Question 1

Question Type: MultipleChoice

A Generative AI Engineer has been reviewing issues with their company's LLM-based question-answering assistant and has determined that a technique called prompt chaining could help alleviate some performance concerns. However, to suggest this to their team, they have to clearly explain how it works and how it can benefit their question-answering assistant. Which explanation do they communicate to the team?

Options:

A- It allows you to break down complex tasks into multiple independent subtasks. This enables the assistant to generate more comprehensive and accurate responses.
B- It allows you to reduce the latency of your applications. By having multiple chains participating in the response as a chain, you increase the rate at which the response is generated.
C- It allows you to decrease the effort involved in crafting a prompt. Chains make it possible to reuse prompt text across multiple different use cases.
D- It reduces the average cost of a typical request. Chains make more efficient use of the tokens produced to generate higher quality responses with fewer tokens.

Answer:

A

Explanation:

Prompt chaining is a fundamental design pattern in LLM application development used to handle complexity. Instead of sending a single, massive, and highly complex prompt to an LLM---which often results in reasoning errors or hallucinations---chaining breaks the logic into a sequence of smaller, targeted steps. For example, a legal assistant might first chain a step to 'identify the legal jurisdiction,' followed by a step to 'extract relevant statutes,' and finally a step to 'summarize the findings.' This modularity improves reliability because each prompt has a narrower focus, making it easier for the model to follow instructions accurately. While it may actually increase latency (contradicting B) and cost (contradicting D) due to multiple API calls, the primary engineering benefit is the significant boost in the quality and robustness of the output. It also allows for intermediate validation and error handling between steps, which is impossible in a single-call architecture.

Question 2

Question Type: MultipleChoice

A Generative AI Engineer at an automotive company would like to build a question-answering chatbot to help customers answer specific questions about their vehicles. They have:

A catalog with hundreds of thousands of cars manufactured since the 1960s

Historical searches with user queries and successful matches

Descriptions of their own cars in multiple languages

They have already selected an open-source LLM and created a test set of user queries. They need to discard techniques that will not help them build the chatbot. Which do they discard?

Options:

A- Setting chunk size to match the model's context window to maximize coverage
B- Implementing metadata filtering based on car models and years
C- Fine-tuning an embedding model on automotive terminology
D- Adding few-shot examples for response generation

Answer:

A

Explanation:

According to Generative AI engineering standards for Retrieval-Augmented Generation (RAG), chunking strategy is a critical optimization variable. Setting the chunk size to match the model's maximum context window (e.g., 4k or 8k tokens) is a poor practice and should be discarded. Large chunks introduce significant 'noise' into the LLM's context, as only a small portion of a massive chunk usually contains the answer to a specific query. This leads to the 'lost in the middle' phenomenon where LLMs struggle to extract relevant information from bloated contexts. Furthermore, large chunks reduce the precision of the vector search. Standard best practices involve using smaller, semantically meaningful chunks (typically 256--512 tokens) with overlap to maintain context. In contrast, metadata filtering (B) is essential for narrowing searches to specific car years, fine-tuning embeddings (C) improves retrieval accuracy for domain-specific technical terms, and few-shot examples (D) guide the LLM's output format and tone.

Question 3

Question Type: MultipleChoice

A Generative AI Engineer is building a system which will answer questions on latest stock news articles.

Which will NOT help with ensuring the outputs are relevant to financial news?

Options:

A- Implement a comprehensive guardrail framework that includes policies for content filters tailored to the finance sector.

B- Increase the compute to improve processing speed of questions to allow greater relevancy analysis

C Implement a profanity filter to screen out offensive language

D- Incorporate manual reviews to correct any problematic outputs prior to sending to the users

Answer:

B

Explanation:

In the context of ensuring that outputs are relevant to financial news, increasing compute power (option B) does not directly improve the relevance of the LLM-generated outputs. Here's why:

Compute Power and Relevancy: Increasing compute power can help the model process inputs faster, but it does not inherently improve the relevance of the answers. Relevancy depends on the data sources, the retrieval method, and the filtering mechanisms in place, not on how quickly the model processes the query.

What Actually Helps with Relevance: Other methods, like content filtering, guardrails, or manual review, can directly impact the relevance of the model's responses by ensuring the model focuses on pertinent financial content. These methods help tailor the LLM's responses to the financial domain and avoid irrelevant or harmful outputs.

Why Other Options Are More Relevant:

A (Comprehensive Guardrail Framework): This will ensure that the model avoids generating content that is irrelevant or inappropriate in the finance sector.

C (Profanity Filter): While not directly related to financial relevancy, ensuring the output is clean and professional is still important in maintaining the quality of responses.

D (Manual Review): Incorporating human oversight to catch and correct issues with the LLM's output ensures the final answers are aligned with financial content expectations.

Thus, increasing compute power does not help with ensuring the outputs are more relevant to financial news, making option B the correct answer.

Question 4

Question Type: MultipleChoice

A team wants to serve a code generation model as an assistant for their software developers. It should support multiple programming languages. Quality is the primary objective.

Which of the Databricks Foundation Model APIs, or models available in the Marketplace, would be the best fit?

Options:

A- Llama2-70b

B- BGE-large

C- MPT-7b

D- CodeLlama-34B

Answer:

D

Explanation:

For a code generation model that supports multiple programming languages and where quality is the primary objective, CodeLlama-34B is the most suitable choice. Here's the reasoning:

Specialization in Code Generation: CodeLlama-34B is specifically designed for code generation tasks. This model has been trained with a focus on understanding and generating code, which makes it particularly adept at handling various programming languages and coding contexts.

Capacity and Performance: The '34B' indicates a model size of 34 billion parameters, suggesting a high capacity for handling complex tasks and generating high-quality outputs. The large model size typically correlates with better understanding and generation capabilities in diverse scenarios.

Suitability for Development Teams: Given that the model is optimized for code, it will be able to assist software developers more effectively than general-purpose models. It understands coding syntax, semantics, and the nuances of different programming languages.

Why Other Options Are Less Suitable:

A (Llama2-70b): While also a large model, it's more general-purpose and may not be as fine-tuned for code generation as CodeLlama.

B (BGE-large): This model may not specifically focus on code generation.

C (MPT-7b): Smaller than CodeLlama-34B and likely less capable in handling complex code generation tasks at high quality.

Therefore, for a high-quality, multi-language code generation application, CodeLlama-34B (option D) is the best fit.

Question 5

Question Type: MultipleChoice

What is an effective method to preprocess prompts using custom code before sending them to an LLM?

Options:

A- Directly modify the LLM's internal architecture to include preprocessing steps
B- It is better not to introduce custom code to preprocess prompts as the LLM has not been trained with examples of the preprocessed prompts
C- Rather than preprocessing prompts, it's more effective to postprocess the LLM outputs to align the outputs to desired outcomes
D- Write a MLflow PyFunc model that has a separate function to process the prompts

Answer:

D

Explanation:

The most effective way to preprocess prompts using custom code is to write a custom model, such as an MLflow PyFunc model. Here's a breakdown of why this is the correct approach:

MLflow PyFunc Models: MLflow is a widely used platform for managing the machine learning lifecycle, including experimentation, reproducibility, and deployment. A PyFunc model is a generic Python function model that can implement custom logic, which includes preprocessing prompts.

Preprocessing Prompts: Preprocessing could include various tasks like cleaning up the user input, formatting it according to specific rules, or augmenting it with additional context before passing it to the LLM. Writing this preprocessing as part of a PyFunc model allows the custom code to be managed, tested, and deployed easily.

Modular and Reusable: By separating the preprocessing logic into a PyFunc model, the system becomes modular, making it easier to maintain and update without needing to modify the core LLM or retrain it.

Why Other Options Are Less Suitable:

A (Modify LLM's Internal Architecture): Directly modifying the LLM's architecture is highly impractical and can disrupt the model's performance. LLMs are typically treated as black-box models for tasks like prompt processing.

B (Avoid Custom Code): While it's true that LLMs haven't been explicitly trained with preprocessed prompts, preprocessing can still improve clarity and alignment with desired input formats without confusing the model.

C (Postprocessing Outputs): While postprocessing the output can be useful, it doesn't address the need for clean and well-formatted inputs, which directly affect the quality of the model's responses.

Thus, using an MLflow PyFunc model allows for flexible and controlled preprocessing of prompts in a scalable way, making it the most effective method.

Question 6

Question Type: MultipleChoice

A Generative AI Engineer at an automotive company would like to build a question-answering chatbot to help customers answer specific questions about their vehicles. They have:

A catalog with hundreds of thousands of cars manufactured since the 1960s

Historical searches with user queries and successful matches

Descriptions of their own cars in multiple languages

They have already selected an open-source LLM and created a test set of user queries. They need to discard techniques that will not help them build the chatbot. Which do they discard?

Options:

A- Setting chunk size to match the model's context window to maximize coverage
B- Implementing metadata filtering based on car models and years
C- Fine-tuning an embedding model on automotive terminology
D- Adding few-shot examples for response generation

Answer:

A

Explanation:

According to Generative AI engineering standards for Retrieval-Augmented Generation (RAG), chunking strategy is a critical optimization variable. Setting the chunk size to match the model's maximum context window (e.g., 4k or 8k tokens) is a poor practice and should be discarded. Large chunks introduce significant 'noise' into the LLM's context, as only a small portion of a massive chunk usually contains the answer to a specific query. This leads to the 'lost in the middle' phenomenon where LLMs struggle to extract relevant information from bloated contexts. Furthermore, large chunks reduce the precision of the vector search. Standard best practices involve using smaller, semantically meaningful chunks (typically 256--512 tokens) with overlap to maintain context. In contrast, metadata filtering (B) is essential for narrowing searches to specific car years, fine-tuning embeddings (C) improves retrieval accuracy for domain-specific technical terms, and few-shot examples (D) guide the LLM's output format and tone.