Episteme - Documentation

Introduction - What is Episteme?

Episteme is a web application designed to accelerate the initial research process for investors and traders by summarizing and structuring investment theses from online financial communities such as subreddits and Seeking Alpha. Upon entering a stock ticker or company name, the application quickly provides a concise overview of what the company does, along with organized arguments supporting bullish (positive) or bearish (negative) perspectives. These arguments are automatically extracted from various sources and critically assessed by analyzing comments, effectively performing a peer-review process to ensure their validity and relevance. Additionally, Episteme calculates an overall market sentiment score, offering a quick snapshot of the general market attitude toward the stock. This structured aggregation of crowd-sourced insights not only streamlines preliminary research but also helps users discover potential investment ideas and evaluate them efficiently.

Backend

Base Setup

Credentials

- All password/credentials/access data is stored in an .env file in a config folder which is included in the .gitignore
- They are then loaded in via dotenv
- I chose .env because saving credentials as environment variables is the standard. However I wanted to be able to save and load them directly to and from a file so I could easily share this file to other workstations when I'm working on it on my laptop for example
- I didn't like working with relative paths since it seemed messy therefore as part of the setup process the user has to set the absolute path to the .env file as an environment variable called `ENV_PATH`
- `ENV_PATH` is then accessed by the program to know where the config is saved

Environments

The app is coded using a Python backend and a React.js Frontend. I chose React because I'm familiar with it and working with a framework is much easier and more straightforward

than using vanilla js. The hooks are very useful and the function based component make the code much more readable and easier to work with.

I chose Python because I am very familiar with using it for backend tasks and because it is the best language for working with AI and Machine Learning like I'm doing here. Sending and receiving requests is straight forward using FastAPI and in combination with SQLAlchemy it makes working with databases much simpler. I'm using FastAPI because the app is only gonna consist of a couple of main endpoints which means the routing part will be relatively easy. Also FastAPI just makes the coding experience much more straight forward and enhances readability significantly since type checking and raising errors works in a much more pythonic way than it does with alternatives like Flask (where I would have to sometimes write dozens of if/else statements just to check if the query meets basic requirements).

It is worth noting that I am planning on rewriting most of the codebase in Golang in the future for performance reasons. If I further decide to implement my own machine learning algorithm I will keep that part in Python but the rest will likely run faster in Go. Additionally it makes a lot of sense to use it because of it's great asynchronous capabilities.

- The app is built inside a virtual environment to prevent dependencies crashing into each other

---

Scraping

The scraping process is essential because it's the gateway to the investment theses which in turn are the heart of the application. This is how the app is going to get different investment theses and their respective criticism. The scraping part of Episteme is subject to change in the future in case I add different sources of investment ideas or otherwise fine-tune the search criteria for the information.

Reddit Scraper

To obtain investment theses from Reddit, I utilized PRAW, an unofficial but widely used Python wrapper for Reddit's official API. PRAW allows free and straightforward scraping of Reddit posts.

The approach involves iterating through a predefined list of relevant subreddits. For each subreddit in this list, a specified number (`num_posts`) of recent posts is retrieved. From each post, the necessary information is extracted, stored in a dictionary, and appended to a list of dictionaries. Finally, this structured list containing all scraped post information is returned for further analysis.

Seekingalpha Scraper

SeekingAlpha doesn't have an official API so I'm using an unofficial one from rapidApi. (free account has a rate limit of 500 requests per day) - *as of 07.04.2025*

- first get a list of all posts/analysis belonging to the given ticker. This returns a list of post ids
- for every post then get the details:

```
post_info = {

            "source": "seekingalpha",

            "title": title,

            "author": author_name,

            "time_of_post": time_of_post_formatted,

            "url": absolute_url,

            "content": filtered_content,

            "comments": comments

        }
```

To get the comments 2 additional requests have to be ran:

- The first request to get a list of all comment ids
- A second request to get the content of each comment by passing the list of ids

I initially had to request the content of each comment separately since the API wasn't clear on that. This was incredibly inefficient so I tinkered around with the API and figured out that `comment_ids` is allowed to be a list. This means it's allowed to get the contents of all comments at once.
This leaves us with the following formula to calculate the amount of requests needed:

```
1 + num_posts * (1 + 1)
```

- the first 1 is to get the **post list**
- the second 1 is to get the **list of all comments**
- and then the third one is to get the **contents of each comment**

Lastly the post content and the comment contents are cleaned up by removing script & style elements, removing URLs and whitespace and unescaping HTML entities.

Database

The next step is creating the database system and structure. I do this before the AI part because it's crucial to save the AI output to the database correctly. Therefore it makes sense to develop this first since I already know what content will be put into the database.

I'm using PostgreSQL for the database. I'm working with structured data so a noSQL approach like mongodb doesn't make a lot of sense. It's also a lot of data I'm dealing with and that data is relational. So for performance and efficiency reasons I won't use SQLite either. Lastly I just like PostgreSQL more than mySQL. It's faster than MySQL by many benchmarks, has a lot more features, better support of SQL standards, and MySQL is not officially supported by many linux distributions. MySQL had the advantage that it was easily available on cPanel and Plesk hosts but since I don't use php and overall these advantages are fading and it doesn't really matter much anymore. Additionally PostgreSQL works great with SQLAlchemy which I've gotten comfortable using so this is my choice.

Database Structure

Entity-Relationship Overview

- **Tickers**
  Stores stock symbols and related metadata.
- **Posts**
  Contains posts or articles related to a ticker. This now also includes an optional `author` field.
- **Points**
  Represents extracted investment thesis points from posts. Each point includes a sentiment score, the extracted text, and an embedding vector for semantic similarity, along with a flag indicating if any criticism exists.
- **Criticisms**
  Contains valid criticisms linked to a specific point, optionally referencing the comment that inspired the criticism.
- **Comments**
  Stores individual comments or feedback related to posts, which may also be referenced by criticisms.

Tables and Fields

1. **Tickers**

Stores information about each stock ticker.

- **Fields:**
  - `id` : Primary Key, Integer.
  - `symbol` : String (up to 10 characters), Unique, Not Null.
  - `name` : String (up to 100 characters).
  - `description` : Text – an AI-generated description based on financial data.
  - `overall_sentiment_score` : Integer, constrained with a check ("overall_sentiment_score BETWEEN 1 and 100"), Nullable.
  - `last_analyzed` : DateTime – timestamp of the last analysis.
  - `description_last_analyzed` : DateTime – timestamp of the last description update.
- **Relationships:**
  - **Posts:** One-to-many relationship with Posts (cascade delete enabled).
  - **Points:** One-to-many relationship with Points (cascade delete enabled).
- **Additional Note:**
  - A custom `__repr__` method is defined for debugging and logging purposes.

---

2. Posts

Contains posts or articles from various sources related to a ticker.

- **Fields:**
  - `id` : Primary Key, Integer.
  - `ticker_id` : Foreign Key referencing `tickers.id` , Not Null.
  - `source` : String (up to 50 characters), Not Null – e.g., Reddit, Substack.
  - `title` : String (up to 250 characters), Nullable.
  - `author` : String (up to 100 characters), Nullable – represents the author of the post.
  - `link` : Text – URL to the original post.
  - `date_of_post` : DateTime – publication timestamp.
  - `content` : Text – full content of the post.
- **Relationships:**
  - **Ticker:** Belongs to a single Ticker.
  - **Points:** One-to-many relationship with Points (cascade delete enabled).
  - **Comments:** One-to-many relationship with Comments (cascade delete enabled).
- **Additional Note:**
  - A custom `__repr__` method provides a concise string representation of each post.

---

3. Points

Extracted investment thesis points from posts.

- **Fields:**
  - `id`: Primary Key, Integer.
  - `ticker_id`: Foreign Key referencing `tickers.id`, Not Null.
  - `post_id`: Foreign Key referencing `posts.id`, Not Null.
  - `sentiment_score`: Integer, with a check constraint ("sentiment_score BETWEEN 1 and 100"), Not Null.
  - `text`: Text, Not Null – the extracted point or thesis.
  - `criticism_exists`: Boolean, default set to False – indicates whether any valid criticism is linked.
  - `embedding`: ARRAY(Float) – stores a semantic embedding vector for similarity comparisons.
- **Relationships:**
  - **Ticker:** Belongs to a Ticker.
  - **Post:** Linked to the originating Post.
  - **Criticisms:** One-to-many relationship with Criticisms (cascade delete enabled).
- **Additional Note:**
  - The custom `__repr__` method returns a brief representation of the point including its sentiment score and a preview of the text.

---

4. Criticisms

Stores valid criticisms associated with specific points.

- **Fields:**
  - `id`: Primary Key, Integer.
  - `point_id`: Foreign Key referencing `points.id`, Not Null.
  - `comment_id`: Foreign Key referencing `comments.id`, Nullable – if the criticism is linked to a particular comment.
  - `text`: Text, Not Null – the content of the criticism.
  - `date_posted`: DateTime – timestamp indicating when the criticism was recorded.
  - `validity_score`: Integer, with a check constraint ("validity_score BETWEEN 1 and 100"), Nullable – quantifies the strength or validity of the criticism.
- **Relationships:**
  - **Point:** Belongs to a specific Point.
  - **Comment:** Optionally linked to a Comment from which the criticism originated.
- **Additional Note:**

- A custom `__repr__` method provides a concise summary of the criticism.

---

5. Comments

Contains individual comments or feedback linked to posts.

- **Fields:**
  - `id`: Primary Key, Integer.
  - `post_id`: Foreign Key referencing `posts.id`, Not Null.
  - `content`: Text, Not Null – the comment's text.
  - `link`: Text – optional URL or reference.
  - `author`: String (up to 100 characters), Nullable – the name of the comment's author.
- **Relationships:**
  - **Post:** Belongs to a specific Post.
  - **Criticisms:** One-to-many relationship with Criticisms (cascade delete enabled), allowing a comment to be referenced by one or more criticisms.
- **Additional Note:**
  - A custom `__repr__` method is defined for ease of debugging and logging.

Sample Data Entries

Tickers

id	symbol	name	description	overall_sentiment_score	last_analyzed	des
1	AAPL	Apple Inc.	...	85	2023-10-15 14:35	202
2	TSLA	Tesla, Inc.	...	70	2023-10-15 14:35	202

Posts

id	ticker_id	source	title	author	link	date_of_pos
1	1	Reddit	Apple's Q4 Earnings	AnalystA	http://reddit.com/post1	2023-10-15 14:35

id	ticker_id	source	title	author	link	date_of_pos
2	2	Substack	Tesla's New Model	AnalystB	http://substack.com/p2	2023-10-15 14:35

Points

id	ticker_id	post_id	sentiment_score	text	criticism_exists	embedd
1	1	1	85	"Strong sales in Q4"	False	[0.134, 0.298, -0.076, 0.415, 0.092, ..
2	2	2	32	"Concerns over production"	True	[-0.213, 0.412, 0.103, -0.349, 0.287, ..

Criticisms

id	point_id	comment_id	text	date_posted	validity_score
1	2	5	"Production issues are temporary"	2023-10-14 12:00	80

Comments

id	post_id	content	link	author
5	2	"Industry reports suggest that production issues are only short-term."	http://example.com/comment5	AnalystX

Setting up the Structure

I'm using SQLAlchemy so I first created a new file called `tables.py` in which I define the Database tables. Using the outlined structure this is relatively straight forward. I'll just create a class for each table containing each of the mentioned fields. For some fields, constraints have to be introduced (e.g. adding a constraint for sentiment_score to be between 1 and

100). This can be done by using SQLAlchemy's `CheckConstraint` function, which allows SQL injection and filtering based on the given argument.
I also need to define the relationships to other tables when using foreign keys. I'll use SQLAlchemy's relationship function and delete orphans when a parent gets removed.
Lastly I provided a `__repr__` function for each table to make it clear what the classes are made out of and therefore provide better readability.

Setting up the Database

Now that the structure is defined I will have to set up the PostgreSQL database so it can accessed in the main program. This means creating a `db.py` file which uses the database credentials to create the engine and session factory which will later be used to access the database.
When creating the SessionLocal object it's important to make sure to disable autocommit and autoflush. I disabled autocommit since I want explicit control over if and when changes are committed to the database and I disable autoflush because it can lead to performance issues when querying frequently (it also allows me to batch queries together and in general have more control over when data is written/synchronized to the database).

Next I'll will create a `deploy_db.py` script. This is optional and not strictly necessary, however I personally like the option to manually deploy the entire DB by just executing a script. `deploy_db.py` checks if the database already exists and creates it if it doesn't. My first approach was to connect to an established database and then check if pg_database (the database register) includes the specified database. However this did not feel pythonic at all and overall seemed like a bad practice since I had to inject SQL directly to go through with this approach. So I researched a bit and found there was actually a separate library called `sqlalchemy_utils` which has `database_exists` and `create_database` functions which take care of this problem in a cleaner way. The deployment script also tries to create the tables defined in the SQLAlchemy models if they don't already exist. This file will not be imported from any other one. However I will later include a check to ensure the database is present before running the main program. If it's not, the program will exit and print an error to the user suggesting he runs the `deploy_db.py` script.

Recommending Stocks to the User

When the User visits the website he will be able to type into a search bar the ticker or stock he wants analyzed. The app needs to:

1. Make sure the stock/ticker he enters is valid
2. Suggest him tickers/stocks which match with his input while he's still typing in real time. This is essential because I want users to be able to enter a stock name and not only a

ticker. Since stock names aren't all formatted the same I have to match the users input with known stocks so he doesn't have to perfectly enter everything.

How does one do that?

I first have to figure out how I'm going to get the stock data required so I can match the users input to known stocks. My first thought was using an API like yfinance (yahoo finance's API) however I quickly realized I'd have to perform too many API requests since I'd have to query the API in real time based on what the user is typing. It would also be significantly slower than an offline solution since the frontend would have to perform a request to the backend which then would have to perform a request to the API. The better way is to reduce online calls to services and use an offline solution. This means downloading the data to the server and then doing the matching/indexing internally.
I was able to find a JSON file from the official SEC website containing 9998 publicly listed companies (it's also very up to date from what I could tell). I'm going to load all stock data into a properly indexed database table and then perform queries against it using an indexed text search.

Indexing the Data

I have to tackle the problem of how I am going to quickly provide partial matches to the user. My first thought therefore was using an Index. The issue here is that since the app will be providing suggestions in real time it will have to search for entries based on partial inputs (The user will enter "goog" and is searching for google and I don't want to wait to give him suggestions until he types the whole word but provide them at this stage already). This is where I remembered that I read about a similar issue in an academic paper [1] a while ago and that there, they split the strings up into substrings and indexed those. After some research I found that this is called **Trigram Indexing** and is a perfect fit for my application. Trigram Indexing works by splitting every Key (in this case the tickers and names of the stocks) into all possible adjacent three character combinations and indexing them. By breaking down strings into trigrams the backend can quickly compare a search query to an indexed text field to find approximate or partial matches. I now can implement this functionality in the deployment script which will create the database, create the table, inject the data from the JSON file and create the trigram indexes automatically. I save the stock_index model into the `models` folder alongside the main database and then the database part of the recommendation backend is done.

Querying the Data on user input

Now that I have the database structure of the stock recommendation system set up I will start with the first step of the backend. As mentioned earlier I'm using FastApi for the backend. I'm using a modular approach in which I define the individual routes in a `routers` folder and then include those inside the `main.py` file in the root directory. In the `stock_query.py` route the app will accept a standard query `q` and set a limit of 10 results to be returned. It's going to use `ILIKE` to search for entries which include the query. This

operation is made significantly faster by the Trigram Index. It then returns the results as a list of Objects.

Now that the app is successfully querying the database for tickers/titles and returning them to the user I can focus on optimizing the order of the results. Right now the results are returned in the order in which they are found. This isn't a terrible approach and often works relatively well since the SEC file I got the data from was sorted by market cap in descending size. This leads to the largest companies being shown first and therefore the likelihood of the person looking for that stock is relatively high. However I want better and more sophisticated sorting, since I want to be able to show relevant results even for smaller stocks.

One way to do this is to check the similarity between the query and the ticker/title. This can be done by using PostgreSQL's `similarity` function to compute a similarity score. The results are then ordered by the "relevance score" which I assigned in descending order.
So I put this approach to the test and compared it to the default approach. For that I used the stock ticker "NU" with the stock title "Nu Holdings Ltd." and input "nu" as a query. Using the default approach the first hit was "TSM" or "Taiwan Semiconductor Manufacturing Co Ltd", this was because of the "nu" in "manufacturing" and because it had a larger market cap than NU, perfectly demonstrating my point of why I need to optimize. However NU was the second hit in the list. With the relevance score in place NU moved down to the third place, however this time the two results that were higher up were more relevant: "NRDE" (NU RIDE INC.) and "NUMD" (Nu-Med Plus, Inc.). The problem here was obvious, the algorithm clearly succeeded at sorting by relevance, only it focused on the title when the ticker is really (in most cases) more important.

Therefore for the next approach I'd have to prioritize the tickers. For that I can use the `CASE` statement to assign a higher priority to matches in the ticker field. I assign a full ticker match the highest priority of 1, a partial ticker match the priority of 2 and the rest (matches in title) a priority of 3. Now NU finally moved up to the top spot. However the results below were quite disappointing this time:
After not being able to find any more priority 1 or 2 results the algorithm returned only priority 3 results in alphabetical order. This means it just returned any stock that had "nu" in it's title in alphabetical order leading to the next result being "PPLT (abrdn Platinum ETF Trust).

After the above testing it becomes clear that the relevance and the case approach both had it's strengths and weaknesses so I'm going to try and combine them. I want to keep giving ticker matches higher priority over title matches however I also want title matches to be sorted by relevance. Therefore a good approach seems to be to use the relevance/similarity score system but add additional weighting to ticker matches. I'll combine PostgreSQL's `CASE` and `SIMILARITY` functions to

- Assign a higher weight for **exact matches**.
- Assign lower weights for **partial matches** and **title matches**.

I'll do this using the following approach:

1. **Exact Matches**
   - Assign a fixed high score (e.g., `1.0`) to exact matches on the `ticker`.
2. **Partial Matches**
   - Use `SIMILARITY()` to calculate how close the query is to the `ticker` and `title`.
   - Weight the `ticker` similarity higher (e.g., `0.8`) than the `title` similarity (e.g., `0.2`).
3. **Relevance Score**:
   - Combine these weighted scores into a single relevance value, sorted in descending order.

And indeed I now get "NU" as the top result followed by "NUE" (NUCOR CORP) and "NUS" (NU SKIN ENTERPRISES, INC.). After a bit of trial and error I found 0.7 weight for the ticker similarity and 0.3 for the title similarity to strike the best results. I could of course do this analysis in a much more sophisticated way by manually creating a dataset with ideal matches for certain queries and then automating evaluation by testing a range of weight combinations based on factors like Precision or Recall (How many of the ideal results appear in the top N results?) and then plotting the performance metrics to facilitate a data driven visualization of the results. However this seems a bit overkill for now so my personal approximation will have to do (Besides this begs the question of how much difference the user is actually even going to notice).

Validating User Input

Now that the user can input data using the recommendation algorithm it's time to actually get that data to the backend, validating it and then using it to provide an analysis to the user.
As I mentioned before I'm using FastAPI for the backend. This is mainly because the app doesn't have to receive a lot of input from the user so the API will be very simple. In essence it only has to take the ticker/name of the stock as input and then do all the analysis in the backend.

However there will be some options connected to that input.
I'll design the API in a very modular fashion (just like the entire project) to make it easy for us to add more options for the user later on. One of the options has to do something with the fact that all data generated from the scraping/analysis is saved to the database. The reason for that is optimization. If AAPL (Apple) was already analyzed 5 minutes ago it makes little sense to do an entire analysis from scratch again. Instead it will use the saved data from the previous analysis. This is done by combining a couple of options. I want to give the user the option to keep costs low by just accessing the already generated analysis from the database (if present) and not generating a new analysis at all.

For now let's start by adding the FastAPI routes for the analysis generation. I'll add a `/check-analysis` route which takes a GET request. It will take a ticker and title as queries.

For the reasons outlined above I now want to check if the inputted stock has been analyzed before so I'll write a simple helper function to check and return if the ticker has been analyzed and the date on which it was analyzed. It will return an `existing_analysis` boolean and a message asking the user if the stock was analyzed before (and when) and ask if they want to create a new analysis.
Now I can add some synthetic data to the database, include and use postman to check if the endpoint works as expected.

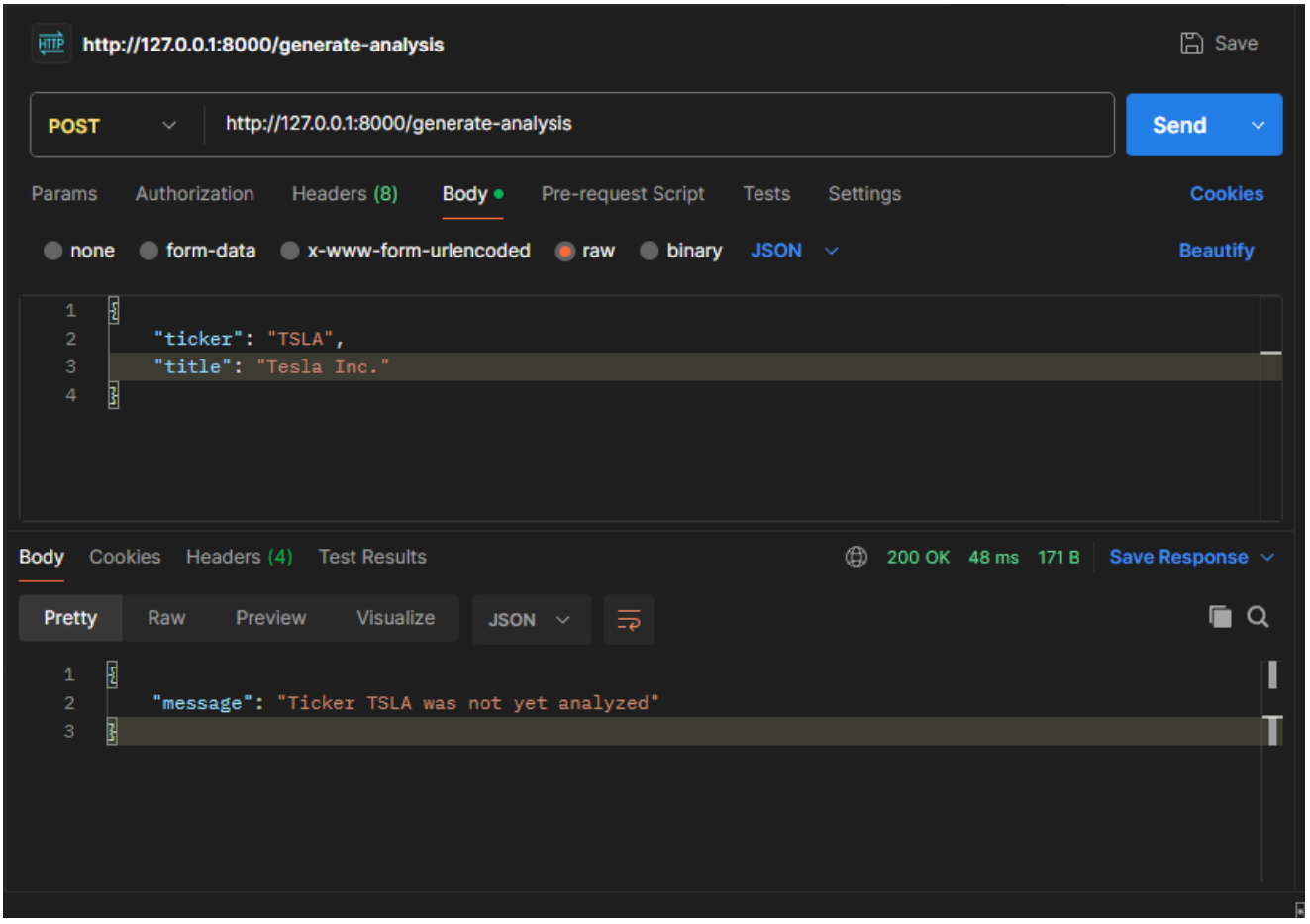


***Image 1.1** The response with a ticker that is not in the DB**

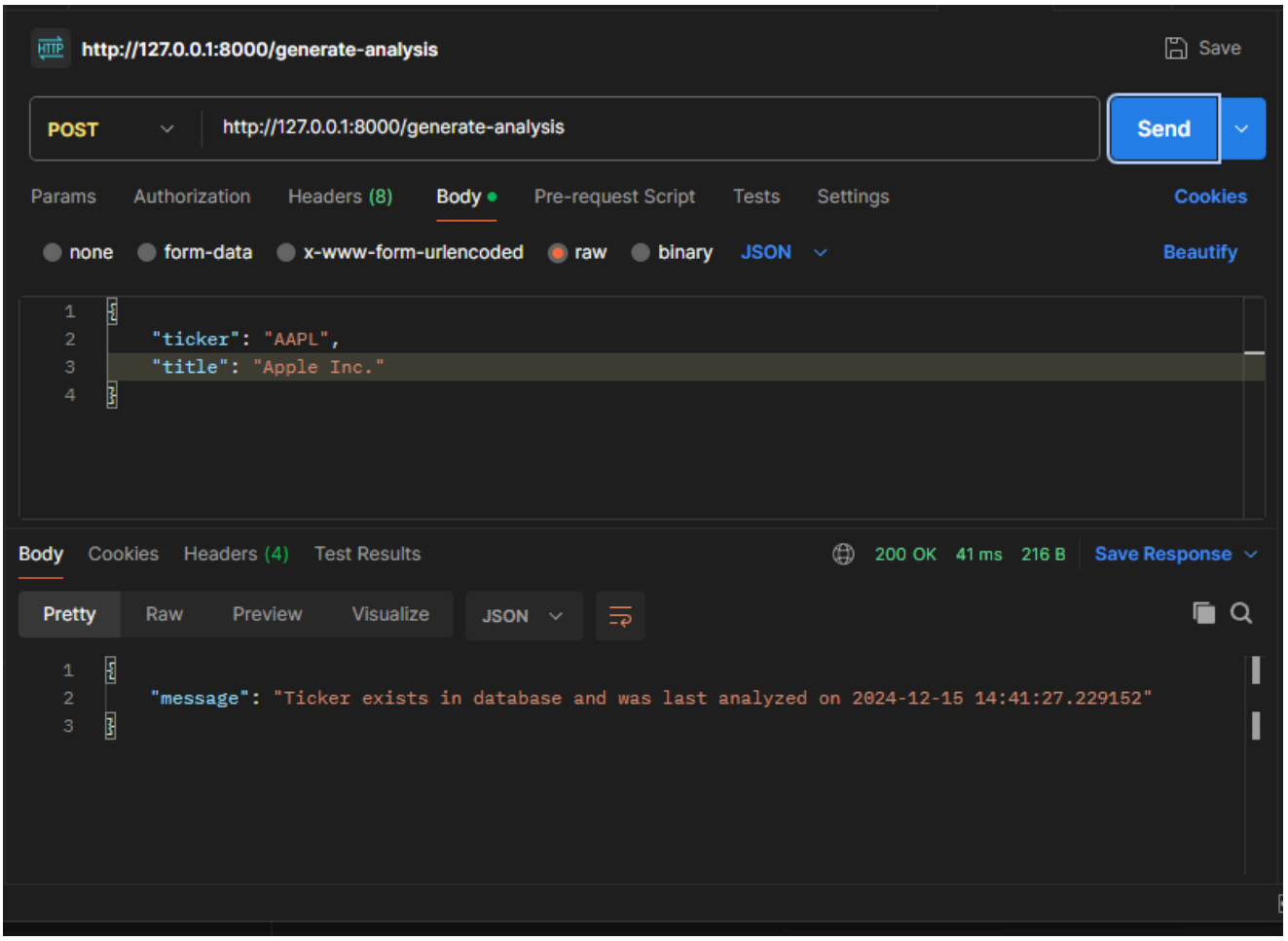


***Image 1.2** The response with a ticker that is in the DB**

Starting analysis and updating Frontend

Next I'll add a second endpoint that is going to get called when the user does want to create a new analysis after being prompted. This function will use FastAPI's Background Tasks functionality to execute the `start_analysis` function asynchronously in the Background. `start_analysis` is then going to call all the other modules that together perform the analysis in the right order. Together with this endpoint I'll also implement a dynamic loading screen which will tell the user at which point of the analysis the backend is currently at.

To update the Frontend on the current state the Frontend has to communicate with it in intervals. There's two main options to build this. I could, for example, go for a bidirectional approach and use webhooks or Server-Sent events to send requests to the Frontend and update it. An easier and simpler option is to just use a polling strategy where I set up a second endpoint which will return the current state of the analysis when polled. I then combine this with the Frontend checking this endpoint in intervals. This is the better option since "live" updates aren't really required for the application hence I don't need to add an additional layer of complexity.

The flow should look like this:

1. **Start analysis** (GET) `/generate-analysis?ticker=XYZ`
    - Response will include a task id under which it'll store the updates for the analysis: `"message": "Analysis for {ticker} started", "task_id": task_id, "status": "started"`
2. **Frontend calls** (GET) `/analysis-status?task_id=1234` in intervals.
    - Response will include the status updates: `"status": "Analysis started", "progress": 0, "ticker": ticker`

Old version

I'll store the tasks in a very simple in-memory storage. This **is not ideal for production**. Ideally I would use something like redis to make sure progress doesn't get lost in case the App crashes. **However** this would require setting up a separate redis server and this just isn't worth the trouble right now, especially because this isn't critical info but just basic progress updates.

I'll create `run_analysis.py` which includes the `start_analysis` function. I'll store the tasks as a `Tasks` dictionary with the keys being the task_id's and the values being a dictionary including the status, progress and ticker.

Within this function I'll just update the individual value of the key of `TASKS` after I run every individual analysis module which will be implemented later.

When an analysis is now started I'll generate a UUID using pythons uuid library and initialize the task in store with a status of "pending". Then the analysis gets generated via the aforementioned Background Tasks module and it returns the task_id to the Frontend.
I'll then add the `/analysis-status` endpoint which imports the `TASKS` dictionary takes the `task_id` as query and uses it to return the correct Task value (if the task is found otherwise it'll throw a 404).

New version (using redis)

I'm adding this part after having just deployed Episteme to production. I mentioned before that using an in memory storage for the tasks is not ideal for production. I initially had a few reasons in mind like error handling, keeping tasks alive and most importantly persistency across restarts.

However after deploying my app and getting a "Task not found" error when generating an analysis it hit me. The biggest issue in memory task storage would cause that I hadn't even thought of. When deploying using a service like gunicorn or similar we want high availability for our web app. Therefore instances of the app load asynchronously and in their own states. Meaning memory is allocated individually which means the different instances don't share the same memory space. This causes the worker responsible for fetching and returning the task status to the user not to have the same memory space as the worker responsible for keeping track of the tasks status in the first place.

This meant that when I deployed the app I had to migrate to redis if I wanted to or not. Luckily this was easier than I thought it would be. Since the task status isn't a lot of data by itself and I plan to clean it from storage after the analysis is done I didn't need a separate redis server or anything like that. Therefore I just integrated two helper functions which update and retrieve the task item from the redis storage. For this the task dictionary is loaded into and destructured to and from json to store it to redis in the expected format. I then just had to switch out all mentions of the `TASKS` dictionary with appropriate use of said helper functions. Lastly as I'v mentioned I want to clean the individual entries from the storage after the analysis is done. To do this I just provided the final update functions with an `ex` parameter which tells redis the TTL (time to live) for that entry. That way after the analysis is finished the task entry will only live for 15 seconds after which it expires and is cleared from storage (I put in 15 seconds to add some buffer for the frontend to render the updated state).

I wanted to add this paragraph so there's no confusion when looking at the code since it's now replaced with redis. To see the code before the changes to redis just have a look at the commit "switched from memory storage for tasks to redis" on the 9th of April 2025.

Putting the Scrapers together

Let's now start with the first step of the analysis process: The scraping.
As documented I wrote both of the scrapers (reddit and seekingalpha) at the very start of the project. Now comes the part where I implement them. I want to run the scrapers asynchronously for better efficiency and speed so I'll use ThreadPoolExecutor to do this. I create a new `scraping.py` file which will run the scrapers in separate threads track their progress and return the result of both scrapers in a dictionary when they're completed.

The scraping functions do have some basic input parameters which can be used to adjust what gets scraped. These options can be submitted by the user in his initial request to create an analysis and will later be available as "filters".

```
background_tasks: BackgroundTasks,

ticker:str = Query(..., description="The ticker symbol for the analysis"),

title: str = Query(..., description="The stock name"),

subreddits: Optional[List[str]] = Query(default=default_subreddits,
description="List of subreddits to scrape"),

reddit_timeframe: Optional[str] = Query(default=default_reddit_timeframe,
description="Timeframe to scrape posts(e.g., 'hour', 'day', 'week', 'month',
'year', 'all')"),
```

```
reddit_num_posts: Optional[int] = Query(default=default_reddit_num_posts,
description="Number of reddit posts to scrape"),

seekingalpha_num_posts: Optional[int] =
Query(default=default_seekingalpha_num_posts, description="Number of
seekingalpha posts to scrape")
```

Error handling

Now this is all pretty good but I now have to get to something which I've been putting off for a bit too long. And that's error handling. Now I do have some basic error handling here and there as outlined before. However I need good error handling for the analysis specifically. Since the analysis takes a lot of moving parts and puts them together, there's a lot that can go wrong. And when it does I (or the user) need to know what went wrong.

Luckily I already built a way to communicate the status of the analysis to the user by adding the `/analysis-status` endpoint and updating a dictionary with the current status of the analysis.

While looking at the `/analysis-status` endpoint I realized I implemented it in a pretty suboptimal way so I'll first improve that.
Currently the endpoint is blindly returning the value of the task_id key in `TASKS`. However since this information will be displayed directly to the user, this means if there's any error or problem we might end up showing the user whatever output was saved to the task_id value without validating it. This is actually bad out of two reasons. First I of course don't want to confuse the user with some strange output but just want to tell him in a simple way that something might have gone wrong. Second this is horrible out of a security perspective since this could lead to use revealing some server internal (possibly sensitive) output to the frontend.

Long story short I'll modify this by instead returning a fixed dictionary of the information I want to display and only retrieve what I really need for each value and display it to the user.

This looks something like this:

```
return {

"status": task.get("status"),

"progress": task.get("progress"),

"error": task.get("error"),

"ticker": task.get("ticker")
```

```
}
```

Note that I use "get()" so the program doesn't crash in case the values are missing.

This still doesn't 100% fix the issue though. As I mentioned I don't want to blindly return every error to the user. Therefore I can't just raise everything to the `TASKS` storage. Instead what I'll do is wrap the `run_analysis.py` flow in a try-except block. If there is any exception however it'll update the `error` key to a standard error message telling the user at what stage the analysis failed (based on the `status` key) and to contact support. I'll then use the logging library to log the error on the server. For that I set up a logging config in main.py specifying two logging files: errors.log & warnings.log. Errors are logged to both the errors.log and warnings.log file but warnings will only be logged to the latter. Errors are exceptions that break the main flow while warnings are exceptions that get skipped. I also specify the format in which errors should be logged.

Database error handling

Database error handling is of course a bit more tricky because I don't want to leave sessions open when an error occurs or not save data or, even worse, save wrong data.
What I'll do here is add a centralized context manager which wraps the session and ensures a rollback if any error occurs. I'll call this context manager `session_scope` and replace all mentions of `SessionLocal` with it. I'll use the contextlib library to do that.
This also makes the code a lot cleaner since I won't have to manually close or commit every time because the context manager will take care of that.

There's also the handling of database warnings. As I mentioned previously for some exceptions weIjust want to log warnings and continue. One example for this is when the app is going through a list of posts or points and there is an issue with one specific item. I don't want the whole flow to break simply because an exception was caught in one post. Instead I'll log the warning an continue.
However with database operations because of the global context manager if any exceptions are caught it will automatically roll back the entire outer transaction including all other posts. Therefore I'll use SQLAlchemy's nested sessions feature. This feature is essentially an abstraction of the PostgreSQL **savepoint** feature which allows the application to set savepoints within a transaction and rollback part of a transaction without affecting the rest. Using that I can simply roll back the part of the transaction that failed (e.g. the specific post that raised the exception).

Using sets to filter out previously checked posts

Now that the data is scraped and it's packed neatly in lists of dictionaries it's time to return to an efficiency feature I have mentioned before. If the ticker has been analyzed before I don't

need to analyze the same posts again so I'll crosscheck with the database to remove any posts that were analyzed before from the data.

First I'll check if the ticker already exists in the database. If it does not it hasn't been analyzed so I'll just create a new ticker in the database so I don't have to do it later and return the initial data back unchanged.

However if the ticker row is found the script start filtering.
Now I're gonna crosscheck the posts by comparing the URL's of the scraped posts with the ones in the database. Now how do I do that most efficiently? Well one might think of just crosschecking by storing the scraped URLs in a list and then iterating over them to check if they're in the database. But wait, if that person would have paid attention to Herr Weins Informatik Q3 Class you would have noticed this operation is O(n) time complexity. Now luckily I know a data structure that is much more appropriate and one of my favorite data structures in general. A set. Since sets work using a hash table under the hood and just calculate the hash of the variable and look it up in the internal table, using a set here leads to the operation working in O(1) time complexity.

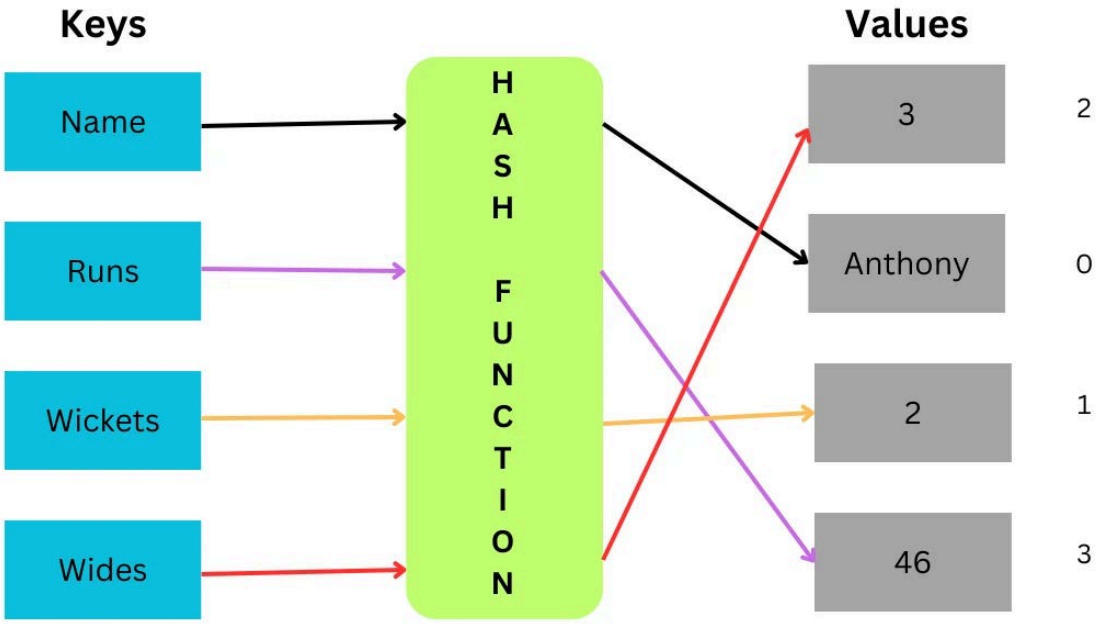


***Image 2.1** An image to demonstrate graphically how sets work and why they're useful here. Hash functions never return the same output for an input which leads to fast look-ups since one doesn't have to iterate over the entire set if they know what they're looking for. Instead they can calculate where the value would be stored based on the output hash and therefore check if it's present at the according position in memory (this is also why hashes are used to encrypt passwords in databases, the companies themselves can't see the users passwords but they **can** check if a given password is the correct one since they will both return the same hash).*

So I first gather all scraped URL's in a set. Then I run a single SQL query to get all the links present in the Database that are also present in the scraped posts dictionaries. This, by the way, is the second advantage of using a set. Since SQLAlchemy has to convert the data I

input to the query into a tuple, this way the app just needs to send one single query instead of building an inefficient for loop.

I then convert these existing links which I know to be duplicates into a set as well. I now run a for loop to compare these two sets with each other and remove all links that are present in both sets. And here it becomes apparent that if I had done this using a list this would be linear time complexity **for each** membership check. It's easy to see how this adds up quite quickly. However since the script is comparing sets using under the hood hash calculation and tables it ends up with constant-time look-ups

Now I get that all these minuscule efforts at scraping just a little bit of efficiency off every operation seem overkill at first. The seem unimportant and likt they're all just little changes only interesting for engineers. However they are not. An analysis will likely take a minute to maybe even a couple of minutes to finish. This is mostly because of the use of Large Language Model API's. This is a pretty long time and since I cannot control how fast Open AI's API will be I should focus on improving **what we can control**. And this means making everything as efficient as possible given the framework and knowledge so that all these little tweaks end up compounding into an overall faster experience.

Adding Posts to the Database

After making sure the posts which were scraped aren't already in the database I have to save said posts in the database. There's essentially two things that I have to save to the database:

1. Our analyzed theses (Points & Criticisms the AI extracted from the posts)
2. The post info which will be linked to the analyzed data so that the app can later show the source of each point/criticism (URL of the post, title, author, etc.)

The order doesn't really matter too much although I decided to essentially first save the posts to the DB.

However remember this is the first time I'm saving anything proper to the database from the script which means I'll first have to do some setup work. At the start of the flow (after receiving the task to do an analysis on a given ticker) the first thing I'll do is check if that ticker is already in the database. If it isn't I'll first have to create a new Ticker entry in the DB.

The script will create a new ticker, setting the symbol to the ticker and retrieving the name of the stock by looking the ticker up in the `stocks` database I implemented earlier to suggest stocks to the user.

After scraping the posts the app will go through each post's info and save it to the database, linking it to the ticker. The program flow will keep track of all posts primary id's after

committing them to the database. This way it can then again pull the new posts info from the database and pass said info to the analysis module in the next step.

Summarizing posts and assigning sentiment

Now I can start with the first step of the analysis. The summarization. I first intended to use gpt-3.5-turbo to summarize the points and then assign sentiment to them separately however after testing I found that using gpt-4o to summarize and assign sentiment all in one step actually works better.
An even better way to do this is to only use Large Language Models like GPT to summarize the points and then use a more traditional transformers based approach to assign sentiment. I should train said model specifically on financial data so that it's good at assigning sentiment in that venue. I could do this using a neural network (transformers) and utilizing transfer learning to use a foundation of pretrained parameters with a preset learning rate for the stochastic gradient descent to build the Deep Learning network. This would mean high accuracy but high costs and more compute. A more traditional approach would be XGBoost with engineered features (word counts, sentiment scores from lexicons like VADER/Loughran-McDonald) or a Random Forest trained on TF-IDF or word embeddings (Word2Vec, FastText) which are much cheaper than Transformers based models. A good middle ground would be to use recurrent models like Bi-LSTM with attention or **GRUs** using embeddings (GloVe, Word2Vec) these can capture sequential dependencies while being easier to train than transformers based models.
However I was advised to not build my own model for this BLL so I'm just going to go with a simple GPT-4o approach.

The script will use the previously saved ID's from the posts it just scraped and get the content of the posts from the database. I then use some prompt engineering to write a prompt that instructs GPT to extract factual thesis points from the post and assign them a sentiment score from 0-100 above 50 being bullish and below 50 being bearish. Then I provide it with the ticker the analysis is for (So it doesn't get confused and extracts points for the wrong stock), the stocks name and the content of the post.
OpenAI luckily supports passing of a JSON schema to force GPT to output the content in a given format so the output can be easily processed without having to make use of a whole lot of Pydantic. The JSON schema I built for this prompt looks as follows:

```
{
    "type": "object",
    "properties": {
        "thesis_points": {
            "type": "array",
```

```
            "items": {
                "type": "object",
                "properties": {
                    "point": {
                        "type": "string",
                        "description": "The extracted thesis point text."
                        },
                    "sentiment_score": {
                        "type": "integer",
                        "description": "The sentiment score, where 50 is
neutral, above 50 is bullish, and below 50 is bearish."
                    }
                },
                "required": ["point", "sentiment_score"],
                "additionalProperties": False
            }
        }
    },
    "required": ["thesis_points"],
    "additionalProperties": False
}
```

Note that I'm coding all of this using OpenAI's asynchronous library in combination with asyncio. This means all posts are getting summarized at the same time leading to O(T) in wall-clock time, no matter how many posts are added it's still going to be taking the same amount of time to analyze all of them as it would to analyze the longest one of them.

This also led to an interesting problem I had to solve. Now the entire function is asynchronous and asyncio is used to parallelize both this function and the filtering out of points to which I'll get to next. After employing these parallel functions the analysis status suddenly didn't work anymore. At least not properly. When trying to get the status of the analysis the request would stall and it would only return a response after around 10-15 seconds. Since I just built in asyncio it was pretty clear that there was likely some synchronous function which was blocking the (now asynchronous) program flow leading to the `/analysis-status` endpoint not being able to run until said function was done. After utilizing the timing library to figure out which part of my program flow was taking so long I

landed on the scraping flow. In hindsight this is pretty obvious. I'm utilizing ThreadPoolExecutor to thread the scraping which means there are synchronous blocking network calls within the scraping modules. This is solved pretty easily by just offloading these blocking calls to `asyncio.to_thread()`.

Fetching a public company profile

Since the service will act as a first "overview" of a company by showing public sentiment & theses it makes sense to also display a basic profile of the company. This will be basic stuff like ticker, name, price, description, etc.

For that I'll use Financial Modeling Prep a public (and free - *up to 250 requests per day - 07.04.2025*) API that allows the backend to fetch basic company info. It'll just fetch the API for said info and return it. Which results in the following response for the ticker AAPL:


```
{
    "symbol": "AAPL",
    "companyName": "Apple Inc.",
    "image": "https://images.financialmodelingprep.com/symbol/AAPL.png",
    "website": "https://www.apple.com",
    "description": "Apple Inc. designs, manufactures, and markets
smartphones...",
    "price": 213.49,
    "exchangeShortName": "NASDAQ",
    "mktCap": 3207068129000,
    "industry": "Consumer Electronics",
    "dcf": 149.70178547238896,
    "beta": 1.178
}
```


Note that the image key always returns a url which should technically lead to the company's logo. It does this when there is no logo too in which case it will link to a 404 URL. I'll therefore only include the URL in the data if the link returns a 200 status code

We'll then also use the yahoo finance API to fetch some additional data about the stock that the FMP API doesn't provide for free. I'll get Forward PE, the mean analyst rating and I'll use the yahoo calendar to extract the next earnings call date from the dataframe the API returns.

Creating an AI Description

While first testing the analysis & summarization of Social Media posts I noticed something. Some points were just general facts about the company that didn't really make sense to

include in bullish/bearish points since those are more helpful to visualize current theses on a stock. These were however nonetheless important facts about the company so I wanted to incorporate them somehow. That's why I got the idea to do an AI generated general description of the company. This will include some important facts about the stock so that the user quickly knows what the company does and where it's value is. I'll use yahoo finance to get some key information about the company and provide a gpt-4o model with that info and additionally let it use the web.

Filtering out Points

A critical part of the analysis involves ensuring that the thesis points extracted from different financial posts are unique and not duplicates. Duplicate points—essentially similar arguments or statements worded differently across multiple posts—could clutter the database and obscure meaningful insights. Thus, it's necessary to find an efficient and accurate method to detect and filter out such duplicates.

Understanding the Problem

Initially, my intuitive approach was to compare text strings directly. However, this method quickly proved inadequate because it fails when points convey the same idea but use different wording. For example:

- "EPS grew by 5% year-over-year"
- "Earnings increased slightly"

Both sentences essentially indicate earnings growth, yet a direct string comparison wouldn't recognize them as duplicates. Therefore, a method capable of understanding semantic meaning is needed.

Introducing Embeddings

To overcome this limitation, I turned to text embeddings. **Text embeddings** represent textual information as high-dimensional numerical vectors. To generate these vectors, neural network models (trained on vast amounts of text data) learn contextual associations and semantic relationships between words and phrases. After extensive training, the model can translate any given sentence into a numeric vector. With the critical property that sentences with similar meanings will have vectors positioned closer together in the embedding space.

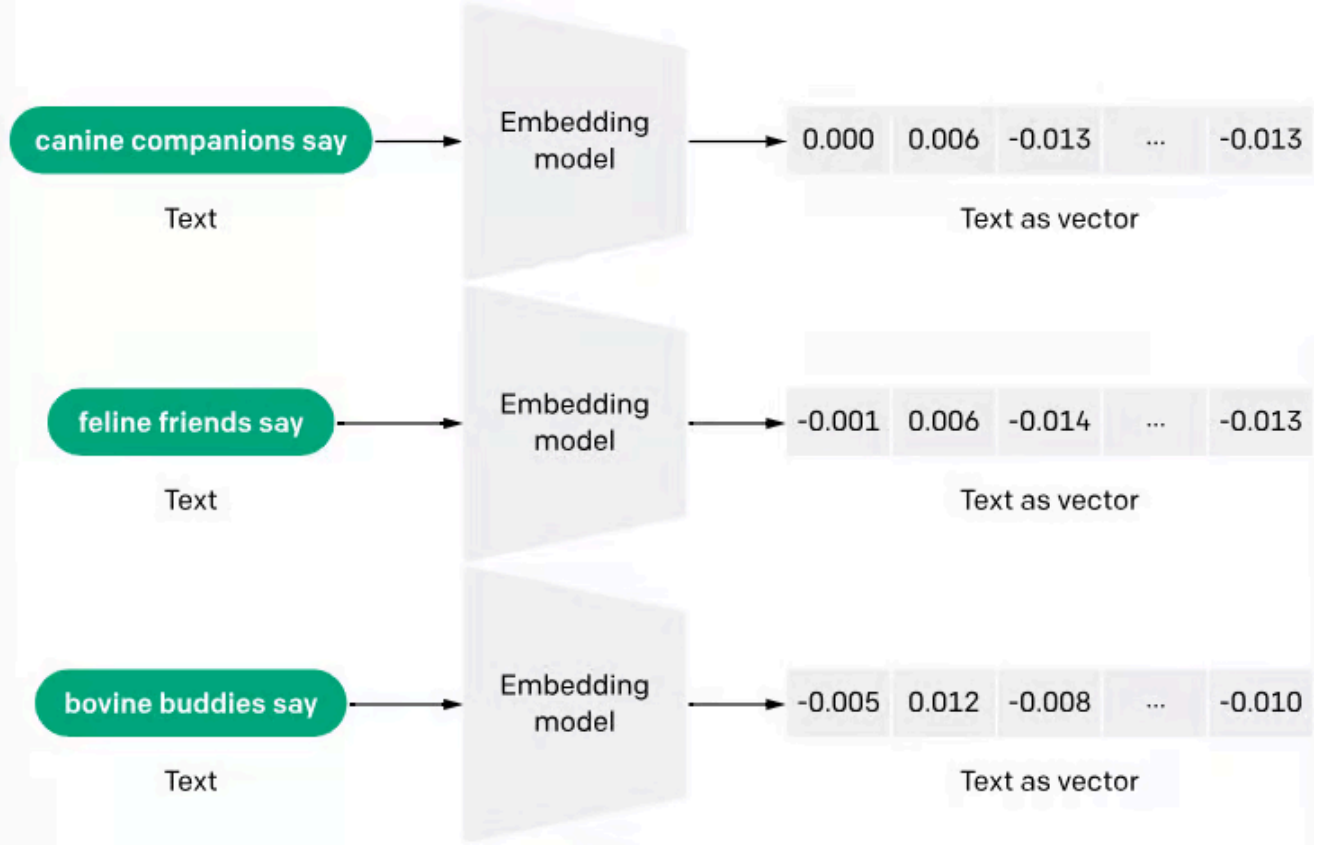


***Image 3.1** This visual shows what the output of an embedding model looks like. If you look at the different weights of the embedding (the vector) you can see that the numbers are relatively close together (0.000, -0.001, -0.005) because the meaning of the three phrases is similar.*

For example, take these example sentences:

- "Revenue increased by 10% in Q2"
- "Sales grew 10% in the second quarter"

They would generate embedding vectors located close together due to their semantic similarity. Visualizing how this "clustering" might look in a two dimensional space makes this concept pretty clear.

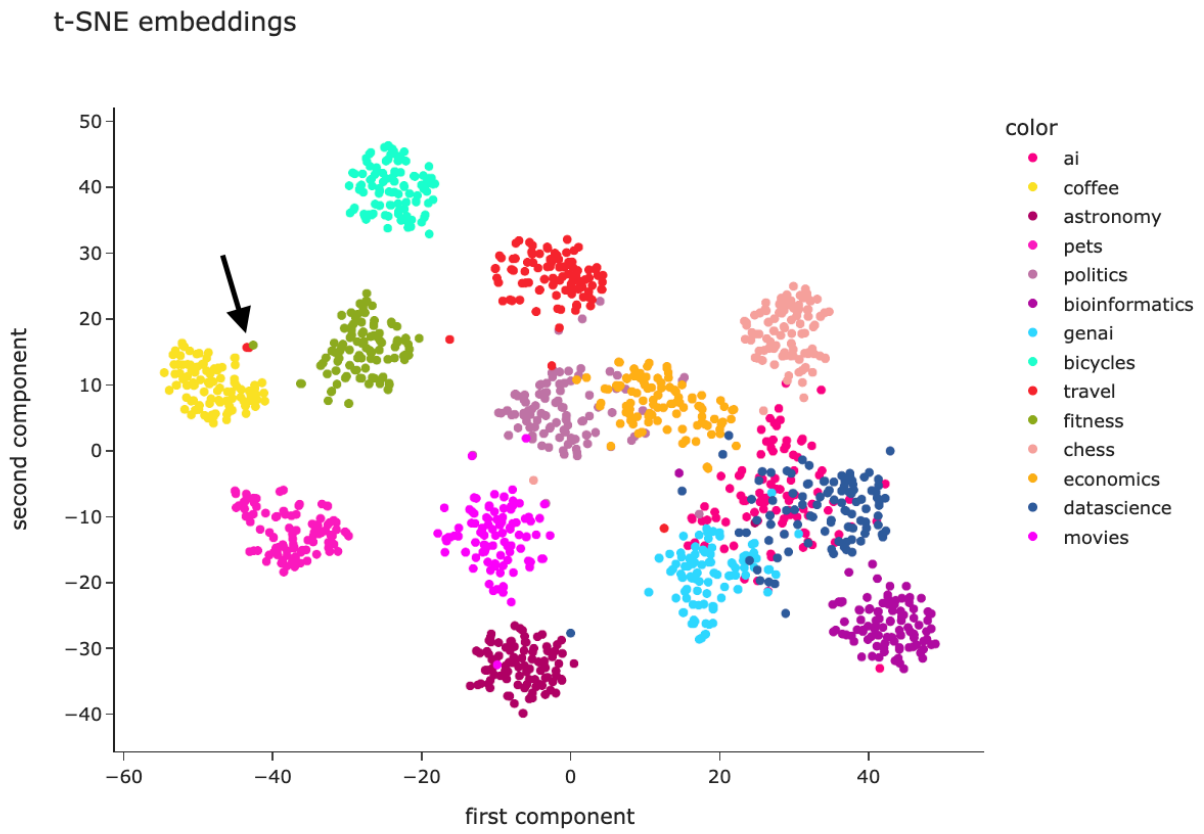


***Image 3.2** Without even going into the more specific individual points it's apparent here that similar topics are spatially close to each other. Politics and economics for example are very close together and even overlap at times while coffee and bioinformatics are further away from each other.*

Cosine Similarity Explained

To quantify how similar two embedding vectors (and therefore two sentences) are, I used **cosine similarity**. Cosine similarity measures the cosine of the angle between two vectors. Defined mathematically as:

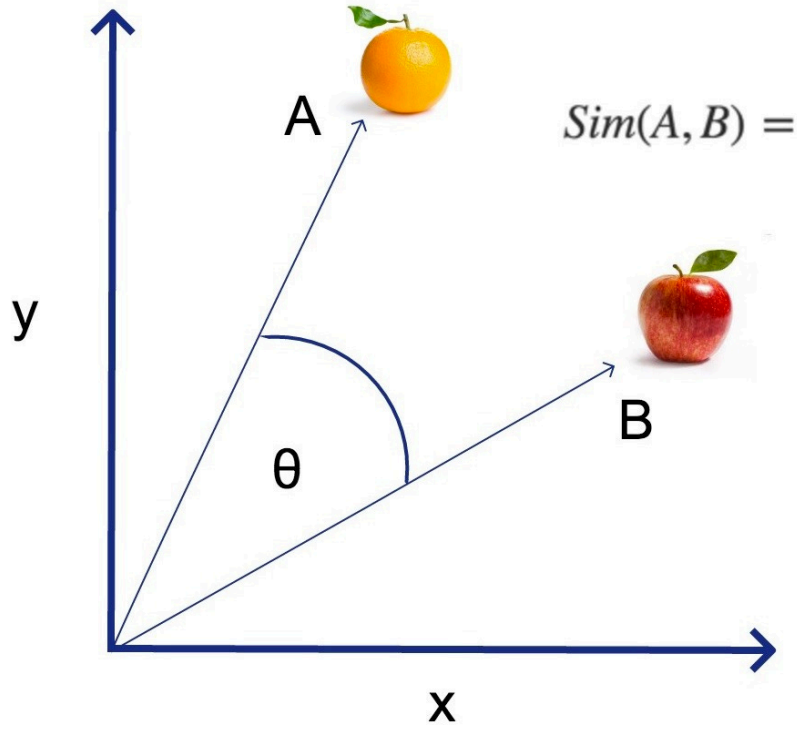
$$\text{Cosine Similarity} = \frac{A \cdot B}{\|A\| \|B\|}$$

Where:

- $A \cdot B$ denotes the dot product (multiplying corresponding elements and summing).
- $\|A\|$ and $\|B\|$ are the magnitudes (lengths) of vectors A and B.

This is pretty self explanatory when looking at the vectors as, well, vectors. Remember that vectors represent directions in space (one can kind of think of them as arrows). Which means if the angle between these two directions is calculated one can quantify how similar they are based on whether they point in the same direction or not.

Cosine Similarity



$$Sim(A,B) = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}$$

***Image 3.3** By calculating the angle you can see if these two vectors point in the same direction or not.**

The cosine similarity value ranges from -1 (completely opposite) to 1 (identical). A value close to 1 indicates highly similar meaning, whereas values closer to 0 imply unrelated content.

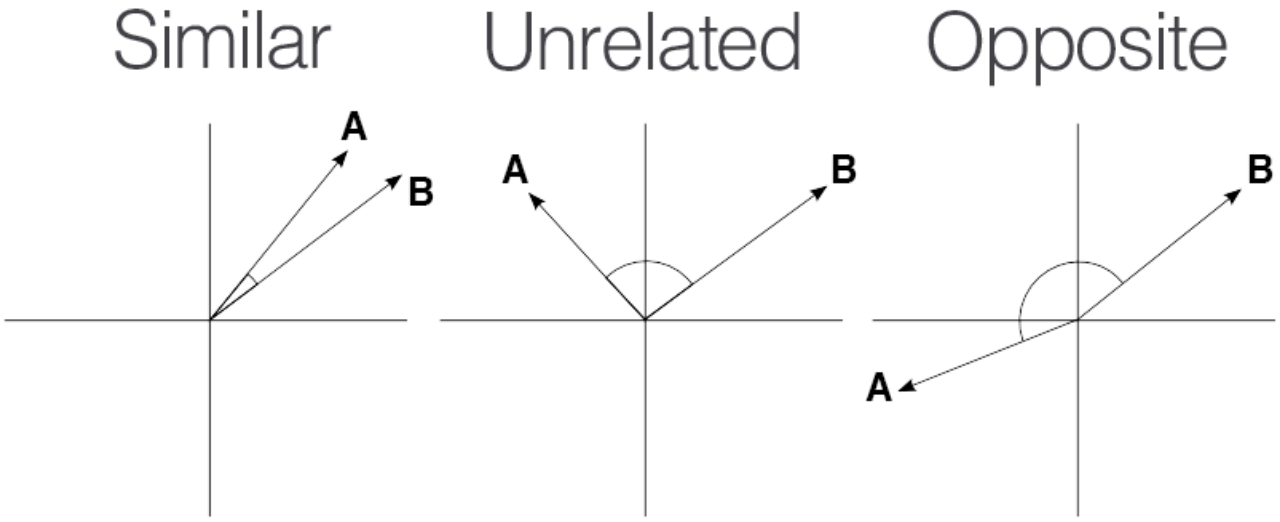


***Image 3.4** Looking at this in a spacial representation this just means the result is going to be 1 if the vectors are perfectly parallel, 0 if they share a 90 degree angle and -1 if they point in the complete opposite direction.**

Evaluating Embedding Models

There is a wide range of embedding models out there so I need to test which one is best for our use case. I then also need to find out the optimal threshold. As I just explained I'm going to be finding duplicates using cosine similarities. And as I've outlined these similarities are values from -1 to 1. This means I'm going to have to decide from which value on I classify a similarity match as a duplicate.

To determine these two factors, I tested three models:

- **OpenAI's text-embedding-ada-002:** A widely used model known for its strong semantic understanding.
- **OpenAI's text-embedding-3-small:** A more recent and cost-effective model offering promising performance.
- **SentenceTransformer (all-MiniLM-L6-v2):** A free, locally-run embedding model often used as a base line for local embedding model.

I wrote a script to conduct automated tests on an expanded dataset consisting of 40 chosen pairs of thesis points, each pair manually labeled as either duplicate (1) or non-duplicate (0). Evaluation metrics included:

- **Accuracy:** Percentage of correct predictions overall.
- **Precision:** Proportion of true duplicate points among all points identified as duplicates.
- **Recall:** Proportion of true duplicates correctly identified out of all actual duplicate pairs.
- **F1 Score:** Harmonic mean of precision and recall, offering a balanced evaluation metric.

Below are the detailed results of these tests at their optimal thresholds:

Model	Avg. Similarity (Duplicates)	Avg. Similarity (Non-Duplicates)	Optimal Threshold	Accuracy	Precision
text-embedding-ada-002	0.926	0.795	0.85	0.950	1.000
text-embedding-3-small	0.737	0.329	0.60	0.950	1.000
SentenceTransformer MiniLM	0.661	0.216	0.40	1.000	1.000

Model-by-Model Analysis

1. OpenAI text-embedding-ada-002

- **Average Similarity:**
  - Duplicate pairs: **0.934**

    - Non-duplicate pairs: **0.794**
- **Observations:**
    - The high similarity scores for both duplicate and non-duplicate pairs indicate that ada-002 produces a very compressed similarity range.
    - A very high threshold (around 0.85–0.90) is required to start differentiating duplicates, as shown by the jump in accuracy (up to 0.975–0.950) and F1 scores (0.978–0.955) at those thresholds.
    - However, this narrow margin makes the model highly sensitive to small changes in input; a slight variation might easily push a non-duplicate over the threshold.
- **Implication:**
    - While ada-002 can perform well when the threshold is set very high, its limited separation between duplicates and non-duplicates makes it less robust for the application.

2. OpenAI text-embedding-3-small

- **Average Similarity:**
    - Duplicate pairs: **0.757**
    - Non-duplicate pairs: **0.306**
- **Observations:**
    - This model provides a robust gap of about 0.451 between duplicates and non-duplicates.
    - Evaluation metrics steadily improve with increasing thresholds, reaching high F1 scores (up to 0.978 at a threshold of 0.60) and maintaining good precision and recall over a relatively wide threshold range (from about 0.45 to 0.60).
- **Implication:**
    - The 3-small model shows reliable and stable performance across a moderate range of thresholds, which is beneficial in production environments where inputs may vary. Its clear separation suggests it can serve as a robust pre-filter.

3. SentenceTransformer (all-MiniLM-L6-v2) – MiniLM

- **Average Similarity:**
    - Duplicate pairs: **0.698**
    - Non-duplicate pairs: **0.195**
- **Observations:**
    - MiniLM yields a strong gap of approximately 0.503 between duplicate and non-duplicate pairs.
    - It achieves perfect classification (accuracy, precision, recall, and F1 all equal to 1.000) at a threshold of **0.40**.
    - However, the performance is highly sensitive-if the threshold shifts even slightly above 0.40, the F1 score drops (e.g., 0.978 at 0.45, then lower at higher

thresholds).

- **Implication:**
  - Despite its sensitivity to threshold variations, MiniLM's outstanding performance in controlled testing indicates its high potential for accurate duplicate detection. I later tested this model in comparison with FinLang on some real world data and MiniLM performed a lot better (although it is crucial to note i tested on a very small dataset). Overall I'll choose MiniLM for now but will continue monitoring performance after deployment to make sure it outperforms FinLang.

4. FinLang (Investopedia-trained model)

- **Average Similarity:**
  - Duplicate pairs: **0.783**
  - Non-duplicate pairs: **0.292**
- **Observations:**
  - FinLang produces a gap of about 0.491 between duplicate and non-duplicate pairs.
  - Its evaluation metrics peak at thresholds between **0.40 and 0.45**, with accuracy around 0.95, near-perfect precision (up to 1.000), recall close to 0.957, and F1 scores around 0.958.
  - Performance remains stable over this moderate threshold range.
- **Implication:**
  - FinLang is specifically tuned on financial data, which is a significant advantage for this use case. It shows excellent performance at a moderate threshold, making it robust and reliable for filtering duplicates in financial texts.

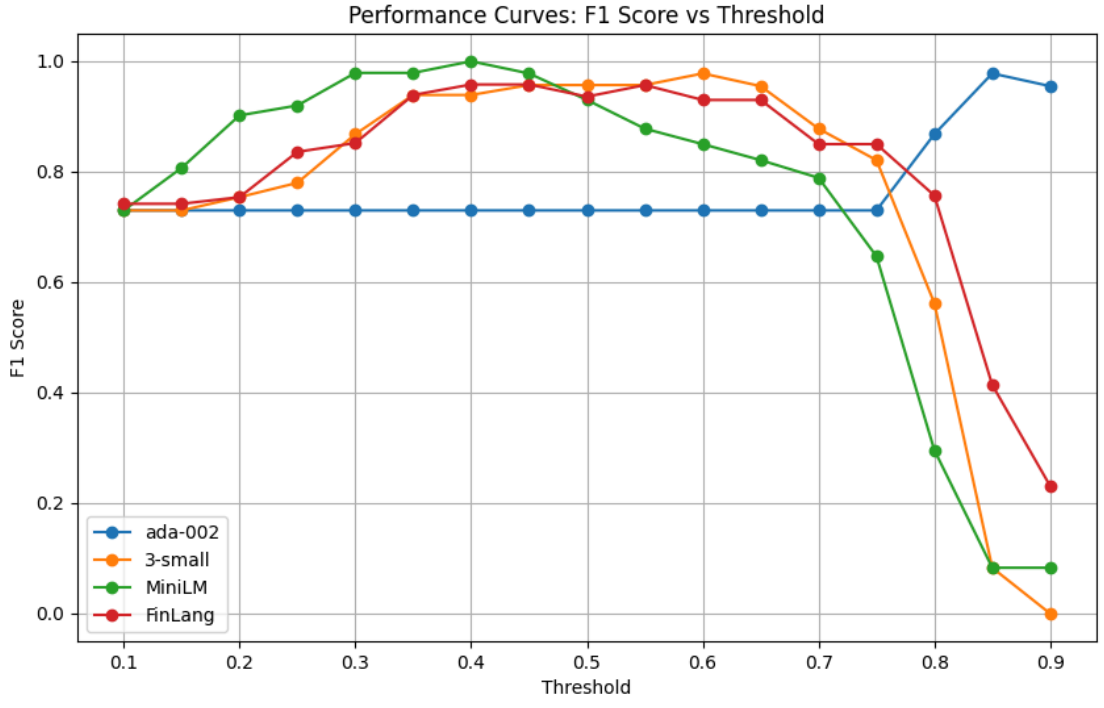


***Image 3.5:** I generated a graph using Matplotlib further illustrating the issue with MiniLM.

This isn't apparent when just looking at the data but in the graph it becomes clear that MiniLM's performance rapidly declines when shifting the threshold. This is an issue because it shows that by proxy I'm risking a lot of misclassification when I start handling more diversed data. (Also the graph shows, in a very apparent way, how terrible ada-002 is for this use case).*

Conclusion

For the duplicate filtering workflow, I'm picking MiniLM over FinLang primarily because FinLang tends to undershoot—often misclassifying valid points as non-duplicates, which negatively affects **precision**. As a result of the below outlined hybrid approach it's sensitive to prioritize catching all possible duplicates—even if it means initially flagging some false positives—over the risk of missing valid duplicate points. MiniLM's ability to robustly flag potential duplicates makes it the preferred model for this application.

Hybrid Approach

Based on these findings, I decided to implement a **hybrid approach**, utilizing Finlang for initial filtering and GPT-3.5-turbo for secondary verification.
This hybrid strategy consists of:

1. **Embedding-based pre-filtering:** Compute embeddings for all points and calculate cosine similarities. Points scoring above approximately 0.60 similarity are flagged as potential duplicates.
2. **GPT-based verification:** Points identified as potential duplicates undergo further evaluation by GPT-3.5-turbo in batches to confirm whether they truly represent identical semantic meaning.

This approach ensures efficiency and cost-effectiveness, combining quick and inexpensive embedding calculations with GPT's nuanced semantic evaluation.

I am going to store the vector embeddings for points I save to the database (which aren't duplicates) as a database Column under the individual Points so I don't have to recompute them when comparing them the next time.

Calculating ticker sentiment

Now that I have the points I can also calculate the ticker sentiment by just grabbing the average of all points in the database. This will be saved to the DB as well.

Extracting Criticisms

Expanding the Database

Comments on platforms like reddit or Seeking Alpha are there for a reason. In this case they are mainly for expressing criticism or approval of the given post. This gives us an additional interesting data point to use. Since comments contain criticism of the investment points in the post I can cross reference them with the points and thereby perform what is essentially peer review on the theses. That is the reason why I specifically extracted the comments earlier while scraping the posts.

In the previous steps what the script did was to already save the posts to the database and then fetch the posts content from there to extract theses and filter them. This is the most straightforward and least error prone way of doing things because it means I prevent any collision or discrepancy between the extracted points an the posts. Therefore it makes sense to do the same thing for the comments. Just retrieve the comments from the newly scraped posts from the database. However when I first built the database I did not know yet that this was going to be the approach which is why I didn't built eh database structure to hold the comments. So that's why it is at this point that I added a `Comment` table to the database. the details of this table are already explained at the top of the documentation in Database Structure. Essentially this table holds content of the comment, the URL and the author. In case you're wondering why the author is nullable, right now I'm not scraping the authors from seeking alpha but just from reddit. I'm also not going to be using the author data point, at least not in the scope of this documentation. Later on (after the BLL) I plan to add an additional feature that tracks the performance of given users. It will simulate trades based off the authors theses and record their performacne to then build a ranking. But again I'm not going to be doing this here just yet, I just thought I should explain so you wouldn't think I'm adding unnecessary columns.

The `Comment` table is linked to post and criticisms which means a post can have multiple comments and a comment can have multiple criticisms (exclusively in the described one-to-many relationship).

Fetching & analyzing comments

Now let's get to the `extract_criticisms.py` module. Since I plan to extract criticisms post for post, I'm again going to use asyncio and AsyncOpenAI to run this concurrently.

I need to do this post by post because each post will have it's own comments, therefore I want to process each posts points with it's respective comments. Since right now the points are just in an unstructured list I'll first have to group them by their post id's. For that I just use `defaultdict` and provide a list as the default factory function so that any non-present accessed key will be set to an empty list. This way it#s possible to simply loop through the points and append each point to the respective post_id's key in the dictionary. So if the post id is `1` I'll add it to the dictionary's `1` key and so on.

Now that I have a dictionary of lists of arrays structured by the posts they were extracted from I can asynchronously pass these dictionary entries to the extraction function.

I'll now first fetch all comments for the post id from the database. I then structure them into a list of dictionaries holding the comment id and the comment content.
I'll be using GPT again to compare the comments/criticisms to the points, however I do not want to pass the entire point data to it. This includes stuff like the vector embeddings of the points, which would massively increase the input tokens while not enhancing the process at all. Therefore I will reduce the point data to just the point content and the sentiment score (which helps add context).
Then I'll craft a system prompt instructing GPT to check if valid criticisms are found in the comments. If he finds no valid criticisms he should keep the point and set the output boolean of `criticism_exists` to `false`. If multiple strong criticisms exist or a criticisms is so strong it invalidates the point it should leave out the point in the final output. If there is criticism but the point remains viable it should link a short summarized version of the criticism to the point, set `criticism_exists` to true and assign a `validity_score` from 1-100 to the criticism depending on how strong it is.
I then provide it with ticker, points and comments (I give it the ticker so it can use the web to validate any criticisms found). Lastly I again provide a JSON schema so the program flow can easily process the data.

Now since I gave GPT a simplified version of the points I now have to merge this simplified version with the original full points and then I can return these merged points.

Note: One small thing worth mentioning is that since I'm running all of this asynchronously I don't want the whole process to fail just because one point failed. Therefore I'll tell asyncio to return exceptions. Then in the main running module I'll filter out all exceptions from the output and log them, just returning the results that did not fail.

Saving everything to the Database & returning data to user

Now I just commit the finished points & criticisms to the database and then save the current date & time to the `last_analyzed` column so I can log when the ticker was last analyzed. Now all relevant data is either in the database or can be fetched using the `get_stock_profile` function. Now I just need to provide an endpoint to in turn provide the frontend with said data. The script structures the data as follows and returns it through the `/retrieve-analysis` endpoint:

```
{
    "company": {

        "ticker": "aapl",

        "title": "Apple Inc.",
```

```
        "description": "Apple Inc. is a leading company in the technology sector...",
        "sentimentScore": 48,
        "logo": "https://images.financialmodelingprep.com/symbol/AAPL.png",
        "website": "https://www.apple.com",
        "price": 188.38,
        "exchangeShortName": "NASDAQ",
        "mktCap": 2829863198000,
        "industry": "Consumer Electronics",
        "earningsCallDate": "2025-04-29 22:00 CEST",
        "analystRating": "buy",
        "forwardPE": 22.669073,
        "dcf": 163.99684034053524,
        "beta": 1.259
    },
    "points": [
        {
            "content": "Shares have increased by about 5% this year.",
            "sentimentScore": 55,
            "postUrl": "https://www.reddit.com/r/investing/comments/1d8zcp3/nvidia_briefly_passes_apple_as_second_most/",
            "postTitle": "Nvidia briefly passes Apple as second most valuable public U.S. company",
            "postAuthor": "StatQuants",
            "postSource": "reddit",
            "postDate": "2024-06-05T00:00:00",
```

```
            "criticismExists": false,
            "criticisms": [
                {
                    "content": "Large firms likely use lobbying...",
                    "validityScore": 70,
                    "commentUrl":
"https://www.reddit.com/r/investing/comments/...
                }
            ]
        },
        ...
```


Note that the `/retrieve-analysis` endpoint takes a second parameter aside from the ticker, that being timezone. Since I'm returning the earnings call date and time I take the timezone as an argument and localize the time returned by thy yahoo finance API before returning it.

CORS Middleware & Origin Handling

To facilitate secure and seamless communication between the React frontend and the FastAPI backend, I implemented Cross-Origin Resource Sharing (CORS) middleware which is provided by FastAPI. This middleware allows controlled cross-origin requests, which are essential for enabling the frontend—served from different origins (e.g., local development servers at `http://localhost:5173` and `http://127.0.0.1:5173`, or the production URL set via the environment variable `FRONTEND_URL`)—to communicate with the backend without security conflicts. Allowed origins are loaded from environment variables, ensuring flexibility across both development and deployment environments.

And we're done with the backend!

Sources

[1] *Mentioned Academic Paper - "A_Practical_q-Gram_Index_for_Text_Retrieval_Allowing_Errors": https://www.researchgate.net/publication/2747370_A_Practical_q-Gram_Index_for_Text_Retrieval_Allowing_Errors

Image Sources

Image 1.1: Local screenshot
Image 1.2: Local screenshot

Image 2.1: https://medium.com/@kuldeepkumawat195/hashing-in-python-sets-and-dictionaries-aa2fdbb3861f

Image 3.1: https://www.datacamp.com/blog/what-is-text-embedding-ai
Image 3.2: https://medium.com/data-science/text-embeddings-comprehensive-guide-afd97fce8fb5
Image 3.3: https://businessanalytics.substack.com/p/cosine-similarity-explained
Image 3.4: https://medium.com/geekculture/cosine-similarity-and-cosine-distance-48eed889a5c4
Image 3.5: Local Screenshot of graph generated with `~testing_scripts/visualize_embedding_models.py`

Frontend

Design

Every good frontend starts with design. I personally believe to create a good frontend you first have to spend time designing it. After you have a proper design you can start writing the code. therefore the first step is creating the design in Figma.

For that it's important to look at what I want to display in the first place. I essentially want a dashboard that displays everything the backend returns.
Since the emphasis in this documentation lies on the coding aspect I won't go to deep into the designing process but using Figma specifically is helpful because it's built for developers and it's therefore very easy to later convert the design into code.

Anyways after putting some time into it I ended up with the following design:

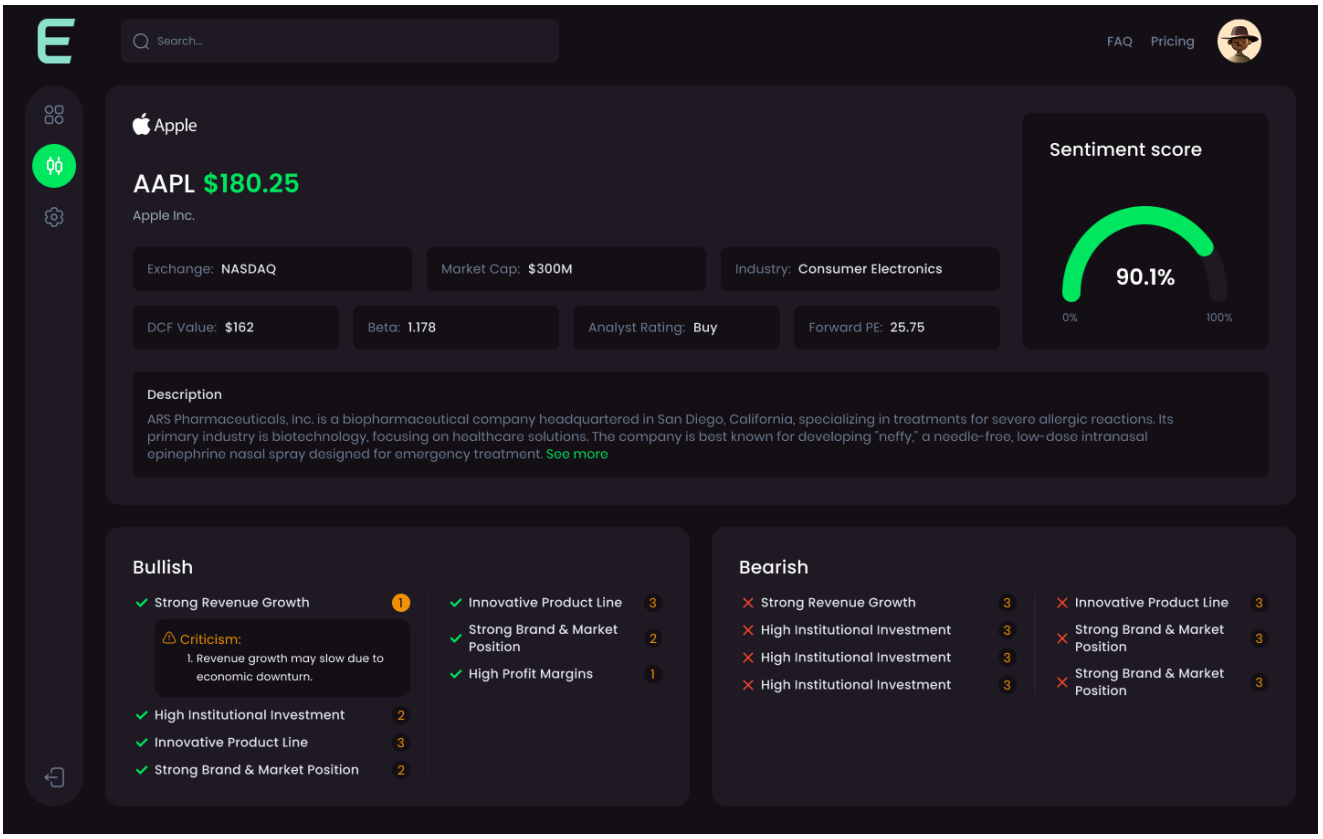


Image 4.1

Note that not every component is necessarily going to be in the end product. For example I don't really intend to add Account management for now but I'd rather have too much and just strip it away when coding it later than the other way around. (I also added a logo at the top).

The Program Flow

For the Frontend I'll use React, Vite and Tailwind. The overall program flow is supposed to be split into three pages.
The first will be the "Home Page" with the search bar in which the user can enter the stock he wants to analyze. After he enters his desired ticker said page will redirect to the route for the second page, the "Stock Page" which is the one I just designed. More specifically it will redirect to `/stock/{ticker}` with ticker being the entered ticker. The app will then send an API request for the given ticker and retrieve the response. From there I'll use the response to display the data. Lastly I'll add a favorites page with the tickers the user has starred.

Stock Page

Since the Analysis Page or "Stock Page" is the one I actually designed in Figma and therefore the one I have the best idea of, I'll start here.

Stock Page Route

Since I'm using Vite I don't have access to Next.js' convenient navigation function to handle routes so react-router-dom will have to suffice. I'll add a new dynamic route `/stock/:ticker` linked to the Stock Page element.

In the stock page I first have to get all ticker data from the backend. I use axios instead of fetch to send out requests because axios is superior to fetch in pretty much every way. It provides automatic JSON parsing, better error handling, interceptors, simplified syntax, upload/download progress, etc. The list goes on but those are the main reasons why axios is my go to option for sending HTTP requests in JavaScript, at least when It's in a project and not just a simple script. One of the biggest advantages axios gives us in a full stack application like this one is allowing us to set a base URL which more often than not is a much cleaner approach then setting a vite proxy. This way I won't have to provide the backend URL every time I make a request but can just do it once in the axiosinstance.js and then just use relative URLs.

When sending requests with axios I also use `AbortController`. This is used to manage and cancel ongoing API requests if the component unmounts or the ticker changes before the request could be completed. I do this because when a component like `StockPage` fetches data in a `useEffect`, there's a chance the user navigates away (unmounting the component) or changes the URL ticker (triggering a new request) before the previous request is completed.

Without `AbortController`, the asynchronous request might complete and attempt to update state on an unmounted component.

After getting the data from the API I have to pass it on. I'll pass the company object down as props to the `StockInfo` component which will hold all of the info for the stock (in the Figma design that's the upper part of the dashboard). Also in the design I want the description to be expandable. Since this is part of the `StockPage` component and might affect spacing of other components returned by it it makes sense to lift the state up from `StockInfo`. So I'll pass down state and toggle function of `isDescriptionExpanded` down too.

I'll also want to render the list of Points below the `StockInfo`. Since the two lists (bullish/bearish) should look and behave identically except for the green checks & red crosses I'll just do one `PointsList` Component and pass down if points are bullish or bearish to render them conditionally. To do that I'll just check if the points sentiment score is greater or less than 50. To avoid re-doing this possibly expensive (depending on length of lists) operation on every render I'll use `useMemo` to cache the result of the filtering until the points change.

Lastly I'll add a loader and error component to display loading and error states and the blue frame for the stock page is finished.

Let's now get to the individual components.

StockInfo

Format data

First I have to format all the data I get passed down so I can display it nicely. I format market cap, earnings date, and analyst rating appropriately. The formatting is mostly basic regex and simple conditionals so I won't go into too much detail here. I'll also determine the color of the DCF, analyst rating, forward PE, and beta based on their value to give the values a bit more meaning. For example if the Analyst rating is "Strong Sell" or "Sell" that's a bad sign and so it will be assigned the color red, "Hold" and it will be yellow, "Buy" or "Strong Buy" and it will be green.

I first engulf the entire component in a section with the standard `bg-widget-background` (This is a common theme for pretty much the entire site as you can see in the design).

Top Section

I'll then first do the company logo followed by the company ticker which, when clicked, will link to the website. I do a hover animation using standard `transition-colors` from white to green to show the user he can click the ticker.

Next to the ticker I'll put the price of the stock and below I'll add the title.

I mentioned a favorites page and this is the first step of implementing it.
I added a star button next to the ticker price along with a simple animation for clicking it. When clicked it will first check if local storage already contains the ticker as a favorite. If it does not it will save the ticker to an array in local storage called `favorites`. If the entry exists already it removes it. In both cases it naturally also updates state to render the star as either filled or just outline.
I'll go into more detail regarding the saving of the favorites to Local Storage in Favorites.

Metrics

Now for the individual information points at the bottom. I created a separate `MetricCard` component which takes value and styling in as props. I then split the components into two rows. I wrapped both of them into one self stretching div. I then wrapped each one in it's own flex container, adding a bit of top margin to the bottom one.

Sentiment Score

For the tickers sentiment score component I want a half circle which engulfs the sentiment score. Both the score and the half circle should change colors from red to orange to green based on the score with the half circle acting as a bar to visualize the score.

I'll take the score as a prop, then I validate that the score is in the expected format. Next I get the score color by setting it to red if it's under 40, orange if it's between 40 and 60 and green if it's greater than 60.

Now I need to construct the half circle. I first tried to import the svg i generated with figma but that turned out to be a suboptimal approach since it didn't work well when trying to make it responsive. Since I'm not an svg professional and this is pretty much as far as my design skills take me I let ChatGPT generate the svg for this. So it's important to note that the dynamic svg was not generated by me. Lastly I just add 0% and 100% labels under the respective ends of the half circle.

Description

If the description is expanded I'll simply display the entire description. If it's not I'll only display the first 500 characters followed by three dots. I'll then append a "See More" or "See Less" button, again depending on if it's expanded or not. As `onClick` I can just set the passed in toggle function for the description.

PointsList

First I'll set the icon to either the green check mark or the red cross depending on the type prop that's passed down. Both icons are exposed in the public folder.
In the design file I stacked the points into two columns next to each other for both lists. Essentially first comes a point on the left then one on the right and the next point is left again but one line below. The easiest way to implement this is to actually split the points up in two columns and rendering them separately. For that I'll use `useMemo` again so I only perform this filtering more than one time if the points change. If the point's index is even I'll push it to the left side, if it's odd I'll place it on the right side. I'll then render a container for the left and the right side column, mapping the point values to a `PointItem` component each.

PointItem

`PointItem` is the individual point item. This component specifically is very interesting because I'll have to implement two pretty challenging features aside from just the rendering of the point.

1. When clicked the Point should open a `PointPreview`. The point preview isn't included in the Figma design File. I essentially want to open a preview that pops up below the point. Similar to the criticisms card (which in turn is in the Figma design). It should show basic data of the post the point is extracted from. Title, author, date of post and of course a link to the original post. This way the user can check exactly where the point came from.
2. Similarly it should display a red bubble next to the point which shows how many criticisms the point has. When clicked it should expand a criticism card displaying all criticisms of the post, their validity score and a link to the comment the criticism was extracted from.

Since it's now relevant to keep track of which of the list items is currently selected to be expanded so it doesn't expand multiple at once, I went back and added a `selectedPointIndex` state to the `PointsList` component. Additionally I added a

`handlePointClick` function which, given an index, sets the `selectedPointIndex` state to be the passed index. I then passed the index of the point, a unique key for the point, the `isSelected` boolean and the `handlePointClick` function all down as props to the `PointItem` components.
The unique key is derived by adding up the original index with the points post URL. Unique keys with dynamic list items like these are important in React because they allow it to identify each item in the list which in turn enables it to track identity between renders. This way React can track, update and re-render only the components that changed instead of resetting everything.
The `isSelected` boolean simply checks if the `selectedPointIndex` is equal to given points index.
The idea of passing `handlePointCLick` down is that `PointItem` will have a way of notifying it's parent component to toggle the selection state for the given item's index.

I'll also use `handlePointClick` for the criticism list expansion. Since bot the post preview and the criticism list will be in the exact same place I'll have to make sure they aren't both expanded at the same time for the same point. I can do this by simply setting `onPointClick` to and index of -1 (no index selected) if the criticism list is opened which will close any open post previews.

When I first implemented this I had an issue. Whenever I was clicking the criticism bubble to open the criticisms the point preview would open. I remembered that I had a similar issue some time ago on another project. After some googling I found out this was because I had a button in a button. The criticism bubble button is inside the bigger button to toggle the post preview. In JavaScript when an event (like `click`) happens the event "bubbles up" through it's parent elements. Now I don't technically have a button in a button. To be exact it's a button in a div. But the div is clickable. Meaning the `click` event bubbles up to the div and triggers it's `onClick` and opening the post preview. I fixed the issue by putting `event.stopPropagation()` at the stat of the `handleCriticismToggle` function.

For the criticism button styling I set it to be a filled dark circle with an orange number when not expanded and an orange filled circle with a dark number when expanded.
Lastly also added some accessibility features to the divs/buttons like an `onKeyDown` handler and `tabIndex`.

CriticismList

Both the `CriticismList` and the `PointPreview` are of course rendered conditionally depending on if they're expanded or not.
For the criticism list I'll display an orange Warning icon followed by a "Criticism:" header (also in orange). Each criticism will consist of the criticism itself followed by a link icon to indicate that the comment of the criticism is embedded as an anchor tag into the criticism text meaning the user can click it to open the comment in a new tab. Then it'll display the validity score below. I wanted the validity score to also be colored depending on it's value so if it's

larger or equal to 75 it will be green, if it's between 50 and 75 it will be yellow and if it's under 50 it will be red.

PointPreview

I want to give the user the possibility of checking where Points came from and validating the source for themselves. I could just provide a link to the post however I want to add an embed that pops up when clicking on the Point which displays some key information about the post and of course includes a link.

At the top of this expendable card I'll render the title of the reddit post as a header. Below it I'll put the source of the post (reddit/seekingalpha), the author, and the formatted date on which the post was posted. Below all of this I'll then put an anchor tag to the original post.

Animation

Now `CriticismList` and `PointPreview` are done. However right now when the user clicks on either the div or the criticism button the card just appears and everything moves down. This looks everything but smooth so I wanted to add an animation to make the card slowly open up vertically.

What I wanted to do here was a bit too complex for traditional CSS transitions so I had to use `framer-motion`'s `AnimatePresence`. Normally when a component gets removed from DOM (in this case setting `isCriticismExpanded` to false) React just removes it. `AnimatePresence` waits for the exit animation before removing it.

To design the animation, `AnimatePresence`, or more specifically `motion.div`, (The element that makes a regular div animatable) requires some arguments. I set `initial` s (the initial state of the div) opacity, margin and height to 0 (since it has to expand), `animate`'s height to auto, opacity to 1 and marginTop to 0.5rem to display it and `exit`'s values to 0, 0, 0 to hide it again when collapsing. Lastly I applied overflow:hidden to make sure the content of the container doesn't spill out during the animation and applied the div to `CriticismList` and `PointPreview`.

Loader

Since the component fetches the data from the API every time it loads there is always one or two seconds of loading time. For this time I wanted to display a loader on the screen so the user doesn't think the page got stuck. For this I'll use the Classic #39 loader from [css-loaders.com](). I think it fits the aesthetic of the app nicely and so I don't think there's a need for me to create my own loader. I just pasted the CSS into my Index.css file and associated it with a CSS class so it's easy to use across the board.

Error Message

I added a custom error message component which is just a red box with the given error message and an error icon.

In `StockPage` I then display said component if an error occurs. This includes handling some common cases like the server not responding.

StockLandingPage

I wanted to also provide a Landing page in case the user goes to `/stock/` without a ticker at the end. This page just tells the user that he should use the search bar in the header or go to the Home page to find and select a stock. I also added a link to `/` where the Home page is going to be implemented.

HomePage

The Home page is just a large version of the logo with a prompt below it to enter a stock ticker to access an analysis for that stock. It then displays a large version of the `SearchBar` component.

SearchBar

The search bar will be the main input component for the user. I'll include it on the HomePage as well as the Header.
It should send new requests to the `/stock-analysis` endpoint every time the user types into the search bar. It should then show the results of the query below the search bar. The user should then be able to click the suggestions generated. When clicked the search bar has to send a request with the ticker to `/check-analysis`. It should then show a modal with text and two buttons. The text should always be the message key from the API response. The buttons should differ based on the `existing_analysis` key. If it's true the message key will look something like this:
"There is an existing analysis for 'aapl' created on 2025-04-
06 17:51:38.025720. Do you want to access it or create a new analysis?"

There should be an "Access Existing" and "Create New". "Access Existing" should just navigate to `/stock/{ticker}` where {ticker} is the ticker the user clicked on. `Create New` should send a request to `/generate-analysis` said ticker.

If `existing_analysis` is false the message will look like this:
"goog was not found in the database. Do you want to generate a new analysis for it?

The two buttons should be "Yes"and "No" with yes sending a request to `/generate-analysis` and "No" just closing the modal.

Every time the search bar's content changes it will send an axios request to the `/stock-query` endpoint. This happens with a timeout of 300 ms so there's a buffer between the

requests. If the user clicks outside the bar at any time it will catch it using an event listener and hide the results.

Then comes the clicking of the search results. When clicked the component will send a request to `/check-analysis` as described. It will then look for the date in the message using regex and and format it appropriately if found. Then it will open the modal. Also since `SearchBar` re-renders a lot (every time the user changes the content) the `handleResultClick` is going to be created from scratch every time. To prevent that (so the performance doesn't suffer) I wrapped `handleResultClick` in a `useCallback` function. `useMemo` doesn't cut it here because if the individual results were optimized components, receiving a new function instance on every render would cause them to re-render unnecessarily.

`Modal` is it's own component. It's props include the basic setter functions again so it can return null when it's not open and render when it is. They also include functions to run when the modal buttons are clicked. This way all functionality from the `Modal` component is set and controlled by the `SearchBar` component. Additionally the entire modal component is wrapped in a div with `stopPropagation()` so it doesn't pass on any of the `click` events upwards.

`handleAccessExisting` navigates the user to `/stock/{ticker}`.

`handleGenerateAnalysis` starts the analysis process. It send's a request to `/generate-analysis` as specified above. Then it navigates to the AnalysisLoadingPage. Specifically: `/loading-analysis/{data.task_id}/{currentTicker}`

AnalysisLoadingPage

The analysis loading page is going to be the page displayed while an analysis is being generated by the backend. It will be the component polling the `/analysis-status` endpoint and updating the user on the status of the analysis generation.

I want to poll the endpoint every 3 seconds. This is done by utilizing `setInterval`. However if the app encounters an error or the progress returned by the API is 10 (meaning the analysis is completed) I want it to stop sending requests. Meaning I'll have to clear the `Timeout` object created by `setInterval()` to stop the interval execution of `checkStatus`. To do that I'll have to store a reference to said object so I can clear it later. The issue here is that if I store this in a standard variable inside the component it will not persist over re-renders since components are nothing else than functions that get re-run with every render, therefore re-initializing all variables contained in said functions. Therefore I used React's `useRef` hook to create a persisting reference to the `Timeout`.

Then I just implemented the same loader as before with an added progress bar underneath it. Said progress bar adjusts it's fill percentage depending on the current progress divided by 10 and multiplied by 100 (10 because there are 10 progress steps). This is followed by a status message which is just the status from the response.

Lastly I added a simple error display and "Go Back" button in case of there being an error message returned by the response.

Favorites

I personally like the idea of being able to "star" certain stocks and having a favorites list to which you can return to essentially check your watchlist. Therefore I'm going to implement that next.

First of all I need to save and load this favorites list. I don't plan on adding an account management system right now especially because none of the features would benefit from it aside from this one. So I'll take a simpler approach and save the list to local storage. This way the user's favorites will get stored locally in his browser. This saving is done by the star component in StockInfo > Top Section.

I created a separate route `/favorites`. This route then displays the `Favorites` component.

Favorites simply loads the `favorites` array from local storage and renders them as a list. For every element in the array it then renders a row container with the logo, ticker, title and sentiment score of the company. Said information is fetched from the `/retrieve-analysis` endpoint with the ticker from the array as the parameter. When an element is clicked it simply links to it's respective stock page URL. This is done through the `Link` component from `react-router-dom`. This way the content is just re-rendered and there's a smooth transition.

Global Components

Here I added some global components. Components that will be shown on multiple pages.

Header

As the last part of the App I added the header and side bar. The header is very simple. It's at the top of the page and is fixed so it scrolls down with the page. It contains the `SearchBar` component in the middle and the Episteme logo with clickable text to the left (the text linking to the main page). The side bar is not rendered when on the `HomePage` since then it'll be in the middle of the screen already.

SideBar

The side bar is used to travel through the 3 main routes: `/`, `/stock`, `/favorites`. All three have their own icon. The `selectedIndex` is determined based off the route the user is currently on. The selected index is then styled appropriately to have a green circle around it. The bar is rendered with `<aside/>` and fixed as well so it scrolls with the screen too. When one of the buttons is clicked the browser navigates to the associated route. Lastly the button clicking is also animated using `framer-motion`'s spring animation.

MainLayout

To make sure that the `Header` and `SideBar` are always rendered on all thee routes I added a `MainLayout` component which imports the two components and then uses `react-router-dom`'s `Outlet` component. This is combination with putting the `/`, `/stock`, `/favorites`, `/stock/:ticker` routes inside `MainLayout`'s `Route` element in `App` renders all it's child components while including padding and appropriate containers.

Sources

- Icons: All icons used are from Heroicons which are free to use under the MIT License. The only exception is the link icon next to the criticisms. That one I let ChatGPT generate because I couldn't find one on Heroicons I liked.
- `SentimentScore` dynamic slider: This was generated with ChatGPT since I needed it to create a custom SVG.
- To create the intial design I used Figma.

Image Sources

Image 4.1: Local Screenshot

Deployment

The backend of Episteme is deployed on a barebone Ubuntu 24.04 VPS from Hostinger, specifically chosen for flexibility and full control over server configuration. The VPS provides 2 CPUs, 8GB RAM, and 100GB of storage, offering optimal performance and scalability for our application needs. The frontend, however, is hosted separately on Vercel due to its lightweight nature, as it requires no dedicated storage, unlike the backend, which utilizes PostgreSQL databases. Initially, the server was securely configured by disabling password-based login and enforcing SSH key authentication exclusively, greatly reducing vulnerability to brute force attacks.

The Python FastAPI application is served using Gunicorn configured with 4 Uvicorn workers, providing efficient asynchronous handling of requests. The FastAPI service was configured via a systemd unit file to ensure automatic restarts, improved stability, and easier management. All necessary environment variables, including credentials, database URLs, and settings, are securely stored within a `.env` file located inside a dedicated `config` subdirectory. The absolute path to this `.env` file is explicitly defined via an `ENV_PATH` environment variables.

The two PostgreSQL databases are set up using tailored Python deployment scripts (`deploy_db.py` and `deploy_stock_index_db.py`). These scripts confirm database existence, create necessary tables, and enable essential extensions like `pg_trgm` for enhanced database performance (as outlined before in the Database section).

After initial deployment, it became evident that the in-memory task storage was unsuitable for a multi-worker Gunicorn setup, due to isolated worker memory spaces. Therefore, Redis was introduced to maintain shared state across Gunicorn workers. Redis was configured locally on the VPS, and helper functions were implemented to serialize task state to JSON for efficient storage and retrieval. A brief time-to-live (TTL) was added for Redis entries upon task completion to ensure efficient resource utilization and to provide a smooth frontend experience.

To achieve automated deployments reminiscent of cloud services such as Vercel, I configured a GitHub webhook alongside a dedicated FastAPI webhook listener endpoint. This setup automatically triggers a deployment script upon receiving GitHub push events, pulling the latest updates from the repository and restarting the application seamlessly. This automation ensures rapid iteration and minimal downtime, significantly improving the deployment workflow. Essentially it allows me to push changes for the backend to my GitHub repo from my personal PC and the changes will be integrated into the live backend in seconds.