🚀 Day 6: Publish Your Agent

Welcome to day six of our AI Agents Crash Course.

We have done a lot of things so far:

- Created a pipeline for processing any GitHub repository
- Ingested this data into a search engine
- Created an agent that uses the search engine as a tool
- Evaluated this agent

But this agent still lives inside our Jupyter Notebook. It's time to make it available for everyone

That's what we will do today:

- Clean the code
- Create a UI for it with Streamlit
- Deploy it to the Internet

At the end of this lesson, you'll be able to share the link to your agent on social media, so anyone can interact with it.

Cleaning up

Maybe while following the course you kept your code organized, but in my case, I have everything in one Jupyter Notebook. By now it's quite messy.

So the first step is to organize everything into multiple clean Python files. This makes our code easier to maintain, test, and deploy.

First, let's create a separate folder called "app".

Inside I initialized an empty uv project and updated dependencies in `pyproject.toml`:

```
None
dependencies = [
    "minsearch>=0.0.5",
    "openai>=1.108.2",
```

```
    "pydantic-ai==1.0.9",
    "python-frontmatter>=1.1.0",
    "requests>=2.32.5",
]
```

I didn't include sentence transformers but feel free to add it if you plan to use vector search.

Now install the dependencies:

```
None
uv sync
```

I created the following files:

- `ingest.py` - handles data loading and indexing from GitHub repositories
- `search_tools.py` - contains our search tool implementation
- `search_agent.py` - creates and configures the Pydantic AI agent
- `logs.py` - handles logging of conversations
- `main.py` - brings everything together for the command-line interface

Let me include the code for each here.

ingest.py

```python
import io
import zipfile
import requests
import frontmatter

from minsearch import Index


def read_repo_data(repo_owner, repo_name):
```

```
    url =
f'https://codeload.github.com/{repo_owner}/{repo_name}/zip/refs/heads/main'
    resp = requests.get(url)

    repository_data = []

    zf = zipfile.ZipFile(io.BytesIO(resp.content))

    for file_info in zf.infolist():
        filename = file_info.filename.lower()

        if not (filename.endswith('.md') or filename.endswith('.mdx')):
            continue

        with zf.open(file_info) as f_in:
            content = f_in.read()
            post = frontmatter.loads(content)
            data = post.to_dict()

            _, filename_repo = file_info.filename.split('/', maxsplit=1)
            data['filename'] = filename_repo
            repository_data.append(data)

    zf.close()

    return repository_data


def sliding_window(seq, size, step):
    if size <= 0 or step <= 0:
        raise ValueError("size and step must be positive")

    n = len(seq)
    result = []
    for i in range(0, n, step):
        batch = seq[i:i+size]
        result.append({'start': i, 'content': batch})
        if i + size > n:
            break

    return result


def chunk_documents(docs, size=2000, step=1000):
```

```
    chunks = []

    for doc in docs:
        doc_copy = doc.copy()
        doc_content = doc_copy.pop('content')
        doc_chunks = sliding_window(doc_content, size=size, step=step)
        for chunk in doc_chunks:
            chunk.update(doc_copy)
        chunks.extend(doc_chunks)

    return chunks


def index_data(
        repo_owner,
        repo_name,
        filter=None,
        chunk=False,
        chunking_params=None,
    ):
    docs = read_repo_data(repo_owner, repo_name)

    if filter is not None:
        docs = [doc for doc in docs if filter(doc)]

    if chunk:
        if chunking_params is None:
            chunking_params = {'size': 2000, 'step': 1000}
        docs = chunk_documents(docs, **chunking_params)

    index = Index(
        text_fields=["content", "filename"],
    )

    index.fit(docs)
    return index
```

I made a few improvements here. The line `_, filename_repo = file_info.filename.split('/', maxsplit=1)` in `read_repo_data` strips the first part of the path (the zip archive name), making it easier for our agent to create references.

I also replaced `chunk` with `content` in `sliding_window`, so the content is always in the `content` field, and our code works with or without chunking.

Finally, we have the `index_data` function that combines all the ingestion steps. This covers what we did in days 1-3.

Days 4 and 5 are in `search_agent.py` and `search_tools.py`.

search_tools.py:

```python
from typing import List, Any

class SearchTool:
    def __init__(self, index):
        self.index = index

    def search(self, query: str) -> List[Any]:
        """
        Perform a text-based search on the FAQ index.

        Args:
            query (str): The search query string.

        Returns:
            List[Any]: A list of up to 5 search results returned by the FAQ
index.
        """
        return self.index.search(query, num_results=5)
```

I created a class instead of just a function like we had in the Jupyter notebook. Previously, it was a global variable that we referenced from a function. Now the `index` is encapsulated inside a class with tools, which makes the code more organized.

search_agent.py

```python
import search_tools
from pydantic_ai import Agent


SYSTEM_PROMPT_TEMPLATE = """
You are a helpful assistant that answers questions about documentation.

Use the search tool to find relevant information from the course materials
before answering questions.

If you can find specific information through search, use it to provide accurate
answers.

Always include references by citing the filename of the source material you
used.
Replace it with the full path to the GitHub repository:
"https://github.com/{repo_owner}/{repo_name}/blob/main/"
Format: [LINK TITLE](FULL_GITHUB_LINK)


If the search doesn't return relevant results, let the user know and provide
general guidance.
"""

def init_agent(index, repo_owner, repo_name):
    system_prompt = SYSTEM_PROMPT_TEMPLATE.format(repo_owner=repo_owner,
repo_name=repo_name)

    search_tool = search_tools.SearchTool(index=index)

    agent = Agent(
        name="gh_agent",
        instructions=system_prompt,
        tools=[search_tool.search],
        model='gpt-4o-mini'
    )

    return agent
```

Here we use a template instead of hardcoding the repository information, so it's more flexible and can work with any code repository.

Then we have another change: `tools=[search_tool.search]` instead of the previous `tools=[text_search]` in `Agent` because the tool we want to use is now a method of the `search_tool` class.

All the materials from day 5 (yesterday) are in **logs.py:**

```python
import os
import json
import secrets
from pathlib import Path
from datetime import datetime

from pydantic_ai.messages import ModelMessagesTypeAdapter


LOG_DIR = Path(os.getenv('LOGS_DIRECTORY', 'logs'))
LOG_DIR.mkdir(exist_ok=True)


def log_entry(agent, messages, source="user"):
    tools = []

    for ts in agent.toolsets:
        tools.extend(ts.tools.keys())

    dict_messages = ModelMessagesTypeAdapter.dump_python(messages)

    return {
        "agent_name": agent.name,
        "system_prompt": agent._instructions,
        "provider": agent.model.system,
        "model": agent.model.model_name,
        "tools": tools,
        "messages": dict_messages,
        "source": source
    }


def serializer(obj):
    if isinstance(obj, datetime):
        return obj.isoformat()
    raise TypeError(f"Type {type(obj)} not serializable")
```

```
def log_interaction_to_file(agent, messages, source='user'):
    entry = log_entry(agent, messages, source)

    ts = entry['messages'][-1]['timestamp']
    ts_str = ts.strftime("%Y%m%d_%H%M%S")
    rand_hex = secrets.token_hex(3)

    filename = f"{agent.name}_{ts_str}_{rand_hex}.json"
    filepath = LOG_DIR / filename

    with filepath.open("w", encoding="utf-8") as f_out:
        json.dump(entry, f_out, indent=2, default=serializer)

    return filepath
```

The main change here is `LOG_DIR = Path(os.getenv('LOGS_DIRECTORY', 'logs'))`. This allows us to configure the log directory using an environment variable. It's useful for deployment and makes our code more flexible.

Finally, **main.py** puts everything together:

Python

```
import ingest
import search_agent
import logs

import asyncio


REPO_OWNER = "DataTalksClub"
REPO_NAME = "faq"


def initialize_index():
    print(f"Starting AI FAQ Assistant for {REPO_OWNER}/{REPO_NAME}")
    print("Initializing data ingestion...")

    def filter(doc):
        return 'data-engineering' in doc['filename']
```

```
    index = ingest.index_data(REPO_OWNER, REPO_NAME, filter=filter)
    print("Data indexing completed successfully!")
    return index


def initialize_agent(index):
    print("Initializing search agent...")
    agent = search_agent.init_agent(index, REPO_OWNER, REPO_NAME)
    print("Agent initialized successfully!")
    return agent


def main():
    index = initialize_index()
    agent = initialize_agent(index)
    print("\nReady to answer your questions!")
    print("Type 'stop' to exit the program.\n")

    while True:
        question = input("Your question: ")
        if question.strip().lower() == 'stop':
            print("Goodbye!")
            break

        print("Processing your question...")
        response = asyncio.run(agent.run(user_prompt=question))
        logs.log_interaction_to_file(agent, response.new_messages())

        print("\nResponse:\n", response.output)
        print("\n" + "="*50 + "\n")


if __name__ == "__main__":
    main()
```

This creates a simple command-line interface for our agent. We use `asyncio.run()` because Pydantic AI's `run` method is asynchronous.

Of course, this code isn't perfect.

We don't have documentation or tests. If you're using Cursor or GitHub Copilot, you can easily add these. I'd also create a CI/CD pipeline that runs tests every time you push

code to GitHub. Finally, the logs should be ideally saved to a proper storage service like S3.

But we won't do that here. Let's focus on getting our agent online.

Streamlit

The code is clean and modular, so we can now use it to create a UI. We'll use Streamlit.

Streamlit is a Python library that makes it easy to create web applications. You write Python code, and it automatically creates an interactive web interface.

Let's install it:

```shell
uv add streamlit
```

I don't know much about Streamlit, and can only create simple interfaces. But luckily, we have many AI coding assistants that can help us.

I'll use ChatGPT today and share the results with you. But you can experiment with your favorite tool too. If you don't have one yet, try ChatGPT. Starting from GPT-4, ChatGPT is quite good at coding tasks. You can also try GitHub Copilot or Cursor.

Pydantic AI has a nice gallery with examples. Among other things, I saw Chat App with FastAPI. We can use it as inspiration with minimal changes.

I'll use it for creating a Streamlit app.

My prompt for ChatGPT:

```
I have this agent I created with Pydantic AI

[insert main.py]

I want to turn it into streamlit code. Base it on the following code for
creating web apps with Pydantic AI:

[insert example from docs]
```

You can see my conversation here.

I got some code and put it into `app.py`:

```python
import streamlit as st
import asyncio

import ingest
import search_agent
import logs


# --- Initialization ---
@st.cache_resource
def init_agent():
    repo_owner = "DataTalksClub"
    repo_name = "faq"

    def filter(doc):
        return 'data-engineering' in doc['filename']

    st.write("🔄 Indexing repo...")
    index = ingest.index_data(repo_owner, repo_name, filter=filter)
    agent = search_agent.init_agent(index, repo_owner, repo_name)
    return agent


agent = init_agent()

# --- Streamlit UI ---
st.set_page_config(page_title="AI FAQ Assistant", page_icon="🤖",
layout="centered")
st.title("🤖 AI FAQ Assistant")
st.caption("Ask me anything about the DataTalksClub/faq repository")

# Initialize chat history
if "messages" not in st.session_state:
    st.session_state.messages = []

# Display chat history
for msg in st.session_state.messages:
    with st.chat_message(msg["role"]):
        st.markdown(msg["content"])
```

```
# Chat input
if prompt := st.chat_input("Ask your question..."):
    # User message
    st.session_state.messages.append({"role": "user", "content": prompt})
    with st.chat_message("user"):
        st.markdown(prompt)

    # Assistant response
    with st.chat_message("assistant"):
        with st.spinner("Thinking..."):
            response = asyncio.run(agent.run(user_prompt=prompt))
            answer = response.output
            st.markdown(answer)

    # Save response to history + logs
    st.session_state.messages.append({"role": "assistant", "content": answer})
    logs.log_interaction_to_file(agent, response.new_messages())
```

Let's run it:

```shell
uv run streamlit run app.py
```

This is what I got:

It worked on the first attempt!

Note that we don't keep conversation history between questions. When you ask the next question, the agent doesn't remember previous exchanges.

We can fix this by passing `message_history` in the `run` method of the agent. You can see how I implemented this in ToyAIKit's `PydanticAIRunner` class.

I played with it and thought: "It would be nice if responses streamed instead of displaying everything at once". Pydantic AI has an example with streaming, so I asked ChatGPT to adjust our `app.py`.

My prompt:

```
None
Can you make it streaming? Here's an example from PydanticAI:

[example]
```

After a few back-and-forth exchanges, I got this code:

```
Python
import streamlit as st
import asyncio

import ingest
```

```python
import search_agent
import logs


# --- Initialization ---
@st.cache_resource
def init_agent():
    repo_owner = "DataTalksClub"
    repo_name = "faq"

    def filter(doc):
        return "data-engineering" in doc["filename"]

    st.write("🔄 Indexing repo...")
    index = ingest.index_data(repo_owner, repo_name, filter=filter)
    agent = search_agent.init_agent(index, repo_owner, repo_name)
    return agent


agent = init_agent()

# --- Streamlit UI ---
st.set_page_config(page_title="AI FAQ Assistant", page_icon="🤖",
layout="centered")
st.title("🤖 AI FAQ Assistant")
st.caption("Ask me anything about the DataTalksClub/faq repository")

# Initialize chat history
if "messages" not in st.session_state:
    st.session_state.messages = []

# Display chat history
for msg in st.session_state.messages:
    with st.chat_message(msg["role"]):
        st.markdown(msg["content"])


# --- Streaming helper ---
def stream_response(prompt: str):
    async def agen():
        async with agent.run_stream(user_prompt=prompt) as result:
            last_len = 0
            full_text = ""
            async for chunk in result.stream_output(debounce_by=0.01):
```

```
                # stream only the delta
                new_text = chunk[last_len:]
                last_len = len(chunk)
                full_text = chunk
                if new_text:
                    yield new_text
            # log once complete
            logs.log_interaction_to_file(agent, result.new_messages())
            st.session_state._last_response = full_text

    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    agen_obj = agen()

    try:
        while True:
            piece = loop.run_until_complete(agen_obj.__anext__())
            yield piece
    except StopAsyncIteration:
        return


# --- Chat input ---
if prompt := st.chat_input("Ask your question..."):
    # User message
    st.session_state.messages.append({"role": "user", "content": prompt})
    with st.chat_message("user"):
        st.markdown(prompt)

    # Assistant message (streamed)
    with st.chat_message("assistant"):
        response_text = st.write_stream(stream_response(prompt))

    # Save full response to history
    final_text = getattr(st.session_state, "_last_response", response_text)
    st.session_state.messages.append({"role": "assistant", "content":
final_text})
```

I don't want to understand all the details of this code (if I wanted to, I'd ask ChatGPT to explain it). But hey - it works!

You could even ask ChatGPT to display tool calls and other debugging information. Feel free to experiment with that.

But for now, let's deploy what we already have.

Deployment

Streamlit Cloud should understand uv, but it didn't work for me. So I exported the dependencies into `requirements.txt`:

```shell
uv export --no-dev > requirements.txt
```

Push all your code to GitHub - that's how Streamlit Cloud will access it.

By now you probably noticed the "Deploy" button at the top right corner. Let's press it.

Congratulations! Your application is deployed.

It won't work immediately though, because we need to provide it with an OpenAI API key. Go to https://share.streamlit.io/, find your app there, and add your key in the secrets:

```
OPENAI_API_KEY="your-key"
```

If you want to be more careful with your API keys, you can create a project in OpenAI (e.g., "streamlit") and use a project-specific key. This way you can easily monitor your spending for this particular project.

That's it! Our app is deployed and working.

Here's my app: https://aiherodefaq.streamlit.app/

By the time you read this, it'll probably be disabled. Every time somebody uses it, I have to pay OpenAI. And I don't want that.

But tomorrow we'll wrap everything up and record a video about it for publishing on social media!

Homework

- Clean up your agent code into modular Python files
- Create a Streamlit interface for your agent
- Deploy your app to Streamlit Cloud and share it (optionally)
- Share your progress on social media

Example post for LinkedIn

🚀 Day 6 of building AI agent: publishing the agent!

Today I took my agent from Jupyter notebook to the Internet:

• Cleaned up messy code into modular Python files
• Built a web interface with Streamlit
• Deployed to the cloud with one click
• Now anyone can interact with my agent!

My project helps users search through [YOUR PROJECT DESCRIPTION].
The agent can answer questions and provide relevant links from the documentation.

Here's my live app: [YOUR_APP_LINK]
My repo: [YOUR_REPO_LINK]

Tomorrow: final touches and course wrap-up!

Following along with this amazing course - who else is building AI agents?
You can sign up here: https://alexeygrigorev.com/aihero/

Example post for Twitter/X

Day 6: My AI agent is LIVE! 🚀

✅ Cleaned up messy notebook code
✅ Built Streamlit interface
✅ Deployed to cloud in minutes
✅ Anyone can now use my agent

From local notebook → production web app

Here's my live app: [YOUR_APP_LINK]
Here's my repo: [YOUR_REPO_LINK]

Next: Course wrap-up & final demo

Join me: https://alexeygrigorev.com/aihero/

Community

Have questions about this lesson or suggestions for improvement? You can find me and other learners in **DataTalks.Club Slack**:

- Join DataTalks.Club
- Find us in the `#course-ai-bootcamp` channel

In the community channel, you can:

- Ask questions about the course content
- Share your implementation and get feedback
- Show off your deployed applications
- Get help with deployment issues
- Connect with other course participants

Don't hesitate to reach out - the community is here to help each other succeed!