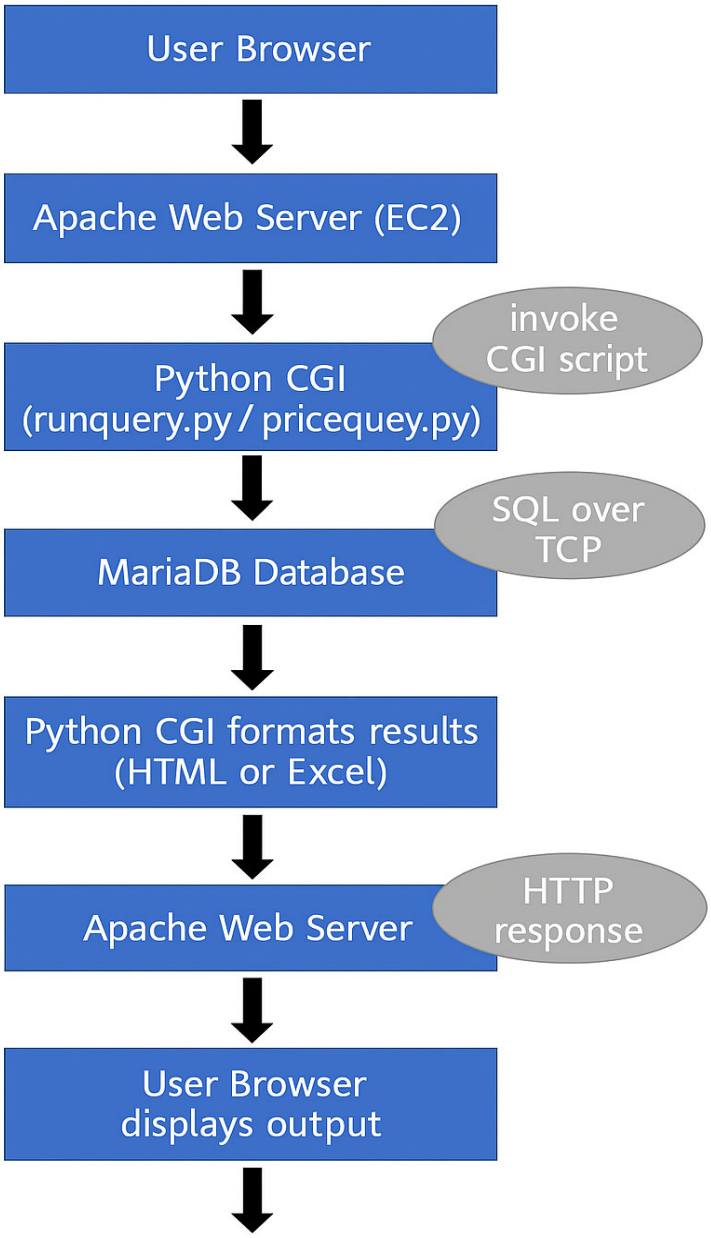
1. Network Architecture & Essay
1.1 Diagram



1.2 Essay

When a user interacts with the web application from Parts I, II, and III, several pieces of the system start working together. The process begins as soon as the user submits a form from the browser. The browser sends an HTTP POST request to the EC2 instance, and this request reaches Apache on port 80. Apache checks the requested URL and sees that it points to one of the Python CGI scripts inside /var/www/cgi-bin. Once the server figures that out, it starts a new process to run the script. The form data is passed into the script through the standard CGI environment, so the Python program can read the user's input. From here, the script acts like a client of its own and connects to the MariaDB server running on the same EC2 instance. Using PyMySQL, the script opens a connection to MariaDB on port 3306 and sends the SQL query. MariaDB processes the query and returns the rows, and the Python script formats the results into an HTML table. After adding the required Content-Type header, the script sends the HTML output back to Apache. Apache then wraps that output inside an HTTP response and returns it to the user's browser.

The browser simply renders the HTML it receives. Walking through the whole process helped me see how each part browser, Apache server, CGI script, and MariaDB passes data to the next, and how they work together to support a dynamic web application. Understanding this flow made the interactions between the components much clearer to me.

1.3 Network flow

1) Browser submits HTML form (SQL query or date/price).
2) Apache on the EC2 instance receives the HTTP request on port 80.
3) Apache invokes the corresponding Python CGI script in /var/www/cgi-bin.
4) The CGI script connects to MariaDB, runs the SQL query, and formats the results.
5) Apache sends the HTML response back to the browser, which displays the table.

Question about CGI and alternative technologies

CGI was useful for teaching, but in practice it has several drawbacks: every request starts a new process, it is hard to manage state, and performance does not scale well. Modern web applications usually rely on other technologies to connect web servers and databases.

One common alternative is using server-side scripting inside the web server itself, such as PHP modules in Apache. In this model, the PHP interpreter runs inside the server process, so each request does not need to spawn a new process. PHP scripts can easily query MySQL and return HTML.

Another approach is application servers and frameworks, such as Java Servlets and JSP, Django or Flask in Python, Ruby on Rails, or ASP.NET. In these systems, the web server forwards requests to an application layer that keeps a pool of worker processes or threads. This layer manages routing, database access, templates, and sessions.

More recently, many systems use RESTful APIs built with Node.js and Express or similar frameworks. The back-end exposes JSON endpoints, and the front-end is built with JavaScript frameworks like React or Vue. The API server handles database queries and returns data instead of raw HTML.

On cloud platforms, serverless functions such as AWS Lambda are another option. A Lambda function can be triggered by an HTTP request through API Gateway, run a short piece of code to query a database, and return a response without managing any servers directly. This model is flexible and can be cost-effective for small workloads.

Overall, these modern alternatives provide better performance and scalability compared to traditional CGI.

2. Testing and reflection

2.1. System Test Screenshots

2.1.1 SSH login screen (initial connection)

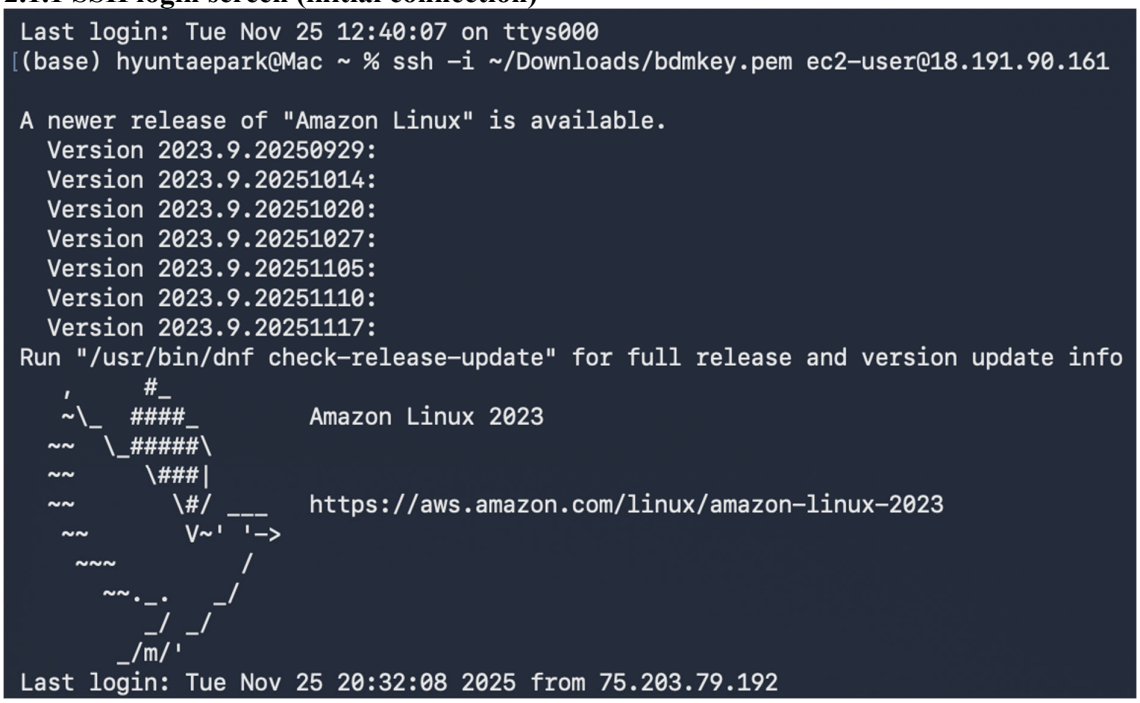


Figure 5.1.1

This screenshot shows the moment I connected to the EC2 instance using SSH. It confirms that the server is running correctly and that I was able to access the Amazon Linux environment without any issues.

2.1.2 Apache access log (including GET /favicon.ico)

Figure 5.1.2

This screenshot shows a portion of the Apache access log recorded during testing. It captures several client requests, including POST requests to the CGI scripts and a GET request for */favicon.ico*. These entries confirm that the web server correctly received and processed the interactions from the browser.

2.1.3 EC2 directory listing: /var/www/cgi-bin

```
[ec2-user@ip-172-31-33-107 cgi-bin]$ ls -l /var/www/cgi-bin
total 12
-rwxr-xr-x. 1 root apache 1622 Nov 25 20:37 pricequery.py
-rwxr-xr-x. 1 root apache 1222 Nov 25 17:44 runquery.py
-rwxr-xr-x. 1 root apache  125 Nov 19 17:13 test.py
[ec2-user@ip-172-31-33-107 cgi-bin]$
```

Figure 5.1.3

This screenshot shows the contents of the */var/www/cgi-bin directory* on the EC2 instance. The CGI scripts used in this assignment *pricequery.py*, *runquery.py*, and *test.py* are all present with the correct permissions, confirming that the server is set up to execute them properly.

2.1.4 MariaDB access and SELECT * FROM price

```
[ec2-user@ip-172-31-33-107 cgi-bin]$ mysql -u root -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 2535
Server version: 10.5.29-MariaDB MariaDB Server

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> USE homework2;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
MariaDB [homework2]> SELECT * FROM price;
+--------+------------+--------+----------+
| ticker | date       | close  | exchange |
+--------+------------+--------+----------+
| IBM    | 2024-02-25 | 120.50 | NYSE     |
| GOOG   | 2024-02-25 | 150.75 | NASDAQ   |
| TSLA   | 2024-02-25 | 180.10 | NASDAQ   |
| T      | 2024-02-25 |  18.20 | NYSE     |
| XOM    | 2024-02-25 |  95.30 | NYSE     |
+--------+------------+--------+----------+
5 rows in set (0.000 sec)
```

Figure 5.1.4

This screenshot shows the process of connecting to the MariaDB server and querying the *price* table. After selecting the correct database, I ran a *SELECT * FROM price*; command, and the output confirms that the sample ticker data was inserted correctly and can be retrieved without errors.

2.1.5 query.html : SQL input form

Figure 5.1.5

This screenshot shows the SQL input page provided by *query.html*. The user can type any SQL command into the text box and submit it using the "Run Query" button. When submitted, the form sends the request to the *runquery.py* script, which processes the SQL command on the server.

2.1.6 priceform.html : date & price input form

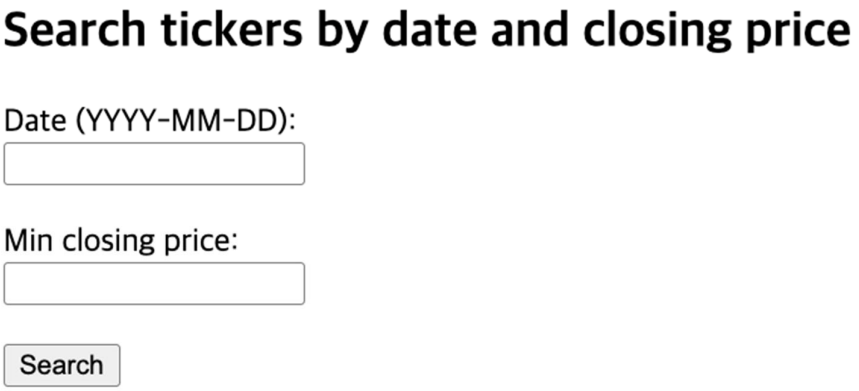


Figure 5.1.6-1

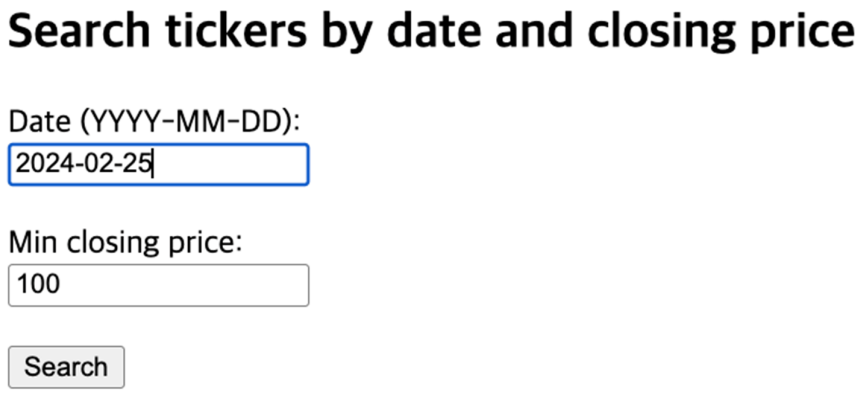


Figure 5.1.6-2

These screenshots show the input form provided by *priceform.html*. The user can enter a specific date and a minimum closing price to filter the stock data. The first image shows the blank form, and the second one shows an example with values filled in. When the form is submitted, the data is sent to the *pricequery.py* script for processing.

2.1.7 pricequery.py output page : successful query

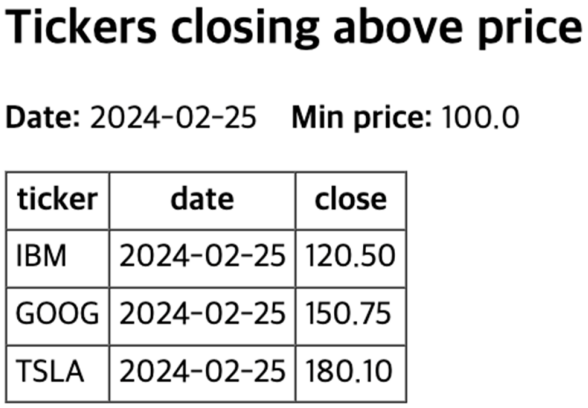


Figure 5.1.7

This screenshot shows the output generated by *pricequery.py* when valid input values are provided. The script returns a table titled *"Tickers closing above price"*, listing only the tickers that meet the date and minimum closing price conditions. It confirms that the CGI script successfully processed the form data and retrieved the correct records from the database.

2.1.8 pricequery.py error page : missing input

Figure 5.1.8

This screenshot shows the error message displayed when the user submits the form without providing both the date and the minimum closing price. The script checks for missing inputs and returns this message to prevent the query from running with incomplete data.

2.1.9 Forbidden 403 error page

Figure 5.1.9

This screenshot shows the 403 Forbidden error that appears when the server blocks access to a resource due to insufficient permissions. It demonstrates how Apache responds when a file or directory is not configured to allow public access.

2.1.10 EC2 directory listing: /var/www/html

```
[ec2-user@ip-172-31-33-107 cgi-bin]$ ls -l /var/www/html
total 36
-rw-r--r--.  1 ec2-user apache    26 Nov 19 16:03 mytest,html
-rw-r--r--.  1 ec2-user apache   137 Nov 19 16:08 mytest.html
drwxr-sr-x. 12 ec2-user apache 16384 Oct  8 14:51 phpMyAdmin
-rw-r--r--.  1 ec2-user apache    20 Oct  1 16:49 phpinfo.php
-rw-r--r--.  1 root     apache   351 Nov 25 18:01 priceform.html
-rw-r--r--.  1 root     apache   254 Nov 25 17:16 query.html
```

Figure 5.1.10

This screenshot shows the contents of the */var/www/html* directory, which stores the static HTML files for the web server. The listing confirms that the pages used in this assignment such as *priceform.html* and *query.html* are correctly placed in the document root and accessible through the browser.

2.2. Reflection

During this assignment, I got a much clearer sense of how the pieces of a web-based system actually work together. Setting up the CGI environment on the EC2 instance took more time than I expected, especially with permissions and Apache settings, and I ran into a few frustrating errors along the way. But working through those issues helped me understand what the server is doing behind the scenes and why certain configurations matter.

I also paid more attention to how the browser, Apache, the Python scripts, and MariaDB all pass information to each other. Seeing the flow from the form submission to the CGI script, then to the database, and back to the browser made the whole process feel more real instead of just theoretical. Completing the SQL query page and the price-check tool helped me see how the back-end logic connects with actual data. Overall, this project helped me feel more comfortable with networking basics and server-side programming.

3. Excel Export using Pandas

In this bonus part of the assignment, I added a feature that lets the web application return the SQL results as an Excel file. The idea was similar to the price query page, but instead of showing the output as an HTML table, I wanted the user to be able to download the results. After receiving the date and minimum price from the form, the CGI script connects to MariaDB and runs the SQL query as usual. I used Pandas to turn the result set into a DataFrame, and then exported it as an .xlsx file using *to_excel()* with *sys.stdout.buffer* so the file could be sent directly to the browser. I also had to set the proper MIME type and header so that the browser treats the output as a file download.

Overall, this part helped me see how Python, Pandas, and the CGI setup can work together to provide features beyond simple HTML output.

3.1 Initial Download Page (Empty Form)

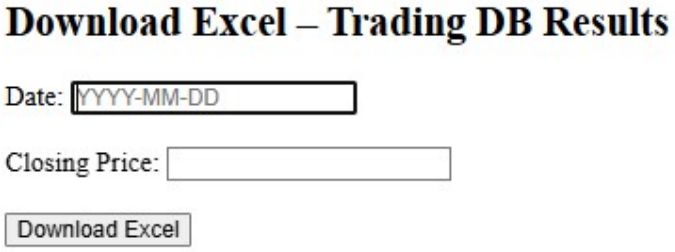


Figure 6.1

This screenshot shows the initial state of the Excel download page. The user is first presented with a simple HTML form that asks for two inputs: the date and the minimum closing price. At this stage, both fields are empty, indicating the default view before any user interaction. This form is used to pass parameters to the CGI script, although in the final implementation the script retrieves all rows from the database regardless of the input.

3.2 Download Page After User Input

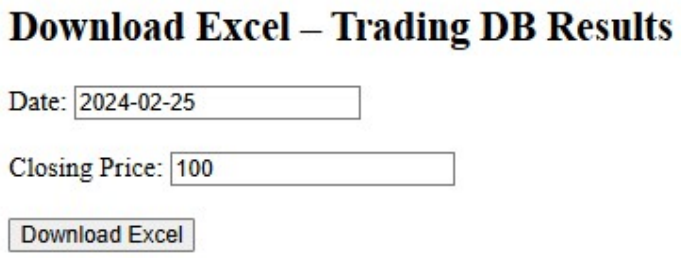


Figure 6.2

This screenshot shows the form after the user has entered a date *2024-02-25* and a minimum price *100*. When the "Download Excel" button is clicked, these values are submitted to the CGI script. Even though the script does not filter based on these parameters in the final version, this step demonstrates the form submission process and how the interface is intended to be used.

3.3 Excel File Generated by the CGI Script

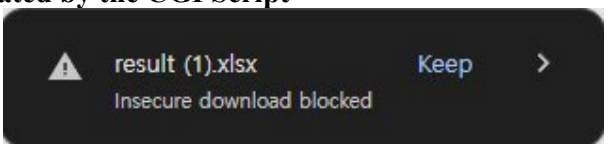


Figure 6.3

This screenshot confirms that the CGI script successfully generated an Excel file *result.xlsx* and sent it to the browser. The browser recognizes the correct MIME type and handles it as a downloadable file. The small warning "Insecure download blocked" is normal for HTTP connections without SSL and does not affect the correctness of the functionality.

3.4 Contents of the Generated Excel File

	A	B	C	D
1	ticker	date	close	exchange
2	IBM	2024-02-25	120.50	NYSE
3	GOOG	2024-02-25	150.75	NASDAQ
4	TSLA	2024-02-25	180.10	NASDAQ
5	T	2024-02-25	18.20	NYSE
6	XOM	2024-02-25	95.30	NYSE

Figure 6.4

This screenshot displays the contents of the Excel file produced by the CGI script. The file includes the four columns *ticker, date, close, exchange* and five rows of data retrieved from the *price* table in the *homework2* MariaDB database. This confirms that the script successfully connected to the database, executed the SQL query, turned the results into a Pandas DataFrame, converted it into an Excel file, and delivered it to the browser.

3.5 SQL Query and Data Retrieval

```
# 1. DB
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="bdmhyuntae",
    database="homework2"
)
cursor = conn.cursor()
```

Figure 6.5

This code block shows how the CGI script connects to the MariaDB server. Using the *mysql.connector* library, the script establishes a connection with the *homework2* database using the root credentials. After the connection is established, a cursor object is created, which is later used to execute SQL queries. This step is essential for retrieving data that will eventually be exported to Excel.

3.6 SQL Query and Data Retrieval

```
# 2. price table
query = """
    SELECT ticker, date, close, exchange
    FROM price
"""
cursor.execute(query)
rows = cursor.fetchall()
columns = [desc[0] for desc in cursor.description]

cursor.close()
conn.close()
```

Figure 6.6

This section of the code demonstrates how the script retrieves data from the database. The SQL query selects *ticker, date, close*, and *exchange* columns from the *price* table. The cursor executes the query, and the results are fetched into Python as a list of rows. Column names are also extracted from the cursor's metadata so that the resulting Excel file can include proper headers. The cursor and connection are then closed to release database resources.

3.7 Converting Query Results to an Excel File

```
# 3. Pandas DataFrame → Excel
df = pd.DataFrame(rows, columns=columns)
output = io.BytesIO()
df.to_excel(output, index=False)
excel_data = output.getvalue()
```

Figure 6.7

In this part, the script uses Pandas to convert the SQL query results into an Excel file. The rows and column names are loaded into a *DataFrame*, which is then written to an in-memory buffer using to*_excel()*. The buffer's binary contents are extracted and later written to *sys.stdout.buffer* so that the browser can download the file. This block demonstrates how Python and Pandas can dynamically generate Excel files without storing them on the server.