1. **Normal Process (Top Part)**:
   * The user asks a question like "Who discovered penicillin?"
   * The LLM generates a query in JSON format to an external API (such as a Wiki API).
   * The API returns the correct JSON response, e.g., {"Extract": "Penicillin was discovered by Alexander Fleming in 1928"}.
   * The LLM converts this API response into a human-readable output, giving the correct answer to the user.
2. **Attack Process (Bottom Part)**:
   * An attacker intercepts the response from the API and alters the data.
   * Instead of the correct answer, the attacker modifies the response to return false information, e.g., {"Extract": "Penicillin was discovered by John Doe in 1930"}.
   * The LLM processes this incorrect data and displays the wrong answer to the user.

### Relating This to Our Code and Defense Mechanisms

The code i implemented can **directly mitigate** this type of attack through several layers of defense:

#### 1. **Response Integrity Verification (Hashing)**:

* In our code, after receiving a response from an external API, i compute a **SHA-256 hash** of the response and compare it with a pre-shared or stored hash.
* **Real-world protection**: If an attacker intercepts and alters the response (e.g., changing the name of the person who discovered penicillin), the hash comparison will **fail**, indicating tampering.
* **Example**: If the correct response should be {"Extract": "Penicillin was discovered by Alexander Fleming in 1928"}, the hash would be calculated on this string. If the attacker changes it to "John Doe in 1930", the hash comparison would catch this and reject the altered response.

#### 2. **Encryption of API Keys**:

* The code uses **RSA encryption** to protect the API key during communication with the external API.
* **Real-world protection**: This ensures that even if an attacker intercepts the request, they cannot steal the API key and use it to send malicious queries or manipulate responses. Encrypted keys safeguard your system from unauthorized use.

#### 3. **Rate Limiting**:

* Our implementation includes rate limiting, preventing any client from making too many requests within a given timeframe.
* **Real-world protection**: This reduces the risk of an attacker spamming the API with malicious or tampered requests, limiting the effectiveness of such an attack. Rate limiting ensures the API isn’t overwhelmed or abused by continuous requests that could manipulate or overload the system.

#### 4. **Input Validation**:

* The code validates inputs to ensure that only predefined, expected cities or terms are sent to the external API.
* **Real-world protection**: This guards against attacks where an attacker might inject unexpected or harmful inputs into the request, which could lead to API misbehavior or vulnerability exploitation. The input validation prevents attackers from manipulating the user's request before it's sent to the API.

### Example of How Our Code Solves the Problem:

Let’s adapt the image example (about the penicillin discovery) to weather data (as code focuses on weather API responses):

1. **Scenario**:
   * The user asks: "What’s the weather like in Karachi?"
   * Our app sends this request to the external weather API.
2. **Normal Process (Secure Flow)**:
   * The weather API returns the correct data: {"city": "Karachi", "temperature": "30°C", "humidity": "60%"}.
   * The code computes the hash of this response and compares it with a known correct hash.
   * Since the data hasn’t been altered, the hash matches, and the response is sent to the user.
3. **Attack Process (Attempted MitM)**:
   * An attacker intercepts the response and modifies it to provide misleading information: {"city": "Karachi", "temperature": "50°C", "humidity": "90%"}.
   * The code computes the hash of this tampered response and compares it with the known hash.
   * The hash comparison **fails**, detecting the tampering.
   * The system rejects the response and does not display the false information to the user.

By incorporating these defense mechanisms, code effectively **prevents tampering** of API responses, as shown in the image. It ensures data integrity and protects users from receiving incorrect or harmful information due to attacks like the one depicted.

### Key Takeaways:

* use of **hash verification** directly addresses the kind of attack, ensuring the integrity of data returned by APIs.
* **RSA encryption** secures sensitive information like API keys, preventing them from being stolen or misused by attackers.
* **Rate limiting** and **input validation** provide further defenses against API abuse and malicious input injection, which could compromise the system's functionality or expose vulnerabilities.

By combining these strategies, our system effectively combats man-in-the-middle attacks and ensures users receive accurate and trusted information.

**In this I use hashes to verify the data so here is the steps how hashes Is work in our scnerio:-**

 **Step 1: Receive the Weather API Data**:

* After server calls the Weather API, it receives a **JSON response** containing weather data like temperature, humidity, etc.
* Immediately, the server generates a hash for this data to protect against potential tampering later.

 **Step 2: Send the Data and Hash to the Client**:

* server sends the **weather data** and the **hash of this data** to the client (user, frontend, or another service).

 **Step 3: Store or Return the Hash with the Data**:

* The **hash** is sent along with the weather data. This ensures that when the data comes back (e.g., during a subsequent verification), you can rehash the received data and compare it with the original hash to detect tampering.

 **Step 4: Verify Data Hash During the Response**:

* Before return the final data to the user or client, **rehash the data** received and compare it with the **hash** that was sent originally. If the two hashes match, it means the data has not been tampered with.