ChatGPT

Building a Finance Q&A Agent with HyperCLOVA X and RAG Integration

Developing a financial question-answering agent involves setting up the coding environment, implementing the agent's logic (including a **Retrieval-Augmented Generation (RAG)** approach for better answers), and deploying the solution as a web service. We will break down the process into three parts: **1) Setup (Git, VS Code, yfinance, HyperCLOVA X)**, **2) Database & Agent Improvements (Advanced RAG and other techniques)**, and **3) Deployment**. Each part is explained step-by-step with code examples and clear workflow descriptions, assuming no prior development experience.

Part 1: Environment Setup (Git, VS Code, yfinance, HyperCLOVA X)

Setting up a proper development environment is the first step. We need to install the necessary tools and libraries, and configure access to data (stock prices) and the HyperCLOVA X language model.

Step 1: Install Git, VS Code, and Python

- **Git**: Git is a version control system used to track code changes. Install Git from the official website (git-scm.com) and configure it on your system. This allows you to clone the project repository and manage code versions. In Visual Studio Code (VS Code), ensure Git is recognized. For example, VS Code's documentation says: "To use Git and GitHub in VS Code, first make sure you have Git installed on your computer… Make sure to restart VS Code afterwards." [1] . After installing, you can use VS Code's Source Control panel to clone and manage repositories easily.

- **Visual Studio Code**: VS Code is a popular code editor. Install VS Code and open it. You can open a folder for your project or clone a repository directly in VS Code. If you have a Git repository URL (e.g. from GitHub), use **File > Clone Repository** in VS Code or run the **"Git: Clone"** command in the Command Palette [2] . This will copy the code to your local machine. If starting from scratch, create a new folder for your project and use **"Initialize Repository"** in VS Code's Source Control to start versioning [3] .

- **Python**: Install Python (version 3.x). You can download it from python.org or use a package manager. Verify the installation by running `python --version` in a terminal. It's recommended to set up a **virtual environment** for the project so that project libraries don't conflict with system libraries. For example, in your project folder:

  ```
  python -m venv venv      # create virtual environment
  source venv/bin/activate  # activate it (Linux/Mac)
  venv\Scripts\activate    # (Windows)
  ```

  This step is optional but good practice. Once activated, the next steps will install libraries into this environment.

Step 2: Install required Python libraries

Open a terminal in VS Code (View > Terminal) and install the necessary packages using `pip`. The key libraries we need are: - **yfinance** – to fetch stock market data from Yahoo Finance. - **pandas** – for data manipulation (yfinance returns data in pandas DataFrames). - **Flask** or **FastAPI** – to build the web service (API endpoint) for the agent. - **requests** – to call external APIs (we'll use it for calling the HyperCLOVA X API).

Use pip to install these:

```
pip install yfinance pandas flask requests
```

This will download the packages from PyPI. For example, yfinance is a Pythonic library to fetch financial market data from Yahoo Finance [4]. It's open-source and convenient for getting historical prices.

Step 3: Configure stock data access with yfinance

yfinance allows us to retrieve historical stock price and volume data easily. We will use this to get the data needed to answer finance questions. To test that yfinance works, try a quick Python snippet in VS Code (you can create a `test.py` file or use the interactive Python console):

```
import yfinance as yf

# Example: Download 1 year of daily data for Apple (AAPL)
data = yf.download("AAPL", start="2022-01-01", end="2022-12-31")
print(data.head())
```

When you run this, yfinance will fetch the daily prices for Apple between the dates specified. The `download()` function returns a pandas DataFrame containing columns like Open, High, Low, Close, Volume, etc., indexed by date [5]. For example, `yf.download("AAPL", start="2020-01-01", end="2021-01-01")` will get Apple's daily data for 2020 [6]. You should see the first few rows printed if successful. This confirms that yfinance can retrieve data.

> **Note:** Yahoo Finance's API (which yfinance uses) is intended for personal/research use [7]. For a competitive or heavy-use scenario, ensure you respect their terms or consider alternative data APIs if needed. If yfinance fails or is too slow for certain tasks (for instance, if we need many tickers at once), we might replace it with another data source or pre-download the data.

Step 4: Set up HyperCLOVA X API access

HyperCLOVA X is Naver's large language model (LLM) service. It's specialized for the Korean language and culture, with strong performance in multilingual tasks [8] [9]. We will use HyperCLOVA X via its **CLOVA Studio API** to handle natural language understanding and generation.

To use HyperCLOVA X: - **Get API Credentials**: You need an API key from Naver's CLOVA Studio. In the context of the Mirae Asset competition (from the PPT instructions), an API key was provided for evaluation. This is usually a long string (prefixed with something like `nv-...`). Keep this key secret. **Do not hardcode it in your GitHub repo** – instead, store it as an environment variable or in a config file that

is not committed. For example, on your development machine you can set an environment variable: - On Linux/macOS: `export NCP_CLOVASTUDIO_API_KEY="your_api_key_here"`. - On Windows: `set NCP_CLOVASTUDIO_API_KEY=your_api_key_here`.

- **Understand the API format**: The HyperCLOVA X API is a RESTful service. You make HTTP requests to Naver Cloud endpoints with your key. The API supports a **chat-completion** style interaction (similar to OpenAI's ChatGPT API). You will send a JSON payload containing a list of messages (with roles like "system", "user", and possibly "assistant") and get a generated answer. We'll use the simplest case: one user message (the question) and possibly a system message with instructions.

- **Make a test API call**: We can use Python's `requests` library. The base endpoint (as of the documentation) for chat completions is:

```
https://clovastudio.apigw.ntruss.com/testapp/v1/chat-completions/
{modelId}
```

where `{modelId}` might be something like `HCX-003` (an ID for HyperCLOVA X model). We include our API key in the header. For example:

```
import os, requests, uuid, json

api_key = os.getenv("NCP_CLOVASTUDIO_API_KEY")  # ensure your API key is set in env
url = "https://clovastudio.apigw.ntruss.com/testapp/v1/chat-completions/HCX-003"
headers = {
    "Content-Type": "application/json; charset=utf-8",
    "Authorization": f"Bearer {api_key}",
    "X-NCP-CLOVASTUDIO-REQUEST-ID": str(uuid.uuid4())  # unique request ID
}
prompt_question = "한국 증시에서 최근 한 달간 가장 많이 오른 종목은 무엇인가요?"
payload = {
    "messages": [
        {"role": "system", "content": "You are a financial assistant AI. Answer in Korean."},
        {"role": "user", "content": prompt_question}
    ],
    "temperature": 0.3,
    "maxTokens": 100
}
response = requests.post(url, headers=headers, json=payload)
result = response.json()
print(result)
```

In this example, we ask (in Korean) "Which stock in the Korean market has risen the most in the last month?" as a test question. The system message instructs the AI to answer in Korean as a financial assistant. The API will return a JSON response containing the model's answer. Typically, the response JSON has a structure like:

```
{
  "id": "...",
  "choices": [
    {"message": {"role": "assistant", "content": "<answer text>"}, ...}
  ],
  ...
}
```

We would extract the `content` of the assistant's message as the answer text.

Note: The exact endpoint and model ID can vary; always refer to official CLOVA Studio guides【2†】. Also, the `X-NCP-CLOVASTUDIO-REQUEST-ID` is a unique ID for the request (you can use a random UUID or timestamp). It's optional but was required in the competition setting for tracking. We use `uuid.uuid4()` to generate a random ID each time.

- **Test the call**: Run the above snippet (with a real API key). You should get a response (or any errors if misconfigured). This ensures you can communicate with HyperCLOVA X. Remember that each call to the API may incur usage of your allotted quota. During the evaluation period, the provided key's usage was monitored but not charged to participants (within limits)【1†】.

With yfinance and the HyperCLOVA X API set up, your development environment is ready. You can now proceed to implement the agent's logic.

Part 2: Database & Agent Logic (Advanced RAG and Improvements)

In this phase, we focus on building the agent's core functionality: retrieving relevant data (stock information), using the language model to interpret queries or formulate answers, and employing techniques to improve accuracy and relevance. **RAG (Retrieval-Augmented Generation)** and agent reasoning techniques will be integrated here to enhance the agent's performance.

Data Preparation and Storage

First, consider how to store and access the stock market data that the agent will need. The agent must answer questions about stock prices, volumes, moving averages, etc., often for specific dates or across periods. Relying on live API calls for every query could be too slow or unreliable, especially if a query needs scanning many stocks. A good approach is: - **Pre-fetch and store data**: For example, retrieve historical daily data for a list of stocks (perhaps all stocks in the relevant market index) and store it locally (in memory, a file, or a simple database). You could use a CSV file or a lightweight database like SQLite to store this data. - **Data update**: If the queries involve "recent" data (e.g. "최근" meaning recently), you should update the data periodically (daily or weekly) so that the agent has up-to-date info. In a competition scenario, updating once per day might be acceptable.

Using yfinance to build a local dataset: You can programmatically download data for multiple tickers. For instance, if you have a list of stock tickers:

```
import yfinance as yf
import pandas as pd
```

```
tickers = ["005930.KS", "000660.KS", "035420.KS", ...]  # example ticker symbols
all_data = {}
for tic in tickers:
    df = yf.download(tic, start="2023-01-01", end="2023-12-31")
    all_data[tic] = df
# Optionally, save to CSV/SQLite
```

Here, `"005930.KS"` might represent Samsung Electronics on the KSE (KOSPI) exchange, etc. You would gather all needed tickers (maybe KOSPI 200 companies, or any universe of interest). The `all_data` dict holds each DataFrame. Storing it in a file or database can allow fast querying: for example, in SQLite, you might have a table of daily prices with columns (Date, Ticker, Open, High, Low, Close, Volume). This one-time data ingestion is analogous to the **document ingestion phase** of a RAG pipeline [10], except our "documents" are structured data.

If maintaining a database is too complex for you as a beginner, you can load data into memory when the agent starts. Just be mindful of memory limits (if thousands of stocks, maybe filter by a certain criteria or focus on a subset to keep it manageable).

Agent Query Processing Logic

Now, let's break down how the agent will understand and answer the questions. The PPT provided examples of tasks (queries) the agent must handle. Key categories included:

1. **Simple factual queries** – e.g. "삼성전자 {Date} 주가 얼마나 올랐어?" (How much did Samsung Electronics' stock go up on {Date}?). This implies retrieving the price on {Date} and perhaps the previous day's price to calculate the increase.

2. **Top-N queries** – e.g. "{Date} 거래량 상위 10개 종목은?" (On {Date}, which are the top 10 stocks by trading volume?). This requires sorting stocks by volume on that date.

3. **Conditional queries** – e.g. "전날 대비 거래량이 15% 이상 오른 종목을 모두 보여줘." (Show me all stocks whose trading volume rose more than 15% compared to the previous day). This implies filtering all stocks where today's volume > 1.15 × yesterday's volume.

4. **Technical indicator queries** – e.g. "{Date} 기준으로 50일 이동평균선을 10% 이상 돌파한 종목은?" (As of {Date}, which stocks have their price more than 10% above the 50-day moving average?). This requires computing the 50-day moving average for each stock up to {Date}, then checking which stocks' {Date} price is 1.1× greater than that average.

5. **Ambiguous or interpretative queries** – e.g. "최근에 많이 오른 종목 알려줘" ("Tell me stocks that have gone up a lot recently") or "고점 대비 많이 떨어진 종목은 뭐야?" ("Which stocks have dropped a lot from their peak?"). These are vague: "recently" and "a lot" or "from their peak" need interpretation. The agent has to decide how to quantify these terms.

For each category, the agent's approach will differ slightly. A simple design is to **pattern-match** or use keywords in the question: - If the question contains keywords like "상위 10" ("top 10"), it's likely a top-N volume query. - If it mentions "이동평균선" (moving average), it's a technical indicator query. - If it has a specific percentage and mentions "전날 대비" (compared to previous day volume or price), it's a conditional filter query. - If the query is more free-form ("많이 올랐다", "최근", "고점 대비"), treat it as an **ambiguous query** that may need a default interpretation or a more nuanced approach.

Implementing logic in code: You can parse the incoming question string and branch logic. For example, in a Flask endpoint (which we will set up in Part 3):

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/agent', methods=['GET'])
def answer_question():
    question = request.args.get('question', '')  # get the question from query params
    # Simple parsing logic:
    answer_text = ""
    try:
        if "이동평균" in question:  # moving average task
            answer_text = handle_moving_average_query(question)
        elif "거래량" in question and "상위" in question:
            answer_text = handle_top_volume_query(question)
        elif "거래량" in question and "%" in question:
            answer_text = handle_volume_change_query(question)
        elif "고점" in question or "많이 오른" in question:
            answer_text = handle_ambiguous_query(question)
        else:
            answer_text = handle_simple_price_query(question)
    except Exception as e:
        answer_text = "죄송합니다, 질문을 이해하는 중 오류가 발생했습니다."  # apologize in Korean if
error
    return jsonify({"answer": answer_text})
```

In this pseudo-code: - We check the question and call appropriate handler functions (to be implemented) for each type. - Each `handle_*` function will use the data (from our `all_data` or database) to compute the answer. For instance, `handle_top_volume_query` might: - Parse the date from the question (the question likely contains a date or says “오늘” (today) etc.). - Look up all stocks’ volume on that date. - Sort and pick top 10. - Retrieve their names (you might have a mapping from ticker to company name, since the answer expects names). - Format the answer as a comma-separated list of names with “입니다.” at the end (a polite ending in Korean). - `handle_volume_change_query` would similarly parse the percentage and date, filter stocks by volume criterion. - `handle_moving_average_query` computes the 50-day moving average (likely using pandas: e.g., `df['MA50'] = df['Close'].rolling(window=50).mean()`) and finds those 10% above it. - `handle_simple_price_query` for queries asking about a specific stock’s increase might just fetch the price on given date and maybe previous date to say it rose by X%.

Dealing with ambiguous queries: This is where using the LLM can really help. For questions like “최근에 많이 오른 종목”, a human might clarify “recently” (does it mean last week? last month?) and “a lot” (top 5 performers? risen over 20%?). As a developer, you can set some **assumptions**: e.g., interpret “최근” as the last 1 month, and “많이 오른” as the top 5 stocks by percentage increase over that period. Document these assumptions so evaluators know your interpretation. Then implement accordingly:

```
def handle_ambiguous_query(question):
    # Define "recently" as last 30 days, "a lot" as top 5 by % gain.
```

```
    end_date = today_date  # assume we consider up to today
    start_date = end_date - pd.DateOffset(days=30)
    gains = []
    for tic, df in all_data.items():
        # ensure we have the last 30 days data
        recent = df[df.index >= start_date.strftime("%Y-%m-%d")]
        if len(recent) == 0:
            continue
        start_price = recent['Close'].iloc[0]
        end_price = recent['Close'].iloc[-1]
        pct_gain = (end_price - start_price) / start_price * 100
        gains.append((pct_gain, tic))
    gains.sort(reverse=True)
    top5 = gains[:5]
    top5_names = [ticker_to_name[tic] for _, tic in top5]
    return ", ".join(top5_names) + " 입니다."
```

This code finds top 5 gainers in the last 30 days. Similarly, for “dropped a lot from their peak”, you could: - Determine each stock’s historical high (peak) – maybe the 52-week high. - Calculate how far current price is below that (in %). - List stocks where the drop is largest (e.g., top 5 drops, or drop > 30%, etc., depending on what “많이” might imply).

By quantifying these, the agent gives a reasonable answer. It may not be the only interpretation, but it’s better than the LLM guessing arbitrarily.

Integrating the LLM (HyperCLOVA X) for answers: We have two main ways to use the LLM in the agent: 1. **Understanding the question**: We could use HyperCLOVA X to classify the query type or parse out the details. For instance, we might send the question to the LLM with a prompt like “Classify this question into one of: [simple, topN, volume_change, moving_avg, ambiguous] and extract any date or percentage mentioned.” The model could return a structured answer (if we craft the prompt carefully). However, this might be overkill if simple keyword logic works. Given the time constraints in a competition, many would do simpler parsing. If we had more time, an LLM could handle variations in phrasing more robustly. 2. **Generating the final answer text**: After we compute the result (say a list of stock names or a number), we can feed that to HyperCLOVA X to produce a nicely phrased answer. For example, we might provide a prompt: “질문: ...\n답변을 위한 정보: 종목 A, 종목 B, 종목 C\n이 정보를 활용하여 자연스러운 한국어로 답변하세요.” (Meaning: “Question: ...\nInformation for the answer: ...\nUsing this, answer in natural Korean.”). The model would then output a full sentence answer.

Using the LLM for final phrasing can be helpful to ensure the answer is fluent and handles edge cases (like if there are no stocks meeting the criteria, it could say “해당 조건을 만족하는 종목이 없습니다.”). If you go this route, be sure to include the necessary context in the prompt and perhaps few-shot examples so the model knows how to format the answer. Also, keep the temperature low (e.g., 0.2–0.3) to avoid randomness, since we want a factual answer.

> **RAG approach:** This strategy of retrieving factual data (stock info from yfinance/database) and then supplying it to the LLM for answer generation is exactly **Retrieval-Augmented Generation (RAG)**. It combines the strengths of a knowledge base with the language fluency of the LLM [11] [12]. By providing the model with factual context, we **mitigate hallucinations** (the model making up answers) and ensure up-to-date information is used [11]. The RAG pipeline typically has an **indexing step** (we did that by preparing data) and a

retrieval + generation step at query time [13] [14]. In our case, instead of documents we retrieve numeric data; but the principle is the same – the LLM is augmented with external data it didn't originally train on. The figure below (from NVIDIA) illustrates a general RAG architecture, where a query triggers retrieval from a vector database and the LLM generates the answer using that retrieved info:

Overview of a Retrieval-Augmented Generation (RAG) pipeline: data is ingested and indexed offline; at query time, relevant information is retrieved and combined with the LLM to produce a final answer [15] [14].

If our agent had a large textual knowledge base (for example, company descriptions or news), we could use a vector store and embeddings to fetch relevant text for a question and give it to the model. For the scope of stock price queries, our "knowledge base" is numeric data, so we retrieve and format it as needed.

Agent Reasoning and Improvement Techniques

Beyond data retrieval, we can incorporate advanced **agent reasoning techniques** to improve the quality of answers: - **Chain-of-Thought prompting**: We can prompt HyperCLOVA X to reason through a problem step by step (internally) before finalizing an answer. This is useful for multi-step problems. For instance, for the "peak drop" question, we might ask the model to first consider how to find a peak, then consider the drop. However, since we can compute this ourselves, chain-of-thought is more applicable if the model itself is figuring out the answer. In our design, we offload reasoning about numbers to code, which is more reliable for calculations.

- **ReAct (Reason + Act)**: The ReAct framework combines the model's reasoning with the ability to take actions (like calling tools or functions) [16] [12]. In a more advanced agent, we could let HyperCLOVA X decide when to use a "tool" (like a function that queries data). For example, we could give the model a prompt that includes instructions like: "If you need data, you can ask for it by saying: `Action: get_data(...)`", and then our code would execute that action and return the result for the model to use. This is quite complex to implement from scratch, but frameworks like LangChain provide such capabilities. The key idea is that the model alternates between thinking (generating a reasoning trace) and acting (invoking a tool) [16]. This can lead to more accurate answers on complex queries because the model can fetch exactly what it needs when it realizes it needs it [12]. For instance, the model's reasoning might be: "Thought: To answer this, I need the list of top gainers in the last month. Action: get_data(top_gainers, last_30_days)", then our code returns that list, then model: "Thought: Now I have the list, I can answer." Finally it outputs the answer. ReAct and similar agent paradigms are advanced, but they show how an AI agent can iteratively use tools to improve outcomes.

- **Prompt Engineering and Few-Shot Learning**: We should carefully craft the system prompt for HyperCLOVA X to ensure it answers in the expected format. We might include examples in the prompt (few-shot examples) of Q&A pairs. For example: "Q: 8월 1일 기준 거래량 상위 3종목은?\nA: 삼성전자, 카카오, LG에너지솔루션 입니다.\nQ: … (next question)…\nA: …". By giving a couple of examples in Korean, the model will be more consistent in format (like always ending with "입니다." and separating names with commas). This is especially helpful if we let the model list stocks or if it needs to decide how many to list for an ambiguous query.

- **Fallbacks and Validation**: The agent's code can include sanity checks. If, say, a query yields no results (maybe no stock met a criteria), instead of returning an empty string or letting the model

hallucinate, we handle it: return a message like "조건을 만족하는 종목이 없습니다.". Another example: if a user asks for top 10 volume but our data only has 5 stocks (maybe they filtered a small exchange), ensure the code handles that gracefully. Always wrap computations in try/except as shown, to catch errors and return a friendly message.

By combining these techniques – a solid data foundation, logical parsing, and leveraging the LLM wisely – our agent becomes both **accurate** (due to data-driven answers) and **fluent** (due to the LLM's natural language generation). Retrieval-augmentation specifically helps provide **real-time data access and reduces hallucinations**, as noted in research [17] [18]. And approaches like ReAct ensure the LLM can "know when it doesn't know" and fetch needed info, increasing reliability [16] [12].

At this stage, you should implement and test your agent logic locally. Simulate some queries by calling your functions directly or hitting the Flask endpoint on localhost (e.g., `http://127.0.0.1:8000/agent?question=...`). Ensure the answers make sense and the format matches expectations (especially the output JSON structure `{"answer": "..."}` as required).

Part 3: Deployment as a Web Service (API Endpoint)

Once the agent works correctly on your machine, the final step is deploying it so that the evaluators (or users) can send HTTP requests and get answers. Deployment means running your Flask (or FastAPI) app on a server with a public URL.

Step 1: Choose a Hosting Environment – In the competition context, it was suggested to use **Naver Cloud Platform (NCP) IaaS** (Infrastructure as a Service) to host the agent, and teams were given some cloud credits. You could also use any cloud or a VPS (AWS EC2, Azure, Google Cloud, etc.); the principle is the same. For NCP, you'd create a Linux (Ubuntu) VM instance. Ensure the VM has: - Python installed (you might need to install Python3 and pip). - Access to the internet (to call the HyperCLOVA API and possibly to fetch any data if not all pre-downloaded). - Sufficient memory/CPU for your operations (our agent isn't extremely heavy; a small instance could suffice).

Step 2: Set up the Server – SSH into your server and replicate the environment: - Install system packages if needed (e.g., `sudo apt update && sudo apt install python3-pip` on Ubuntu). - Pull your code from GitHub: Since you kept the repository private (per instructions), you might clone using an access token or upload the code via SCP. Alternatively, you can push your code to the VM using Git if you add your SSH key. - Install the Python requirements on the server via pip (just like you did locally). - Set the environment variable for `NCP_CLOVASTUDIO_API_KEY` on the server. On Linux, you can put it in `~/.bashrc` or create a systemd service file exporting it (explained shortly). The key should not be hardcoded in code on the server either, for security.

Step 3: Run the Flask app – For quick testing, you can simply run `python app.py` (assuming your Flask app is in app.py and listens on `0.0.0.0:8000`). However, for a more robust deployment, consider using a production WSGI server like **Gunicorn**:

```
pip install gunicorn
gunicorn --bind 0.0.0.0:8000 app:app
```

This will run the Flask app and listen on port 8000 for incoming requests. Gunicorn is more stable and can handle multiple requests better than the default Flask dev server.

Make sure the cloud server's firewall allows port 8000, or configure NCP's security group to allow access on that port from the evaluators' IP (or open it temporarily for testing). If using NCP, you might have to assign a public IP to the instance as well.

Step 4: Testing the deployed endpoint – Use the provided evaluation format to test. The competition gave a sample call like this:

Example of how the evaluation system will call your agent's API endpoint. It sends a GET request with the question as a parameter and expects a JSON answer.

In the example above (from the PPT), the evaluator calls `http://<your_server_ip>:8000/agent?question=...` with headers including an Authorization Bearer token and a Request-ID. Your agent should respond with `{"answer": "..."}`. The code snippet printed shows the expected output for the query "거래량이 전날 대비 15% 이상 오른 종목을 모두 보여줘", and the answer was "애드바이오텍, 아난티, 엠에프엠코리아 입니다." (a list of stock names).

A few deployment considerations: - **Include CORS or not?** Probably not needed since the evaluator is just calling your endpoint directly (not via a browser). - **Logging**: Print logs to console or a file to monitor queries and any errors. This will help debug if the evaluator's queries trigger something unexpected. For instance, you can log `question` and `answer` each time. - **Performance**: Ensure that for tasks like top 10 volume or scanning all stocks, your implementation is efficient enough. If you pre-loaded data, queries should be fast (a few milliseconds to filter and sort in pandas or SQL). Calling the HyperCLOVA API will be the slowest step (maybe a few hundred milliseconds or more per query, depending on network), which is usually fine. Just be cautious not to send an overly large prompt; keep the prompt concise (only relevant info) to minimize latency and token usage.

- **Robustness**: The agent should not crash if a weird question comes in. For safety, you might handle a generic else-case by returning "지원하지 않는 질문 유형입니다." ("That type of question is not supported.") or a polite apology, instead of crashing. This way the API always returns a JSON answer.

- **Security**: Since the evaluator includes an `Authorization: Bearer <API_KEY>` header in requests, you could verify it to ensure the request is from a trusted source. For example:

```
AUTH_TOKEN = "the_expected_token"
# In the request handler:
auth_header = request.headers.get("Authorization", "")
if auth_header != f"Bearer {AUTH_TOKEN}":
    return jsonify({"answer": "Unauthorized"}), 401
```

  However, in the provided example, that token was the **same HyperCLOVA API key** used for calling the LLM. They likely included it so that teams could use that key directly for the LLM calls (meaning you don't need to store it separately; you could extract it from the header). This is a bit unusual, but it might have been done to ensure each team's usage is tracked. If that's the case, adjust your code to use `api_key = request.headers.get("Authorization").split(' ')[1]` as the key for LLM calls, instead of an environment variable. Check the competition instructions to be sure. In any case, **do not expose this key** in any response or logs.

Step 5: Final Checks and Submission – Once deployed, test the API from an external location (e.g., your own PC or a cloud function) by replicating the evaluation call format:

```
curl -H "Authorization: Bearer <your_key>" -H "X-NCP-CLOVASTUDIO-REQUEST-ID: test-001" \
  "http://<your_server_ip>:8000/agent?question=삼성전자 8/1일 주가 얼마나 올랐어?"
```

This should return a JSON with the answer. If everything looks good, you can provide the endpoint URL to the evaluators as required (often they asked to put it in the README). Keep your server running throughout the evaluation period (in the PPT, they noted the evaluation would happen between 8/1 and 8/6). You might use a tool like **screen** or **tmux** on Linux to keep the process running even if you disconnect. For a more formal setup, you could create a systemd service so that the app starts on boot and restarts on failure, but that might be beyond the needs of a short-term competition.

Finally, double-check that your GitHub repository (or the code submission) contains all necessary code (especially any logic or data files) and documentation. Document any assumptions or special features of your agent in the README. This helps judges understand your approach. For example, explain how you interpreted "recently" and "a lot" for the ambiguous queries, how frequently you update your data, and how you use HyperCLOVA X in your solution.

By following these steps, you've set up your environment, implemented an intelligent RAG-powered agent, and deployed it for use. Despite being a complex project, we broke it down into manageable stages. Good luck with your financial Q&A agent – with accurate data retrieval and the power of HyperCLOVA X, it should be well-equipped to handle the queries!

Sources:

- Naver CLOVA HyperCLOVA X overview and technical report [8] [9]
- yfinance usage for stock data retrieval [19]
- NVIDIA on Retrieval-Augmented Generation (RAG) benefits and pipeline [11] [15]
- Prompting Guide on ReAct (reasoning+acting) paradigm for LLM agents [16] [12]
- VS Code documentation on using Git integration [1] [2]

---

[1] [2] [3] Introduction to Git in VS Code
https://code.visualstudio.com/docs/sourcecontrol/intro-to-git

[4] [7] yfinance · PyPI
https://pypi.org/project/yfinance/

[5] [6] [19] yfinance: 10 Ways to Get Stock Data with Python | by Kasper Junge | Medium
https://medium.com/@kasperjuunge/yfinance-10-ways-to-get-stock-data-with-python-6677f49e8282

[8] Naver debuts HyperCLOVA X LLM • The Register
https://www.theregister.com/2024/04/08/naver_cloud_hyperclova_llm_sovereign_ai/

[9] Korean Specialized LLM? Overview of HyperCLOVA | by Contemporary Korea | Sep, 2023 | Medium
https://contemporarykorea.medium.com/why-hyperclova-the-future-of-korean-llm-60c7567593e0

[10] [11] [13] [14] [15] [17] [18] RAG 101: Demystifying Retrieval-Augmented Generation Pipelines | NVIDIA Technical Blog
https://developer.nvidia.com/blog/rag-101-demystifying-retrieval-augmented-generation-pipelines/

12 16 ReAct Prompting | Prompt Engineering Guide
https://www.promptingguide.ai/techniques/react