To build a robust and scalable document retrieval system as described in the task for the AI Intern role at VIT Campus Hiring 2025, it is essential to break down the entire project into modular components while ensuring efficient coding practices, proper system design, and optimal implementation of technologies. The backend system you are tasked with should serve as a crucial building block for retrieving relevant documents that large language models (LLMs) can use as context during inference. In this task, you'll be handling different components like database interaction, caching strategies, concurrency, and API design.

Problem Breakdown

The problem statement specifies that you are required to develop a backend system for document retrieval that must be optimized for performance, accuracy, and scalability. The system should be designed in such a way that it can handle real-time requests, cache responses for faster retrieval, and scrape documents in the background. Below is an in-depth explanation of how each component should be tackled and implemented.

1. **Backend Development**

The first task is to build a reliable backend that will serve as the core of the document retrieval system. It should be built using Python or Go, as specified in the task. The backend's primary function is to interact with a database that stores all the documents and retrieve the most relevant documents based on a user's query.

The following aspects must be considered when developing the backend:

- **Programming Language Choice**: Either Python or Go are permissible for the development of the backend. Python is well-suited for handling text-based operations and has a rich ecosystem of machine learning and NLP libraries such as TensorFlow, PyTorch, and HuggingFace. Go, on the other hand, is known for its performance efficiency and can be a good choice for high-concurrency applications, though it lacks Python's extensive NLP libraries.
- **Modularity**: Ensure that the backend is divided into smaller, reusable modules. For instance, have separate modules for handling API requests, database interaction, caching, and concurrency.

2. **Database Design and Interaction**

Storing documents in a well-designed database is critical. When choosing a database, consider how the data will be accessed and queried. The goal here is to ensure fast and efficient retrieval of documents. Some popular choices for databases include:

- **Relational Databases (SQL-based)**: These are useful when there is a clear structure to the documents being stored, and you can model them in terms of rows and tables.
- **NoSQL Databases**: These are often better for handling unstructured or semi-structured data, which is typical of documents. Databases like MongoDB or Elasticsearch are ideal choices here. Elasticsearch, in particular, is known for its full-text search capabilities, making it suitable for document retrieval.
- **Encoders**: You are allowed to use any document encoders of your choice. One effective method is to use sentence encoders like Sentence-BERT, which can encode documents into vector representations. These vectors can then be stored in the database and used for similarity-based retrieval.

The documents should be stored in a vectorized form to facilitate efficient retrieval using similarity-based queries. The database should be designed in a way that supports fast querying, potentially using inverted indices, especially if Elasticsearch is used.

3. **Caching Strategy**

Caching is critical for improving the system's overall performance. When a user makes a search request, the system should be able to return results from the cache if available, reducing the need to interact with the database for every query.

- **Choosing the Right Cache**: The task suggests that solutions like Memcached might not be the best for this particular use case. Alternatives like **Redis** can be used, which supports various caching strategies and offers faster retrieval of data.

- **Cache Eviction Policies**: To optimize cache storage, implement suitable cache eviction policies. Common strategies include Least Recently Used (LRU), Least Frequently Used (LFU), or Time-to-Live (TTL) based caches.

- **Response Caching**: Cache the responses of document searches for future reuse. For example, if a user requests similar documents with the same query text, the system should return the cached results instead of reprocessing the request.

Caching will also help in rate-limiting scenarios. If a user makes multiple requests, the system can respond quickly from the cache instead of executing the full pipeline again.

4. **Concurrency and Multithreading**

One important part of the project is that there should be a background process that scrapes news articles when the server starts. This means your backend needs to be capable of handling multiple tasks simultaneously.

- **Concurrency Model**: You can use a multithreaded or multiprocessing approach to handle background tasks like scraping articles while still responding to incoming API requests. Python's threading or asyncio libraries can be helpful in this case.

- **Background Scraping**: The server should automatically start a separate thread to scrape news articles once the system is up and running. Scraping can be done using libraries like Scrapy or BeautifulSoup for Python, and Gocolly for Go.

- **Worker Threads**: Implement a worker system to handle these background scraping tasks. The scraped articles can then be stored in the database and vectorized for future searches. Make sure this process is separate from the request-handling process to avoid performance bottlenecks.

Concurrency will also be helpful when handling multiple search requests simultaneously. By utilizing threads, the system can cater to multiple users efficiently without causing delays.

5. **API Design**

The API design is one of the key scoring criteria, and it needs to be well-structured, secure, and optimized for performance. The following endpoints should be implemented:

- **/health Endpoint**: This endpoint is used to check if the API is active and running. It can simply return a 200 OK status along with a simple response like "API is active".

- **/search Endpoint**: This is the main endpoint of the application, where users will input a search query, and the system will return a list of the top results based on the query. The `/search` endpoint should accept parameters like:
  - **Text**: This is the query text input by the user.
  - **top_k**: Specifies the number of top results the user wants.
  - **Threshold**: Specifies a threshold for similarity scores.

  The system should return results based on the similarity between the query and the documents in the database. Similarity can be computed using cosine similarity or other distance metrics between the query vector and the document vectors.

- **User ID**: The system must track each request made by users. If the user already exists in the system, the frequency of API calls should be incremented. If the user makes more than 5 requests, the system should return HTTP status code 429 (Too Many Requests).

Make sure that these endpoints are well-secured, especially when dealing with user IDs. You may consider using API tokens or OAuth for user authentication and authorization.

6. **Rate Limiting and Monitoring**

To prevent abuse of the system, rate limiting needs to be implemented. As specified in the task, if a user makes more than 5 requests, the system should return a 429 status code. Rate limiting can be done using Redis to store the number of requests per user in a time window.

Additionally, the system should log the inference time for each request. Logging can be done using structured logging tools like Python's logging module or logging frameworks in Go.

7. **Error Handling and Logging**

Robust error handling is crucial for creating a reliable system. The system should be able to gracefully handle edge cases like empty queries, malformed requests, or database timeouts. All errors should be logged for debugging purposes. Use structured logging to capture relevant details like the timestamp, user ID, and error message.

For logging, you can consider using centralized logging solutions like **Elasticsearch** with **Kibana** or **Prometheus** for monitoring.

8. **Dockerization**

Once the backend is complete, the next step is to **Dockerize** the application to make it easier to deploy and scale. Docker allows you to package the entire application along with its dependencies into a container, which can then be deployed anywhere.

- **Dockerfile**: Create a `Dockerfile` that specifies the environment setup, dependencies, and how to run the application.
- **Docker Compose**: If the application depends on multiple services (like a database, cache, or background worker), use Docker Compose to define and manage these services in one configuration file.

Dockerizing the application will make it easier to deploy on cloud platforms like AWS, GCP, or Azure.

9. **Advanced Features (Bonus Tasks)**

For additional scoring points, you can implement advanced features such as:

- **Re-Ranking Algorithms**: After retrieving the top results based on similarity, you can apply re-ranking algorithms to further refine the results based on more specific metrics (like user preferences or document freshness).

- **Fine-Tuning Scripts for Retrievers**: You can fine-tune the document retrievers using domain-specific data to improve the accuracy of the system.

10. **System Design and Best Practices**

Throughout the implementation, follow good system design practices. This includes:
- **Scalability**: Ensure that the system can scale horizontally by distributing the database and cache over multiple nodes.
- **Security**: Secure the API using appropriate authentication mechanisms.
- **Modularity**: Write modular, reusable code with clear separation of concerns.

Conclusion

Building a document retrieval system for LLM inference is a complex task that requires careful planning and execution. By focusing on efficient database design, caching, concurrency, API development, and rate-limiting, you can create a high-performing system that meets the outlined requirements. Dockerizing the application ensures that it can be easily deployed and scaled.