Pandelis Margaronis
Hamilton College
Class of 2025
pmargaro@hamilton.edu

Docu-Dive: A Look Inside the Development Process

Brainstorming Ideas

- At first, I thought about my passions in sports, soccer, health, and fitness
- I knew I wanted to build something that leverages Generative AI and Large Language Models (LLMs), but I wasn't sure what I wanted to work with
- I thought about a health and nutrition app that utilizes existing nutrition APIs, but the costs were prohibitive, so I continued exploring other options
- I decided it was best to address a problem I had experienced directly and spoke with friends. I spoke mainly with people that did not have a technical background. I wanted to solve a real problem, and thought that this would align closer to an experience I'd have in the industry, where I need to not just build a tool, but also a tool that can practically help users… that users will want to use over something else
- Attending a liberal arts school, I witnessed my friends struggle with the tedious task of digging up evidence from extensive readings for essays. This frustration, combined with my internship plans falling through due to budget cuts, motivated me to create a solution.
- GenAI LLMs often hallucinate results and provide vague citations, making them unreliable for finding specific quotes or evidence within large documents. Even when somewhat accurate results are provided, they are not grounded with citations. Users are hesitant to trust an LLM in these scenarios.
- My goal was to address these issues by creating a tool that offers precise, chunk-level citations, ensuring accuracy and reliability.
- So, I decided on a Document Searching application

First Approach: Create a local tool (Backend)

- Watched many videos about Retrieval Augmented Generation (RAG) from IBM, freeCodeCamp, and various individual contributors
- I read a bunch of articles and looked into subtopics such as vector databases, vector embeddings, and generally the math that powers RAG and takes place behind the scenes
- I researched free options to achieve RAG, using local and free tools that were open source. This is how I came across ChromaDB and Ollama - although Ollama 3.1 was not yet available, the models I used sufficed.
- I drew inspiration from online resources using ChromaDB and Ollama Mistral 7B and was able to get a tool working locally.
- Langchain released new documentation, so some of the code I was drawing inspiration from was outdated. I gained valuable experience reading through the updated Langchain documentation.

- Another obstacle I had was some AWS configurations to use AWS Bedrock for embeddings. Amazon Bedrock embeddings are numerical representations that capture the semantic meaning of text data. Embeddings are used in various natural language processing (NLP) tasks, such as document search. I was having some issues with utilizing this resource, but was able to read documentation and use error-catching to debug the issue. It was good to run into an AWS-related issue early on and get practice debugging it. I got some practice creating and attaching custom policies and navigating the AWS Bedrock management console.

Building the frontend:

- Then, once the backend tool was working locally, I decided to build a frontend. I had a little experience with React previously, so this was a great learning opportunity.
- Once again, I watched a lot of videos on React and also read a lot of documentation. I kept my design fairly simple, as I wanted the UX to be seamless for someone who does not have a technical background.
- I got feedback from my friends for this, and sent them pictures of the UI asking them to send me feedback and suggestions.

Front End Obstacles / Resolutions:

Button Alignment:

- Issue: Aligning the "Submit" and "Clear" buttons within the query input container.
- Resolution: Adjusted CSS styles and the HTML structure to ensure the buttons were aligned properly, using flexbox for layout.

Moving the "Clear" Button:

- Issue: Placing the "Clear" button at the bottom of the response section.
- Resolution: Modified the component structure and CSS to position the "Clear" button correctly within the response section.

Centering Elements:

- Issue: Centering the "Browse Files" button within the drop-box div.
- Resolution: Adjusted the CSS to move the "Browse Files" button further down and center it within the div.

Displaying Query Results:

- Issue: Formatting the query results, especially handling line breaks and displaying sources properly.
- Resolution: Adjusted the response format and used CSS to ensure proper display and spacing for the query results and sources.

Bullet Points in Response:

- Issue: An extra bullet point appearing in the sources list.
- Resolution: Ensured proper list handling in the JSX to avoid adding an extra bullet point.

    Implementing Toggle for Sources:
    - Issue: Displaying sources only when the user clicks a "See Sources" button.
    - Resolution: Added a button to toggle the visibility of the sources and managed the state to show or hide the sources as needed.

    Styling the Application:
    - Issue: Ensuring the overall look and feel of the application were visually appealing and user-friendly.
    - Resolution: Made several adjustments to the CSS, including colors, margins, padding, and font sizes, to achieve the desired appearance.

Returning to the Back-End:

- At this point, I wanted to incorporate AWS S3 into the process. There was not a particular need for this, and it did not affect the user experience in any way. However, I've noticed that many job postings list AWS S3 experience in their preferred qualifications. Further, many software engineers working in the industry right now, who I've been fortunate to speak to and hear about their experiences, have mentioned that they work with S3 on a daily basis. I wanted to make sure I can upload files and delete files from an S3 bucket within my code, hence why I implemented this feature "behind the scenes".

Integrating the Frontend with the Backend:

- I faced several issues connecting your frontend to your backend. Here are a few notable ones:

1. AWS IAM Configuration:
    - Issue: Difficulty enabling access and configuring models within the AWS management console, particularly with custom policies and specific model access.
    - Resolution: Thoroughly reviewed AWS IAM and Bedrock documentation and tested configurations incrementally to ensure correct access and functionality.
2. Handling REST API Requests:
    - Issue: Ensuring that the API requests from the frontend (React) to the backend (FastAPI) were correctly formatted and handled.
    - Resolution: Debugged and tested the POST and GET requests, ensuring the correct content type and payload were sent.
3. File Upload and Retrieval:
    - Issue: Managing the file upload process and storing files in AWS S3, followed by retrieving and querying these files.
    - Resolution: Implemented proper handling of form data and ensured the S3 bucket was correctly configured. Added detailed error handling and logging.
4. API Endpoint Creation / Adjustment:
    - Issue: Integrating the React frontend with the FastAPI backend, ensuring smooth communication between the two.

    - Resolution: Ensured the API endpoints were correctly defined and accessible, and the frontend was making requests to the correct URLs.
5. Displaying Query Results:
    - Issue: Formatting and displaying the results from the backend query on the frontend, including handling the sources.
    - Resolution: Used React state management to handle and display the response, and formatted the sources properly.
6. Handling Clear-Button Functionality:
    - Issue: Implementing a clear button that resets the uploaded files and query history.
    - Resolution: Added a clear button in the frontend that sends a request to the backend to clear the files and resets the state in React.

Transitioning Towards Deployment:

- At this point, I looked towards deploying my application. I had some previous experience with AWS, as I had used EC2 and RDS to host a PostgreSQL database. I was familiar with IAM role and policy creation / management, as well as the configuration and consistent monitoring of an EC2 instance, but I had never deployed a full-stack application on my own.
- The first step was fairly simple: using AWS S3 and CloudFront to host the frontend of the application. I did not really encounter any issues with this, utilizing online resources, documentation, and YouTube videos to help me along the way.
- Then, I rushed into the rest of the deployment process (lesson of its own). I was eager to get my application deployed, and rushed into unfamiliar territory with Elastic Beanstalk (AWS EBS), DNS Propagation, Inbound / Outbound Rules on Set Ports, and general Networking challenges. I also purchased the domain docu-dive.com before researching extensively how to register the said domain. So, when the time came to navigate AWS DNS and Route 53, I was stuck again. Despite reading abundant documentation and watching tutorials, as well as attempting to debug with ChatGPT, I could not solve the problems I had.
- I spent many days at this stage, working long hours and trying various potential solutions without success. In the end, some of my errors were related to my REST APIs, but I didn't know where exactly they were stemming from, since I rushed into the deployment process and was overwhelmed by the unfamiliar territory.
- Also, at this stage, I was worried that the Ollama server was going to cause issues. When I was testing locally prior to deployment, I'd often have issues since Ollama would push through frequent updates and I did not have a CD pipeline set up to adjust accordingly - when I did not catch the need for updates, I'd go through a frustrating debugging process only to realize that the server needed an update. Foreseeing possible issues after deployment at the AWS-level, I pivoted from using Ollama's 7B Mistral model to using OpenAI's API.

Backtracking, Researching Docker and Containerization:

- I decided to backtrack and research Docker and Containerization. I was able to write down a step-by-step process for deployment that I understood from front to end.
- As I went through the deployment process, I recognized multiple bash scripting efforts that were mundane and repetitive, so I automated them by developing some overarching bash scripts, which executed the redeployment process at the expense of a single command - this saved me a lot of time, and made me realize how important it is to automate such things, and in an industry-setting, to develop internal tools that streamline these processes.
- I also was better prepared for error-checking at this stage. I was constantly checking AWS logs, conversing with ChatGPT, reading forums to research my errors and learn from others, monitoring AWS and the health of my instance, seamlessly working among security groups, inbound and outbound rules, custom JSON policies, the nginx reverse proxy configuration, and more. **My first failed experience at deployment was immensely important for my second successful one.**

Tech Stack for Deployment:

- Docker:
  - Building Docker images.
  - Managing Docker containers.
  - Using Docker Compose for multi-container applications.
- AWS:
  - EC2:
    - Setting up and configuring EC2 instances.
    - Managing security groups and inbound/outbound rules.
  - ECR:
    - Pushing and pulling Docker images to/from AWS Elastic Container Registry.
  - IAM:
    - Configuring AWS credentials and policies.
- Nginx:
  - Setting up Nginx as a reverse proxy.
  - Configuring Nginx for SSL termination using Let's Encrypt certificates.
- FastAPI:
  - Adjusting RESTful APIs with FastAPI, debugging new-found errors / shortcomings
  - Handling CORS with FastAPI middleware.
  - Managing routes and endpoints for file uploads and database interactions.
- Certbot:
  - Generating and renewing SSL certificates with Let's Encrypt.

    - Integrating SSL certificates with Nginx.

Networking and Security:

- DNS Configuration:
    - Managing DNS records for custom domains.
    - Configuring A and CNAME records.
- SSL/TLS:
    - Securing websites with HTTPS.
    - Setting up SSL certificates for secure communication.
- Security Groups:
    - Configuring inbound and outbound rules.
    - Allowing specific IP ranges and ports.

Final Deployment:

- Deployment Overview:
    - Described the end-to-end deployment process from Python files to Docker images and AWS services.
- Preparation:
    - Organized Python backend files and created Dockerfiles for both frontend and backend.
- Docker Image Management:
    - Built, tagged, and pushed Docker images to AWS ECR.
- ECS and EC2 Setup:
    - Configured ECS clusters, task definitions, and services. Set up EC2 instance for Nginx.
- Nginx Configuration:
    - Configured Nginx as a reverse proxy to route traffic appropriately.
- Deployment Script:
    - Automated the process of building, tagging, pushing Docker images, and updating ECS services with a Bash script.
- Verification:
    - Listed tasks in ECS to ensure services were running correctly.

For detailed, technical instructions on deployment see:

READ.ME: https://github.com/pdm21/RAGv1-Full-Stack/blob/main/README.md

If I were to re-do everything, or make another web application soon:

- I would spend more time in the systems-design stage. I would plan out the application from front to end, considering the pros and cons of different tech stacks and foreseeing challenges that may arise along the way. This way, I can come up with the best approach for solving the problem at hand, while preparing a strategy that remains sound amid challenges related to frontend/backend integration, REST API creation and routing, AWS-related IAM issues, Cloud-level deployment obstacles, and other issues I learned from.
- I would improve the application by enabling CI/CD pipelines, so that I can consistently update my code and make changes without worrying about manual, redundant processes related to taking down the application and re-deploying it again.
- I would push to GitHub more frequently. Although it sounds obvious, I sometimes overlooked this while in a flow state. I realized how important it is to only make small adjustments between code-pushes. I would also take breaks between "pushes" and new versions, to ensure I am still on track and evaluate my progress. This would be like simulating a code review that would take place in the industry within a team setting.

In a brief summary, I was able to build a full-stack multi-document-search web-app by utilizing Retrieval Augmented Generation techniques to get the most out of Generative AI LLMs. I used React.js for the frontend, and Python, Langchain, AWS Bedrock, ChromaDB, and FastAPI for the backend. I used Docker to containerize the frontend and backend, AWS ECR to store the Docker images, and AWS EC2 to run the containers, with Nginx configured on EC2 to manage traffic and SSL/TLS certificates for secure access.

Thanks for following along.

LinkedIn: https://www.linkedin.com/in/pandelismargaronis/
GitHub Repositories:
    Docu-Dive: https://github.com/pdm21/RAGv1-Full-Stack
    About-Docu-Dive: https://github.com/pdm21/About-Docu-Dive
Docu-Dive: www.docu-dive.com / docu-dive.com
Hosted Website: https://about-docu-dive.netlify.app/

If you have any feedback, questions, or other comments, please don't hesitate to reach out to me: pmargaro@hamilton.edu