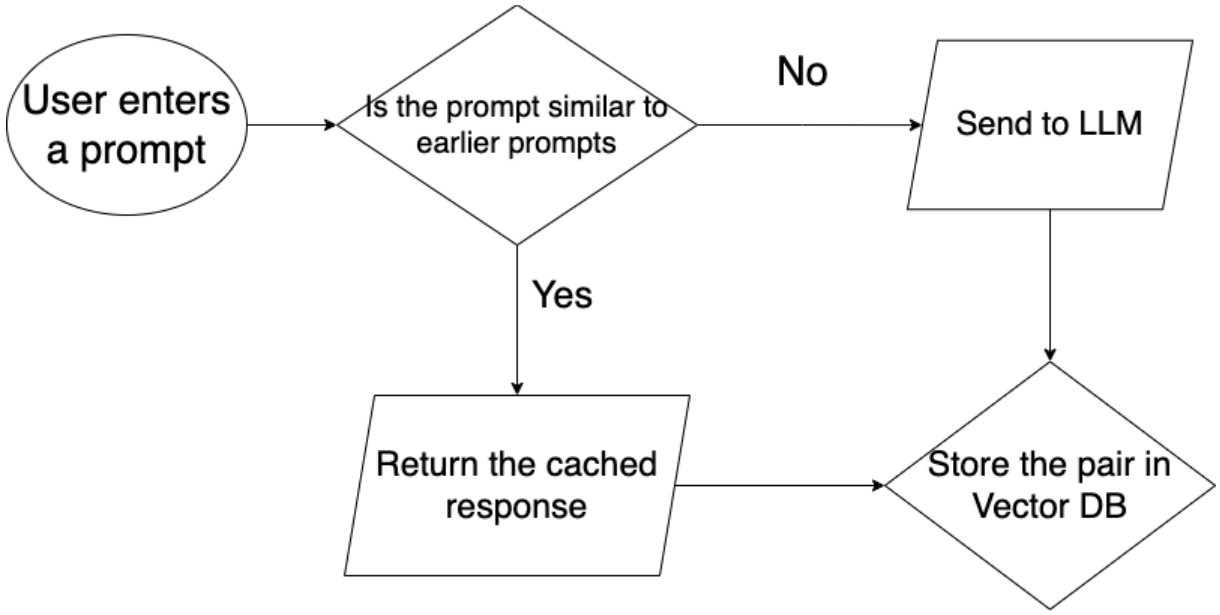
# Intent Detection

## Problem Solving Process

The whole point of this project was to build a system that can automatically classify the prompts into three specific categories which are course equivalency, program pathway, and others. The main challenge wasn't getting decent accuracy; the real problem was building a system which can be useful when tons of similar queries are coming and don't want to waste the computational resources for classifying the same thing repeatedly.

I started with a basic approach – first begin by connecting to the vector database (Weaviate in this task) as it is the main component in this task. I created a collection with just two columns: 'prompt' and 'response'. Then, I started Ollama which is used to run an LLM locally. Played with some prompts to see how the model works and then created a modelfile to write a system prompt such that the language model would act and generate as per the system prompt. So, next step was to connect the model with the vector database and set the similarity function to return cached responses when a similar prompt was entered.

The process was pretty straightforward once the architecture was figured out:




The tricky part was getting the similarity threshold right. Too high, and all the prompts go to LLM; too low and I would be getting irrelevant cached results.

## Overall Framework

The system has three main components that work together:

1. FastAPI Backend: This handles all the API requests and orchestrates everything. It receives prompts, manages the similarity search, calls the LLM when needed, and store results. The API is pretty straightforward – just one main endpoint that takes a prompt and similarity threshold, then returns the classification along with whether it was cached or freshly generated.
2. Weaviate Vector Database: This is where the magic happens for caching. Every prompt gets vectorized and stored along with its response. When a new query comes in, we search for similar vectors and if we find something close (based on the similarity threshold), we just return that cached result instead of bothering the LLM.
3. Ollama with custom model: I used Llama 3.1 8B as the base model but created a custom modelfile with very specific system prompts. The model is trained to output exactly one of the three categories: 'course equivalency', 'program pathway', or 'others'. No extra text, explanation or assumptions just classification.

The frontend is just a bonus – it's a clean chat interface that lets users interact with the system and adjust the similarity threshold in real-time.

## Tools and Systems Used

FastAPI: Chose this because it's fast, has great automatic API documentation, and handles async operations well. The built-in request validation with Pydantic models made it really easy to ensure the API gets the right data format every time.

Weaviate: Went with Weaviate Cloud because it handles vector operations out of the box. Weaviate's integration with text embeddings was straightforward and it was easily manageable. Plus, their free tier was enough for testing.

Ollama: This was perfect for running the LLM locally. No API costs, good performance and I could customize the model with my own system prompts. The 8B model was perfect for this task as it balanced speed and accuracy.

Llama 3.1: Used this as the base model because it's proven to work well for classification tasks and 8B parameters gave me good performance without needing crazy hardware.

Sklearn: Just needed this for the classification reports. It does the job well for evaluation metrics.

For the frontend, I kept it simple with vanilla HTML/CSS/JavaScript as I just needed a chat interface.

## Evaluation Results

Ran this system on 90 prompts (30 for each label). Here's what I found:

The overall accuracy was decent when I tested for the first time around 88%. I felt that I should tweak the system prompt as some prompts from 'others' label were misclassified as 'program pathway'. I reduced Temperature from 1 to 0.5. Fortunately, accuracy improved to 92%. SO started testing different Temperature settings and tweaked the system prompt a few times till my final test. At last, I achieved 96% accuracy. As the prompts were running on a new collection whenever I tested, very less responses were cached. But after it had enough prompts in the database, I started seeing cached responses. They were very fast as it didn't pass through LLM.

The classification worked best for clear-cut cases. "Program Pathway was the most accurate as all prompts returned correct answer. "Course Equivalency" prompts were also pretty easy to classify because they usually contain certain keywords. The "others" category was trickier because it's basically everything else. Sometimes the model would misclassify general questions as "program pathway" but the error rate was acceptable.

One interesting finding was that the similarity threshold setting really mattered. At 0.8, all responses were going to LLM because none of the prompts were that similar. At very low threshold, I would get false positives. So, I kept 0.6 as similarity threshold. I know this is very low similarity but our model which converts embeddings sometimes generate variations in embeddings even if the prompt is exactly same. So, this creates a decent difference in ideal and real cases, that's why I kept it at 0.6 and for now it's working well. The final classification report is shown below.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| course equivalency | 1.00 | 0.93 | 0.97 | 30 |
| others | 1.00 | 0.93 | 0.97 | 30 |
| program pathway | 0.88 | 1.00 | 0.94 | 29 |
| accuracy | | | 0.96 | 89 |
| macro avg | 0.96 | 0.96 | 0.96 | 89 |
| weighted avg | 0.96 | 0.96 | 0.96 | 89 |

The system stored every interaction in Weaviate, so the cache kept growing and getting better over time. This is exactly what we'd want in a real-life system.

## GenAI Assistance

Using AI for this project was honestly essential, but not in the way most people think. I didn't just copy-paste code from ChatGPT and call it a day. Instead, I used it more like a helping hand.

For testing, I used GenAI to help generate the prompts in my dataset. This was crucial because I needed all kinds of variation to really test the model's ability. The AI helped me think of different ways students might phrase the same questions, edge cases I hadn't considered, and variations in language that would challenge the classification system. Without this variety, I would have just tested obvious cases and missed potential failure points.

For the frontend, AI was useful for the CSS styling and making the interface look decent. I'm not a frontend person, so having AI help with the animations and responsive design meant I could focus on the actual functionality. For the backend, GenAI helped me understand different approaches and clean up my code to make it more efficient.

Overall, AI probably cut my development time in half, but the key was knowing when to use it and when to just figure things out myself.

The final system works well and demonstrates the core concept: intelligent caching can make LLM-based classification systems much more efficient without sacrificing accuracy. In a real deployment, this approach could save significant costs and latency.