Note: While automated retraining is supported in this architecture, it isn't required, and caution must be taken in cases where it is implemented. It is inherently difficult to automate selecting the correct action to take from model monitoring alerts. For example, if data drift is observed, does it indicate that we should automatically retrain, or does it indicate that we should engineer additional features to encode some new signal in the data?

- **SCHEDULED**

  If fresh data are regularly made available, rerunning model training on a defined schedule can help models to keep up with changing trends and behavior.

- **TRIGGERED**

  If the monitoring pipeline can identify model performance issues and send alerts, it can additionally trigger retraining. For example, if the distribution of incoming data changes significantly or if the model performance degrades, automatic retraining and redeployment can boost model performance with minimal human intervention. This can be achieved through a SQL alert to check whether a metric is anomalous (e.g., check drift or model quality against a threshold). The alert can be configured to use a webhook destination, which can subsequently trigger the training workflow.

When the retraining pipeline or other ancillary pipelines themselves begin to exhibit performance issues, the data scientist may need to return to the development environment and resume experimentation to address such issues.

Implementing MLOps on Databricks

MLOps Stack provides a means of accelerating the creation of an MLOps workflow similar to the one outlined in the reference architecture section above. This repository provides a customizable stack for starting new ML projects on Databricks. After instantiating, a new project will have CI/CD pipelines and a number of example ML pipelines such as a model training pipeline, model deployment pipeline and batch inference pipeline, among others.

These pipelines are deployed to specified Databricks workspaces as Databricks Workflows using the Databricks CLI with Databricks asset bundles. Databricks asset bundles in particular enable the ability to programmatically validate, deploy and run Databricks Workflows such as Databricks jobs, and Delta Live Tables. Additionally it provides the ability to manage MLflow resources within Databricks such as MLflow experiments and MLflow models.

CHAPTER 6

LLMOps

With the recent rise of Generative AI we are prompted to consider how MLOps processes should be adapted to this new class of AI-powered applications. In this section we focus on the productionization of large language models (LLMs), the most prevalent form of Generative AI.

LLMs have splashed into the mainstream of business and news, and there is no doubt that they will continue to disrupt countless industries. In addition to bringing great potential, they present a new set of questions for MLOps:

- Is prompt engineering part of operations, and if so, what is needed?
- Since the "large" in "LLM" is an understatement, how do cost/performance trade-offs change?
- Is it better to use paid APIs or to fine-tune one's own model?
- ...and many more!

The good news is that "LLMOps" (MLOps for LLMs) is not that different from traditional MLOps. However, some parts of your MLOps platform and process may require changes, and your team will need to learn a mental model of how LLMs coexist alongside traditional ML in your operations.

In this section, we will explore how MLOps changes with the introduction of LLMs. We will discuss several key topics in detail, from prompt engineering and fine-tuning, to packaging and cost/performance trade-offs. We also provide a reference architecture diagram to illustrate what may change in your production environment.

What changes with LLMs?

For those not familiar with large language models (LLMs), see this summary for a quick introduction. In short, LLMs are a new class of natural language processing (NLP) models that have significantly surpassed their predecessors in size and performance across a variety of tasks, such as open-ended question answering, summarization and execution of near-arbitrary instructions.

From the perspective of MLOps, LLMs bring new requirements, with implications for MLOps practices and platforms. We briefly summarize key properties of LLMs and the implications for MLOps here, and we will delve into more detail in the next section.

KEY PROPERTIES OF LLMS	IMPLICATIONS FOR MLOPS
LLMs are available in many forms: ▪ Very general proprietary models behind paid APIs ▪ Open source models that vary from general to specific applications ▪ Custom models fine-tuned for specific applications	**Development process:** Projects often develop incrementally, starting from existing, third-party or open source models and ending with custom fine-tuned models.
Many LLMs take general natural language queries and instructions as input. Those queries can contain carefully engineered "prompts" to elicit the desired responses.	**Development process:** Designing text templates for querying LLMs is often an important part of developing new LLM pipelines. **Packaging ML artifacts:** Many LLM pipelines will use existing LLMs or LLM serving endpoints; the ML logic developed for those pipelines may focus on prompt templates, agents or "chains" instead of the model itself. The ML artifacts packaged and promoted to production may frequently be these pipelines, rather than models.
Many LLMs can be given prompts with examples and context, or additional information to help answer the query.	**Serving infrastructure:** When augmenting LLM queries with context, it is valuable to use previously uncommon tooling such as vector databases to search for relevant context.
LLMs are very large deep learning models, often ranging from gigabytes to hundreds of gigabytes.	**Serving infrastructure:** Many LLMs may require GPUs for real-time model serving. **Cost/performance trade-offs:** Since larger models require more computation and are thus more expensive to serve, techniques for reducing model size and computation may be required.
LLMs are hard to evaluate via traditional ML metrics since there is often no single "right" answer.	**Human feedback:** Since human feedback is essential for evaluating and testing LLMs, it must be incorporated more directly into the MLOps process, both for testing and monitoring and for future fine-tuning.

The list above may look long, but as we will see in the next section, many existing tools and processes only require small adjustments in order to adapt to these new requirements. Moreover, many aspects do not change:

- The separation of development, staging and production remains the same.
- Git version control and the MLflow Model Registry in Unity Catalog remain the primary conduits for promoting pipelines and models toward production.
- The Lakehouse architecture for managing data remains valid and essential for efficiency.
- Existing CI/CD infrastructure should not require changes.
- The modular structure of MLOps remains the same, with pipelines for model training, model inference, etc.

Key components of LLM-powered applications

In the following section we will delve into the more detailed aspects of building AI-powered applications using LLMs. The field of LLMOps is quickly evolving, however the following have emerged as key components and considerations to bear in mind. Some, but not necessarily all of the following components make up a single LLM-based application.

Prompt engineering

Prompt engineering is the practice of adjusting the text prompts given to an LLM to elicit more accurate or relevant responses. While it's a nascent field, several best practices are emerging. We will discuss tips and best practices and link to useful resources.

1. Prompts and prompt engineering are model-specific. A prompt given to two different models will generally not produce the same results. Similarly, prompt engineering tips do not apply to all models. In extreme cases, many LLMs have been fine-tuned for specific NLP tasks and do not require prompts. On the other hand, very general LLMs benefit greatly from carefully crafted prompts.
2. When approaching prompt engineering, go from simple to complex: track, templatize and automate.
   - Start by tracking queries and responses so that you can compare them and iterate to improve prompts. Existing tools such as MLflow provide tracking capabilities; see MLflow LLM Tracking for more details. Checking structured LLM pipeline code into version control also helps with prompt development, for git diffs allow you to review changes to prompts over time. Also see the section below on packaging model and pipelines for more information about tracking prompt versions.
   - Then, consider using tools for building prompt templates, especially if your prompts become complex. Newer LLM-specific tools such as LangChain and LlamaIndex provide such templates and more.
   - Finally, consider automating prompt engineering by replacing manual engineering with automated tuning. Prompt tuning turns prompt development into a data-driven process akin to hyperparameter tuning for traditional ML. The DSPy framework is a good example of a tool for both defining and automatically optimizing LLM pipelines.

Resources

There are lots of good resources about prompt engineering, especially for popular models and services:

→ DeepLearning.AI course on ChatGPT Prompt Engineering

→ DAIR.AI Prompt Engineering Guide

→ Best practices for prompt engineering with the OpenAI API

→ Replicate blog post - A guide to prompting Llama 2

3. Most prompt engineering tips currently published online are for ChatGPT, due to its immense popularity. Some of these generalize to other models as well. We will provide a few tips here:
   - Use clear, concise prompts, which may include an instruction, context (if needed), a user query or input, and a description of the desired output type or format.
   - Provide examples in your prompt ("few-shot learning") to help the LLM to understand what you want.
   - Tell the model how to behave, such as telling it to admit if it cannot answer a question.
   - Tell the model to think step-by-step or explain its reasoning.
   - f your prompt includes user input, use techniques to prevent prompt hacking, such as making it very clear which parts of the prompt correspond to your instruction vs. user input.

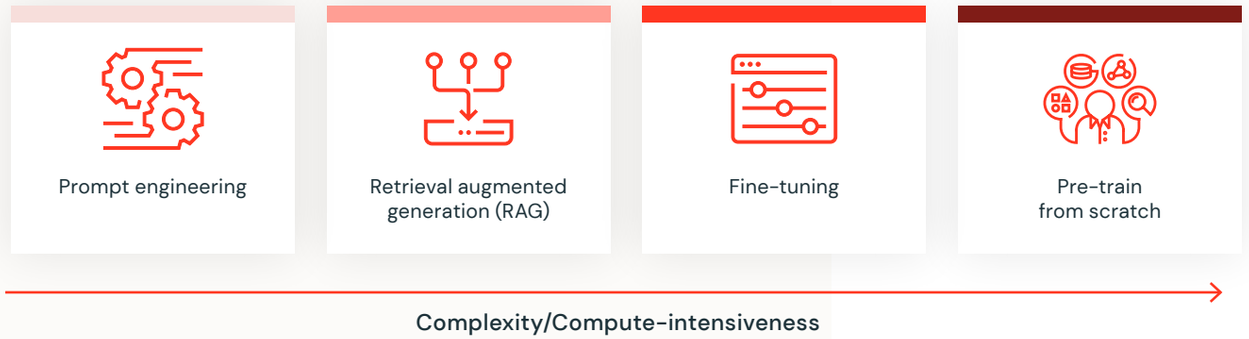
Leveraging your own data

While prompt engineering can yield remarkable results, it might not always suffice, especially in circumstances where domain-specific knowledge is required, or where contextual data is frequently updated. Incorporating your own data in LLM-powered applications can not only enhance a model's performance, but may also provide a strategic edge through customizing a model's output to your specific domain or use case requirements. Leveraging proprietary data can be the key differentiator in achieving superior results and gaining a competitive advantage.

We provide a high level overview of the various approaches that can be taken to leverage your own data with LLMs, and explore broadly when they should be used. In subsequent sections we will unpack these approaches in more detail.

Method	Definition	Primary use case	Data require-ments	Training time	Advantage	Considerations
Prompt engineering	Crafting specialized prompts to guide LLM behavior	Quick, on-the-fly model guidance	None	None	Fast, cost-effective, no training required	Less control than fine-tuning
Retrieval augmented generation (RAG)	Combining an LLM with external knowledge retrieval	Dynamic datasets & external knowledge	External knowledge base or database (e.g. vector database)	Moderate (e.g. computing embeddings)	Dynamically updated context, enhanced accuracy	Significantly increases prompt length and inference computation
Fine-tuning	Adapting a pre-trained LLM to specific datasets or domains	Domain or task specialization	Thousands of domain-specific or instruction examples	Moderate — long (depending on data size)	Granular control, high specialization	Requires labeled data, computational cost
Pre-training	Training an LLM from scratch	Unique tasks or domain-specific corpora	Large datasets (billions to trillions of tokens	Long (days to many weeks)	Maximum control, tailored for specific needs	Extremely resource-intensive

Note that the techniques outlined above are not mutually exclusive of one another. Rather, they can (and should) be combined to take advantage of the strengths of each. For instance, while you might fine-tune a model for a specific task, you can also employ prompt engineering to guide its response at inference time. Similarly, a pre-trained LLM can be further enhanced with RAG to dynamically fetch and incorporate external knowledge.



When considering which approach to take, like any ML application, it's crucial to assess your specific needs, business objectives, and constraints. A good rule of thumb is to start with the simplest approach possible, such as prompt engineering with a third-party LLM API, to establish a baseline. Once this baseline is in place, you can incrementally integrate more sophisticated strategies like RAG or fine-tuning to refine and optimize performance. Using standard MLOps tools such as MLflow is equally crucial in LLM applications to track performance over different approach iterations.

- The choice of technique will be informed by a range of factors, including (but not limited to):
- The volume and quality of your data
- Required application response time
- Computational resources available
- Budgetary constraints
- The specific domain or application at hand

Regardless of the technique selected, building a solution in a well-structured, modularized manner will ensure that you will be prepared to iterate and adapt as you uncover new insights and challenges. In the following sections we will look at each of these techniques to leverage your data, outlining the considerations and best practices associated with each.

Retrieval augmented generation (RAG)

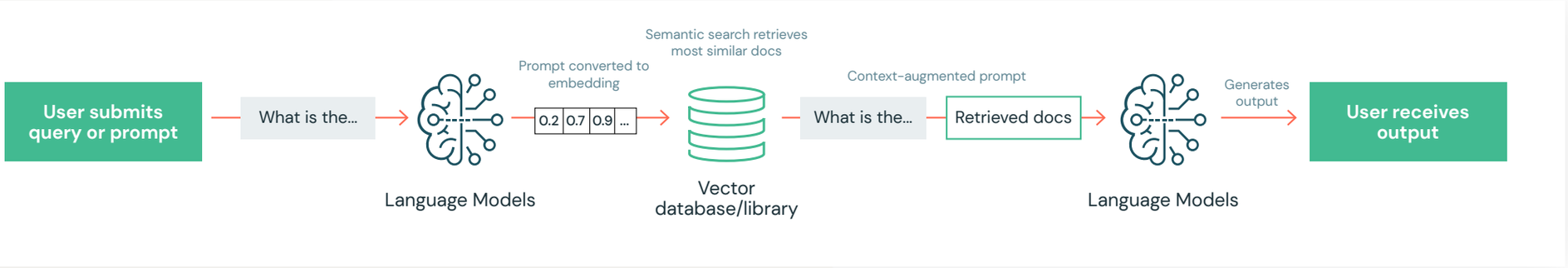
Retrieval augmented generation (RAG) offers a dynamic solution to a fundamental limitation of LLMs: their inability to access information beyond their training data cut-off point. RAG connects static LLMs with real-time data retrieval. Instead of relying solely on pre-trained knowledge, a RAG workflow pulls relevant information from external sources and injects it into the context provided to a model. RAG workflows are often used in document question-answering use cases where the answer might evolve over time, or come from up-to-date, domain-specific documents that were not part of the mode's original training data.

RAG provides a number of key benefits:

- **LLMs as reasoning engines:** RAG ensures that model responses are not just based on static training data. Rather, the model is used as a reasoning engine augmented with external data sources to provide responses that are up-to-date, accurate and relevant.
- **Reduce hallucinations:** By grounding the model's input on external knowledge, RAG attempts to mitigate the risk of producing "hallucinations" — instances where the model might generate inaccurate or fabricated information.
- **Domain-specific contextualization:** A RAG workflow can be tailored to interface with proprietary or domain-specific data. This ensures that the LLM's outputs are not only accurate but also contextually relevant, catering to specialized queries or domain-specific needs.
- **Efficiency and cost-effectiveness:** In use cases where the aim is to create a solution adapted to domain-specific knowledge, RAG offers an alternative to fine-tuning LLMs (discussed below) by enabling in-context learning without the overhead associated with traditional fine-tuning. This can be particularly beneficial in scenarios where models need to be frequently updated with new data. Note that where RAG reduces development time and cost, fine-tuning by contrast reduces inference time and cost.

TYPICAL RAG WORKFLOW

There are many ways to implement a RAG system, depending on specific needs and data nuances. Below we outline one commonly adopted workflow to provide a foundational understanding of the process.



Source: Databricks - Large Language Models: Application through Production

- **User prompt**
  The process begins with a user query or prompt. This input serves as the foundation for what the RAG system aims to retrieve.
- **Embedding conversion**
  The user's prompt is transformed into a high-dimensional vector (or embedding). This representation captures the semantic essence of the prompt and is used to search for relevant information in the database.
- **Information retrieval**
  Using the prompt's vector representation, RAG queries the external data sources or databases. A vector database (see below) can be particularly effective here, allowing for efficient and accurate data fetching based on similarity measures.
- **Context augmentation**
  The most relevant pieces of information are retrieved and concatenated to the initial prompt. This enriched context provides the LLM with supplemental information which can be used to produce a more informed response.

Resources

→ Databricks — Using MLflow AI Gateway and Llama 2 to Build Generative AI Apps

→ LangChain question-answer example

→ Eugene Yan — Patterns for Building LLM-based Systems & Products

- **Response Generation**
  With the augmented context, the language model processes the combined data (original prompt + retrieved information) and generates a contextually relevant response.
- **Feedback Loop**
  Some RAG implementations might encompass a multi-hop feedback mechanism (see the following paper for an example). In cases where the response is deemed unsatisfactory, the system can revisit its search criteria, tweak the context, or even refine its retrieval strategy, subsequently generating a new answer.

Vector Database

Retrieval augmented generation (RAG) hinges on the efficient retrieval of relevant data. At the heart of this retrieval process are embeddings — numerical representations of text data. To understand how RAG utilizes these vectors, let's first distinguish between a number of terms.

Vector index

A specialized data structure optimized to facilitate similarity search within a collection of vector embeddings.

Vector library

A tool to manage vector embeddings and conduct similarity searches. They predominantly:

- Operate on in-memory indexes.
- Focus solely on vector embeddings, often requiring a secondary storage mechanism for the actual data objects.
- Are typically immutable; post-index creation changes necessitate a complete rebuild of the index.

Examples of vector libraries include FAISS, Annoy, and ScaNN.

Vector database

Distinguished from vector libraries, vector databases:

- Store both the vector embeddings and the actual data objects, permitting combined vector searches with advanced filtering.
- Offer full CRUD (create, read, update, delete) operations, allowing dynamic adjustments without rebuilding the entire index.
- Are generally better suited for production-grade deployments due to their robustness and flexibility.

Examples of vector databases include Chroma, and Milvus.

In the RAG workflow described in the previous section we assume that the retrieved external data has been converted into embeddings. These embeddings are stored in a vector index, managed either through a vector library, or more holistically with a vector database. The choice between the two often hinges on specific requirements of the application, the volume of data, and the need for dynamic updates.

BENEFITS OF VECTOR DATABASES IN A RAG WORKFLOW

The following are a number of reasons why a vector database may be preferable over a vector library when implementing a RAG workflow:

- **Holistic data management:** Storing both vector embeddings and original data objects allows a RAG system to retrieve relevant context without needing to integrate with multiple systems.
- **Advanced filtering:** Beyond just similarity search, vector databases allow for application of filters on the stored data objects. This ensures more precise retrieval, enabling RAG to fetch contextually relevant and specific information based on both semantic similarity and metadata criteria.
- **Dynamic updates:** In fast-evolving domains, the ability to update the database without a complete rebuild of the vector index ensures that the language model accesses the up-to-date information.
- **Scalability:** Vector databases are designed to handle vast amounts of data, ensuring that as data grows, the RAG system remains efficient and responsive.