Generative AI Handbook

1 Application Considerations

1. Prompt Management
    - Define a process for storing and testing several prompts. This is used to simplify the prompt evaluation process.
2. Evaluation Framework
    - Define key evaluation metrics
    - Define a process for calculating these metrics using variable prompts
3. Application Framework
    - Define if the application will leverage native APIs or external frameworks
4. Token Consumption
    - Define a process for calculating the number of input and output tokens. This is used for cost analysis, optimizing token constraints, and debugging.
5. Guardrails
    - Define a process for eliminating the impact of malicious prompts or exposure of sensitive data
6. General Topics
    - Can vector embeddings + similarity search improve the prompt context window (i.e. would RAG help)?
    - Is there a need to leverage a prescriptive or autonomous agent?
    - Are there free model alternatives, which meet security requirements, that can be used for benchmarking?

2 Developer and Solution Architect Notes

2.1 Rate Limits

2.1.1 Overview

When working with LLMs, rate limiting errors are very common. There are two distinct rate limits and it is important for developers to identify which rate limit the error relates to. One rate limit is requests per minute and the other is tokens per minute. These limits vary across provider and model version, and the model documentation states both of these values. Depending on the model, the requests per minute may be configurable up to the specified threshold. There might be a mechanism to increase this limit (ex: Provisioned Throughput for Bedrock), but this will likely involve a higher cost. The tokens per minute limit is most often not configurable.

2.1.2 Requests per Minute Limit Errors

To reduce the number of LLM requests, developers can batch requests or implement delays between subsequent requests. Developers should implement a process to track the number of requests sent, or identify this from the error message/API endpoint. This will allow them to determine the degree to which they need to limit the requests.

2.1.3 Tokens per Minute Limit Errors

To reduce the number of tokens consumed, developers can start by restricting the number of output tokens generated by the LLM. The model's request payload will have a parameter to specify this. Developers should implement a process to determine the number of input and output tokens consumed by a request (this could leverage an LLM API endpoint). The average token counts can be used to set the maximum number of output tokens.

2.2 Chunking

2.2.1 Overview

"Context stuffing" involves providing lots of data to an LLM and letting it figure out what to do with the data. Some LLMs struggle to extract relevant information when given large contexts, and thus the answer quality decreases and the risk of hallucination increases. These LLMs provide better results when given fewer, more relevant context windows.

The goal of chunking is to split content into smaller chunks that are semantically related, but have as little noise as possible. This simplest form of chunking is fixed-size chunking. This involves splitting the original content based on a fixed number of characters or tokens. The sizes of the resulting chunks will all be smaller than the fixed chunk. A more complicated form of chunking is recursive chunking. In this method, developers input a set of delimiters and a max size, and the algorithm will attempt to create chunks based on these delimiters, in addition to the max size. This hierarchical and iterative chunking process usually works better than fixed-size chunking.

2.2.2 Determining Chunk Size and Chunk Overlap

The two common chunking parameters are chunk size and chunk overlap. Developers should use the LLM's maximum input context window size as a starting point when determining these values. After knowing the upper bound on the context window, developers should decide the number of roughly disjoint pieces of information they want to include in the context window for a single LLM invocation, which will be the number of chunks in the context window. Chunks don't have to be disjoint, but developers can make this assumption when initially estimating the number of chunks. The default number of chunks is usually three, but developers should decide if they should deviate from this or simply use the default value as a starting point. If developers feel that the LLM will perform better with context that spans a larger number of disjoint segments, then they should use more than three chunks, and if not, they should use three or fewer chunks. Based on the number of chunks, developers can reverse engineer the approximate chunk size by using the maximum input context window size.

After performing this initial estimation, developers must also consider the minimum chunk size for their use case. The LLM will need sufficient context to provide an adequate response, so having a large number of small chunks may result in each chunk containing incomplete data. For example, the original data may consist of ten restaurant descriptions. One chunk distribution can have each restaurant description in its own chunk (ten chunks). Another chunk distribution can have half of each restaurant description in its own chunk (twenty chunks). This distribution might produce lower quality LLM responses than the first distribution since the LLM input context window will contain incomplete data if it doesn't include both chunks for each restaurant. For this reason, it is important for developers to strongly consider the minimum sufficient chunk size, and use this value to alter the number of chunks.

The chunk overlap specifies the number of tokens/characters overlapping between adjacent chunks. Developers should start with a relatively small overlap (20 tokens) and then gradually increase/decrease the value depending on the chunks generated. This value may also depend on the nature of the data. For instance, if the data is highly structured (ex: extracted from a table), it might not make sense to have any overlap across chunks.

Ultimately, developers should evaluate several chunk sizes and chunk overlaps to identify the parameters with the optimal chunks.

2.2.3 Parent-Child Document Retriever

RAG applications may benefit from using a Parent-Child Document Retriever, which utilizes a multi-level chunking strategy. RAG applications first perform vector similarity search, retrieve the chunks associated with the most similar vectors, and include these chunks in the LLM prompt. Vector embeddings account for semantics at a local level (adjacent words) and at a global level (entire paragraphs/pages). For this reason, vector embeddings for large chunks may include extra noise if the local semantic meaning differs from the global semantic meaning, which is likely the case. In general, similarity search performance is lower for large embeddings, which correspond to large chunk sizes. This behavior makes it challenging to determine the optimal chunk size, since similarity search performs better with smaller chunks, but the LLM prompt needs adequate context, so it requires larger chunks.

The Parent-Child Document Retriever addresses this challenge. Developers need to create two chunk sizes and two chunk overlaps: one for large chunks and one for small chunks. The original data is first split into large chunks, and then the large chunks are split into small chunks. The small chunks contain a reference to the large chunks they were derived from (ex: ID). The small chunks are used to create vector embeddings, which are used during similarity search. This component of the retriever (child) enables the application to realize the benefits of smaller chunks, which contain less noise. After the application retrieves the most similar small chunks, it obtains the corresponding large chunks, and includes these chunks in the LLM prompt. This component of the retriever (parent) enables the application to realize the benefits of larger chunks, which provide the necessary context to the LLM for it to generate an informed response. Since this retrieval method uses two levels, developers need to evaluate two sets of chunk sizes and chunk overlaps.

2.3 LangChain Development

When developing with the LangChain framework, it is recommended to pull down and extend the larger, more complex implementations. This is most notably the core classes (ex: AzureOpenAI, Agent). After pulling down the code from the relevant API version, developers can modify it for their use case. This simplifies extending LangChain's existing classes as well as the debugging process. This method can also be used to incorporate logging and extra error handling.

The most common alternative involves using LangChain directly out of the box. This reduces the application's flexibility since it can only support the functionality that LangChain supports. This approach also requires developers to stay up-to-date with LangChain's API. There have been multiple occasions where API updates result in breaking changes to code developed for older versions.

LangChain provides integrations with cloud providers. For example, LangChain provides support for Amazon Bedrock. When using these integrations, developers can create a client based on the cloud provider's SDK (ex: boto3 for AWS), and pass this client to LangChain. Alternatively, LangChain will setup this client itself. When developers explicitly create the client and pass it to LangChain, it generally results in fewer errors than letting LangChain initialize the client.

2.4 Prompt Management

Prompt management is essential to ensure developers can evaluate multiple prompts. It is common for developers to write one prompt, evaluate it, make a minor change to the prompt, and then re-evaluate it. This ad-hoc testing is fine for initial exploration and testing, but it usually is missing a form of prompt storage, so developers are forced to recreate past prompts if they want to go back to them. This increases development time as developers may spend extra time recreating previous prompts or retesting old prompts.

Solution architects and developers should create a prompt management process as early as possible. This will streamline the testing and evaluation processes. This process can be as simple as parameterizing the prompt template used by an LLM (or other GenAI tool). Then, developers can create several prompt templates and provide a unique identifier (ID,

filename, etc.) at runtime via a configuration parameter. Developers should store the output for each prompt along with its unique identifier. This allows developers to quickly and easily compare multiple prompts.

2.5 Evaluation Framework

An evaluation framework is essential to efficiently compare LLM architectures, specifically architecture approaches and all the parameters involved. The framework defines specific metrics used to evaluate the performance of the system. The framework interfaces closely with the application's prompt management logic. The framework should be used to iterate over several prompts as well as variations for key parameters (chunk size, similarity search method, re-ranking method, token limit, etc). The framework should calculate and output the metrics for each appropriate combination, which developers and solution architects can reference to identify the approach that works the best.