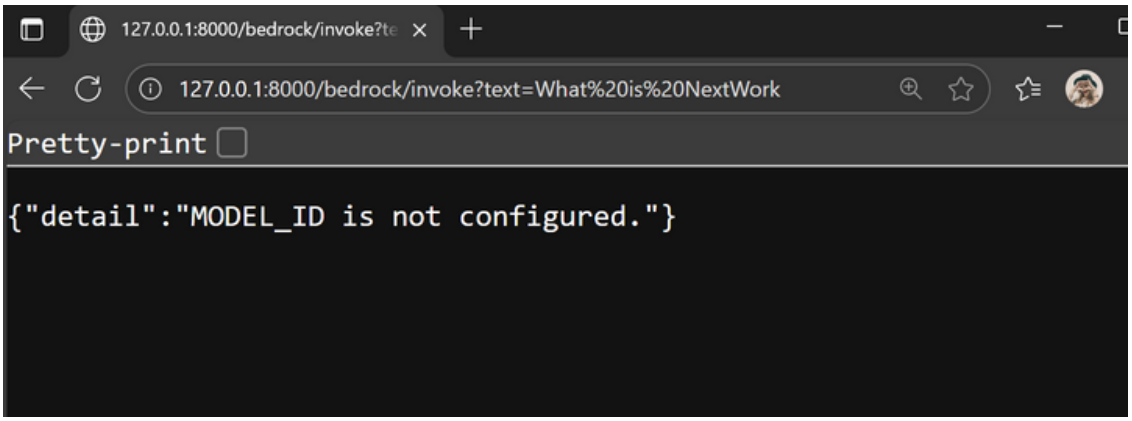
500 Internal server error when check Talk to AI directly



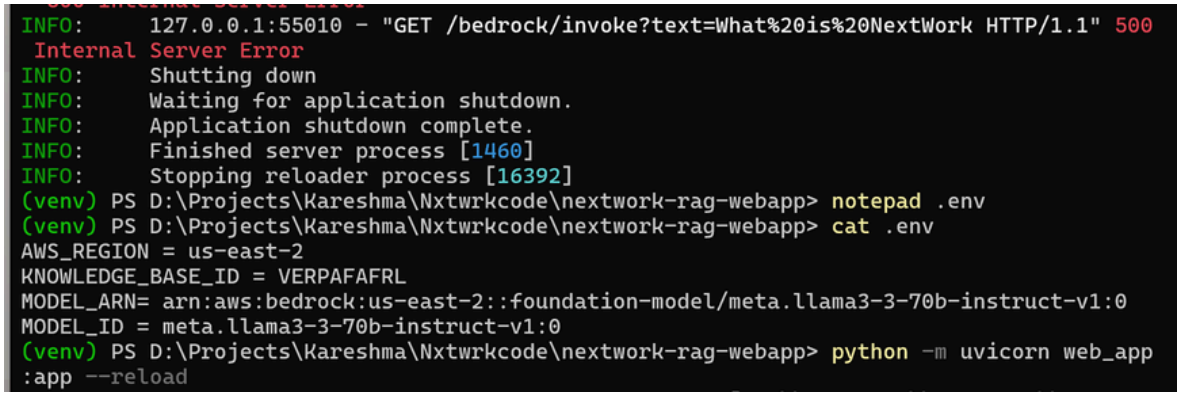
No resonse from chatbot when Checked Talk to AI directly

The terminal says 500 Internal Server Error. Looks like there's an error in the API...

The **Talk to AI directly** checkbox controls whether the chatbot uses Retrieval-Augmented Generation (RAG) or not. When **checked**, the chatbot will **bypass the Knowledge Base** and talk directly to the AI model (Llama 3.3 70B Instruct). In this mode, the chatbot will answer questions based on its general knowledge, without using your specific documents.



Model ID error

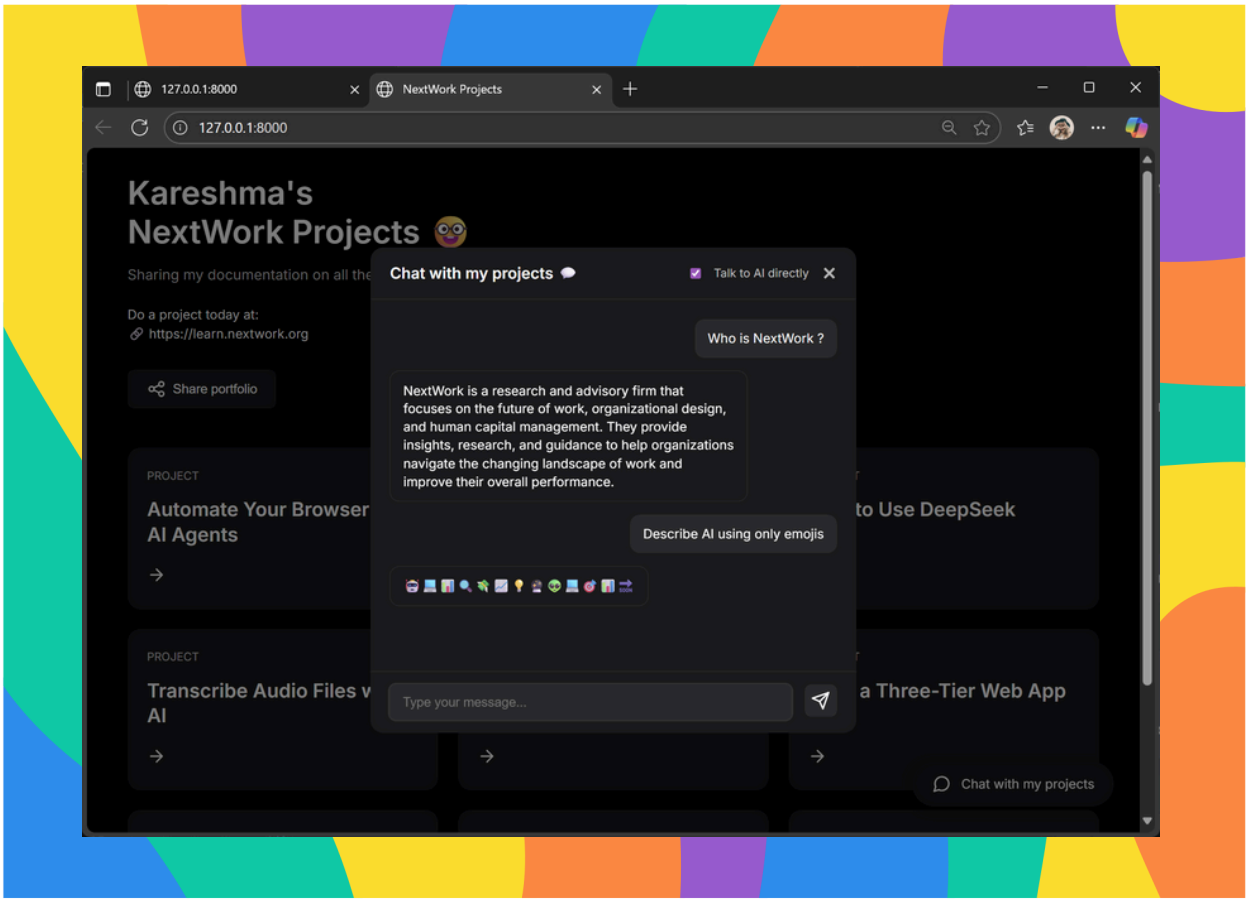


Adding Model ID to .env file

Error {"detail":"Model ID is missing"}

This is beacause **MODEL_ID haven’t configure** in environment variable.

MODEL_ID is the **ID of the AI model** to call it directly. It's a simpler ID that just names the model, without any region or account information. That's why it's called an ID, not a Model ARN!



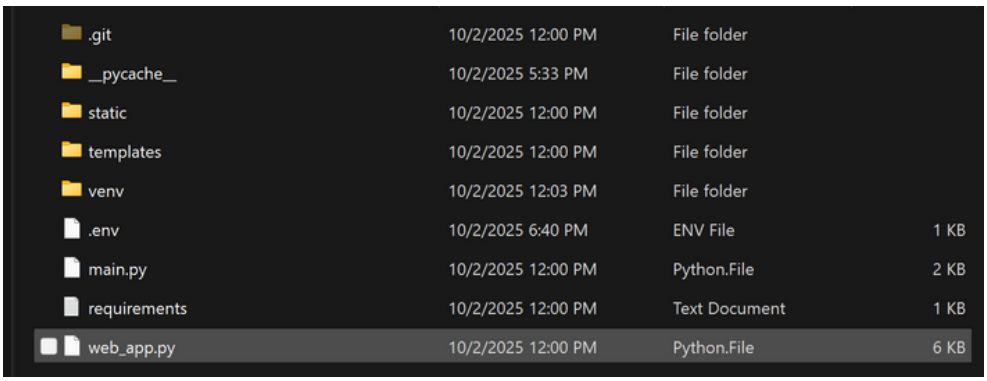
Web app showing response AI directly

Breaking down the Web App

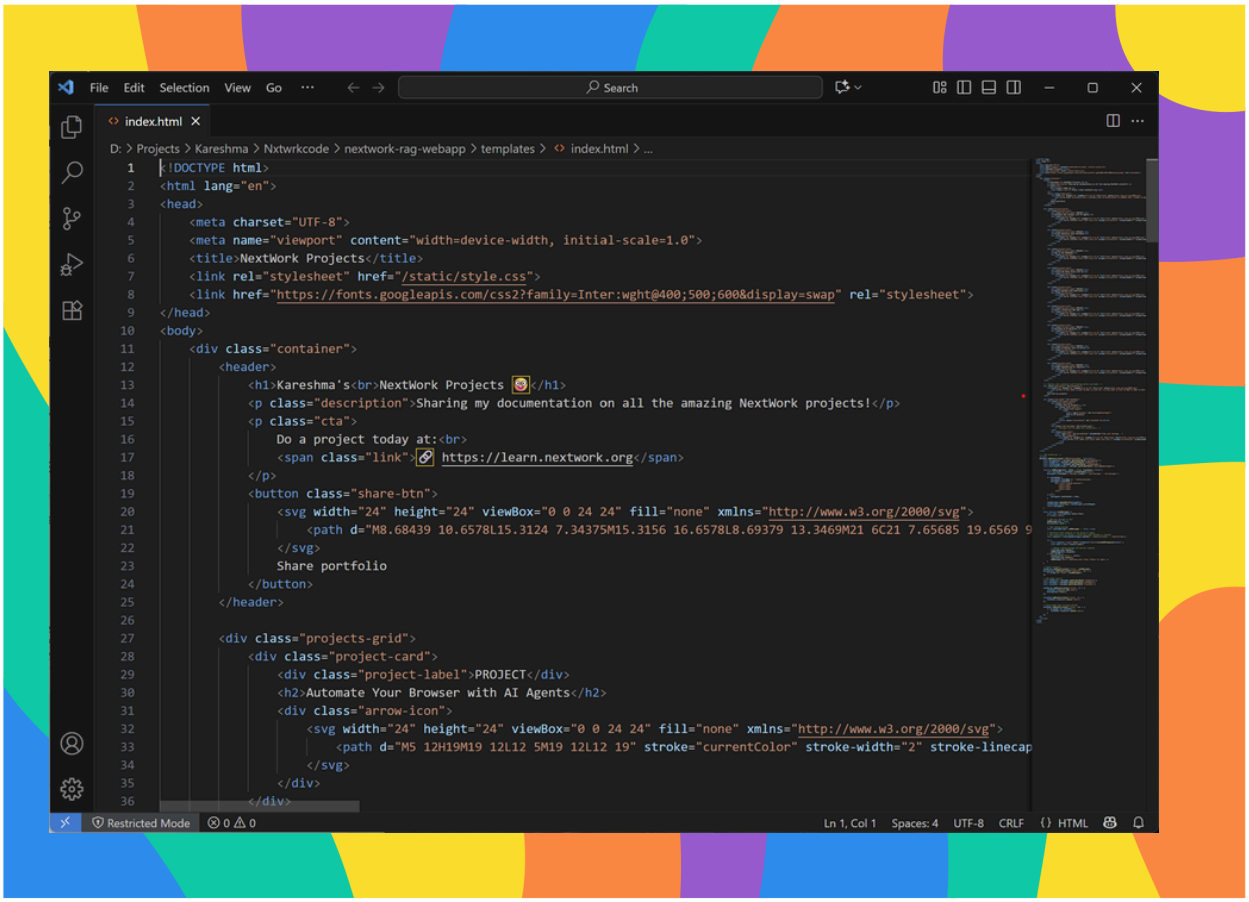
In this secret mission, I'm going to investigate the full web app's code, understand how its frontend, backend, and data layers work together, and explore how chat interactions are handled. I'll also compare **running** the **API alone (main.py)** versus **running** the **complete web app (web_app.py)** to see how everything integrates into a functional chatbot interface.

When a user sends a chat message, the frontend JavaScript captures the text and sends it as a **GET request** to the /bedrock/query endpoint. The backend (web_app.py) receives the request, queries the Knowledge Base via Bedrock, and generates a response. That response is sent back to the frontend, where the JavaScript displays both the **user's message** and the **chatbot's reply** in the **chat window.**

web_app.py extends the API by adding a **full web interface** on top of the core backend in main.py. It serves **HTML pages, CSS,** and **JavaScript** via **FastAPI,** uses **Jinja2 templates** to dynamically render pages, configures logging for debugging, and introduces a new endpoint (/bedrock/invoke) that lets the **AI model respond directly**, bypassing the Knowledge Base.



All files to run a full web app for RAG chatbot



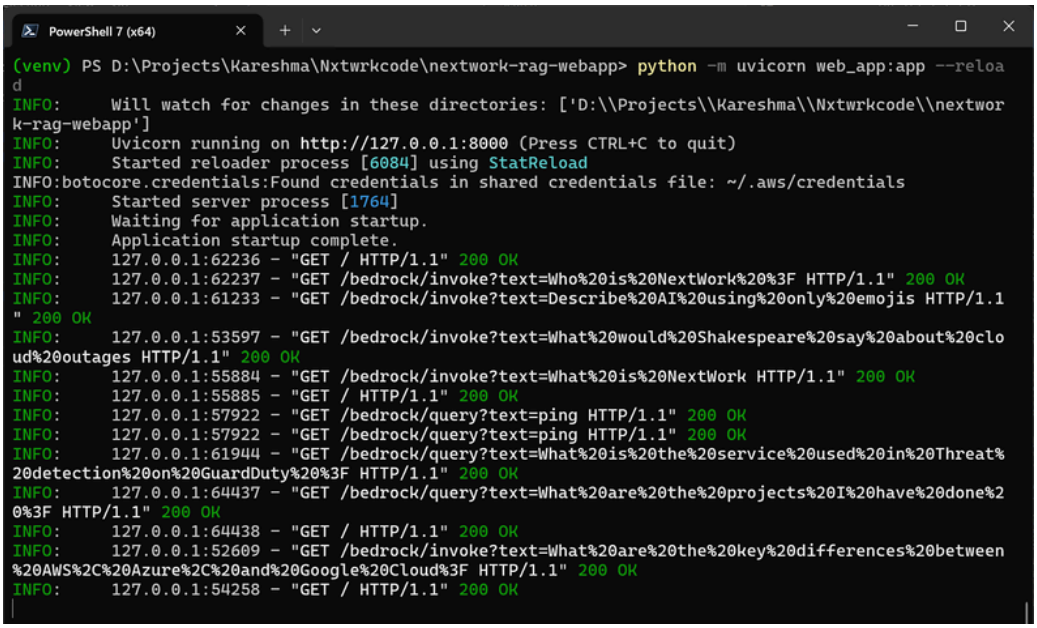
Index.html file of Web app

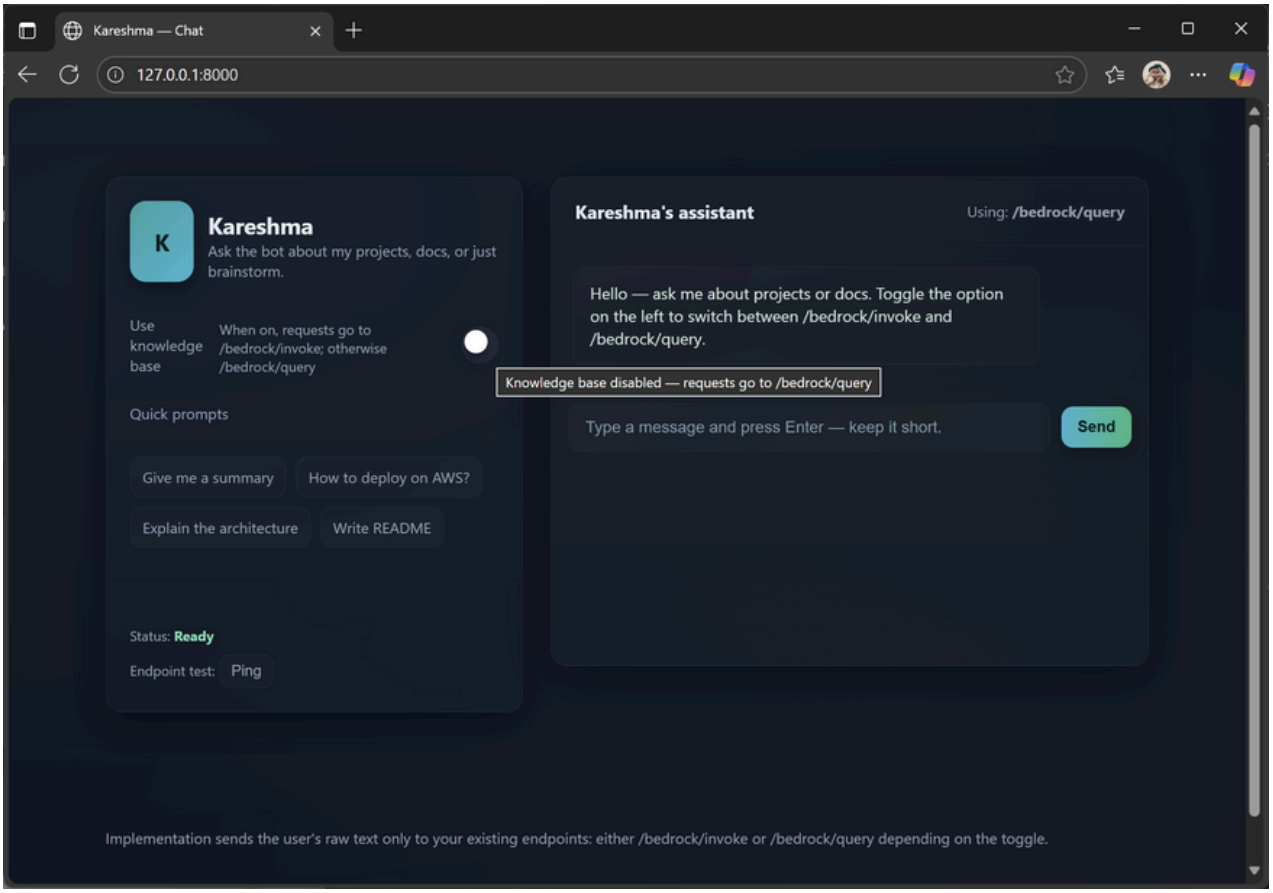
Customizing the Frontend

In this secret mission, I'm going to give my chatbot web app a fresh look by **customizing the frontend.** The goal is to make the interface feel more personal and engaging, instead of just sticking with the default design. I'll either generate a **brand-new HTML** file or tweak the existing index.html, then integrate it into my project and test the updated design in action.

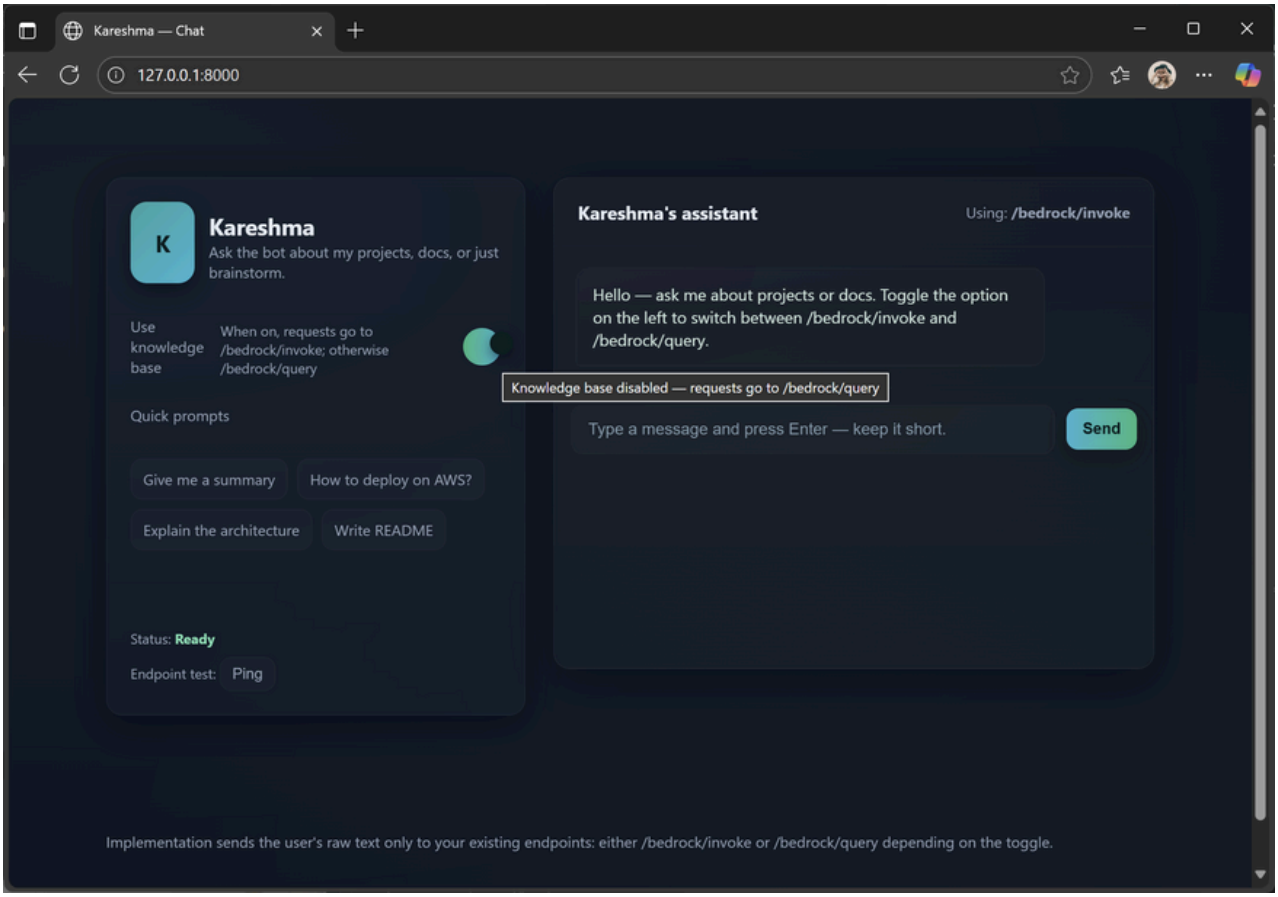
I want to experiment with customizing the frontend because it gives me the freedom to design a chatbot interface that feels more engaging and unique, instead of being stuck with a default look. This is possible because as long as the **API endpoints stay the same,** the **backend doesn't care** how the **frontend looks** it just sends and receives data. That separation lets me play with design while keeping the functionality intact.

My customized interface now features a clean, modern dark theme with two distinct panels. On the left, there's a profile-style card showing my name, a **toggle** to switch between /bedrock/invoke and /bedrock/query, quick prompt buttons, and a status section. On the right, the chat assistant panel feels more polished, with a neatly styled conversation area, an input box, and a bright send button. It looks like a professional dashboard rather than a plain chat box.

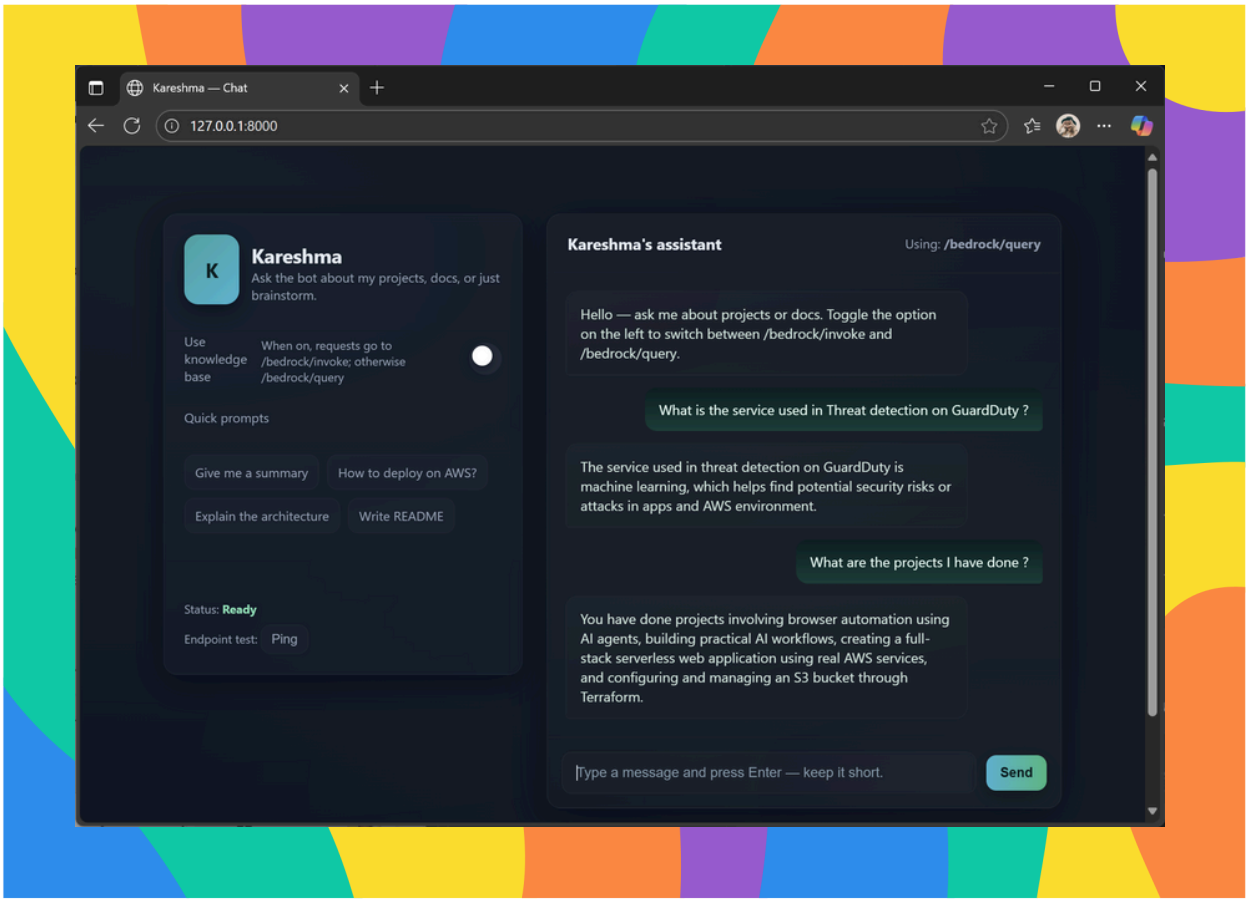




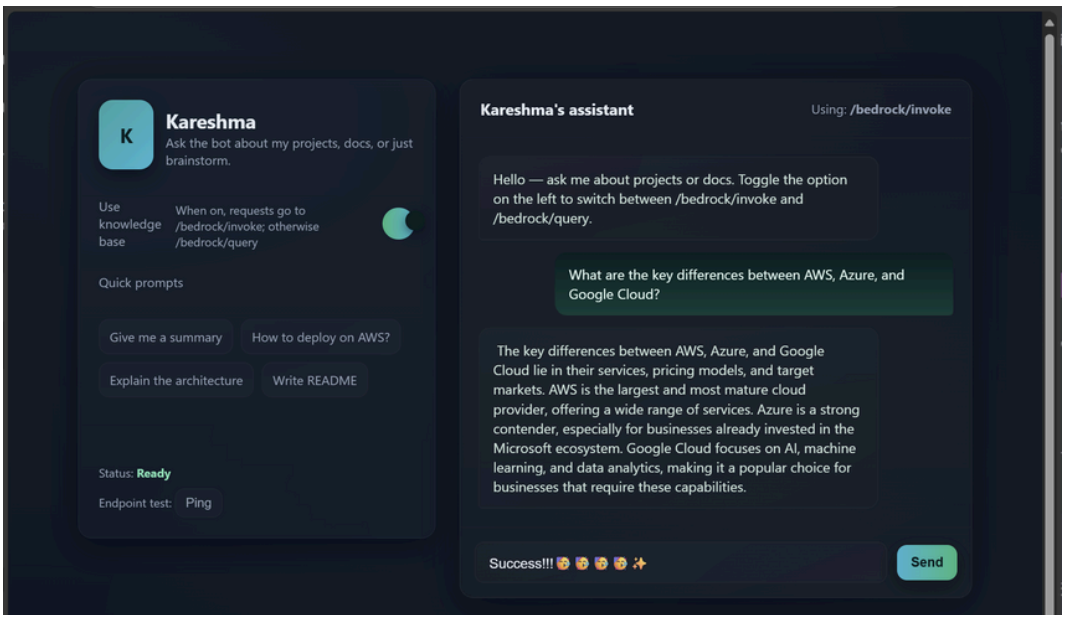
Customize Web app Query option - Enabled RAG



Customize Web app Invoke - Talk to AI directly



Query response



Invoke response

Delete the Resources

Now that we've built and tested our chatbot, it's time to clean up the AWS resources we created. Cleaning up ensures you don't get charged for unused resources and keeps your account tidy.

Resources to delete:

- Bedrock Knowledge Base
- OpenSearch Serverless Collection (vector store)
- IAM Access Key
- S3 Bucket
- Virtual Environment and cloned repository

1) Delete the Knowledge Base

1. Open the Bedrock Console.
2. From the left menu, choose Knowledge Bases.
3. Select your Knowledge Base.
4. Choose Delete.
5. Type delete in the confirmation field and confirm.

2) Delete the Vector Store (OpenSearch)

💡 What's a vector store?

A vector store organizes data by meaning, not just keywords. For example, searching "quick breakfast ideas" might also match "5-minute morning meals.

Delete the Resources

- Open the OpenSearch Console.
- From the left menu, choose Collections.
- Select your vector store.
- Choose Delete.
- Type confirm in the confirmation field and confirm.

3**) Delete IAM Access Key**

1. Open the IAM Console.
2. From the left menu, select Users.
3. Select your IAM user → Security credentials tab.
4. In Access keys, find your key.
5. Select Actions → Delete.
6. Deactivate it, then type the access key ID to confirm deletion.

💡 Why delete access keys?

They provide programmatic access to your AWS account. Removing them prevents unauthorized access.

Delete the Resources

4) Delete the S3 Bucket

1. Open the S3 Console.
2. Select your bucket (e.g., bedrock-kb-data-<your_initials>).
3. Choose Empty → type permanently delete → confirm.
4. Once empty, select the bucket → Delete.
5. Type the bucket name (e.g., nextwork-rag-documentation-<your_initials>) to confirm.

5) Deactivate Virtual Environment

1. Go back to your terminal.
2. Stop the server with Ctrl+C.
3. Deactivate your environment:

CMD → deactivate

You should see the (venv) prompt disappear.

💡 Why deactivate?

It ensures clean separation for future projects and avoids accidental dependency usage.

Kareshma
Rajaananthapadmanaban

Follow up for more interesting projects !!!

Check out nextwork.org / my profile for more projects