O'REILLY®

What Is LLMOps?

Large Language Models in Production

Abi Aryan

REPORT

What Is LLMOps?

Large Language Models in Production

Abi Aryan

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

What Is LLMOps?

by Abi Aryan

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

- Acquisitions Editor: Nicole Butterfield
- Development Editor: Sarah Grey
- Production Editor: Kristen Brown
- Copyeditor: nSight, Inc.
- Interior Designer: David Futato
- Cover Designer: Susan Thompson

- Illustrator: Kate Dullea

- May 2024: First Edition

Revision History for the First Edition

- 2024-05-03: First Release

978-1-098-15429-5

[LSI]

Introduction

Since 2023, LLM-based applications such as ChatGPT have taken the world by storm, convincing not hundreds or thousands but millions of users about the potential of AI in day-to-day life. This massive jump in the capability of machine learning models has not only led to the creation of several new tools, libraries and frameworks to operationalize LLMs but also presented us with several new open challenges that are unique to this new class of generative language models. This change is partly attributed to two factors: (1) the large size of these new language models, which can learn and store more patterns; and (2) the generative nature of these models, which allows them to generate information based on multiple static and dynamic sources, and personalize it to mimic human expression. As such, the adoption of these large language models (LLMs) has given rise to a new engineering discipline and framework, LLM Operations (LLMOps), which deals with the challenges and best practices for productionizing these large language models. In this high-level report, I will lay down the problems and challenges with LLM-based applications, define LLMOps, and go into how LLMOps guides these challenges.

Chapter 1. LLM Applications

A large language model (LLM) is a statistical model trained on large amounts of text data to emulate human speech for natural language processing tasks (Figure 1-1), such as information extraction, text classification, speech synthesis, summarization, and machine translation. LLMOps, thus, is a framework to automate and streamline *large language model* (also called a *foundational* or *generative AI* model) pipelines.

While task-specific models for natural language processing (NLP) have been used in practice for a while, recent advances in NLP have shifted public interest to more task-agnostic models that allow a single model to do all of the tasks listed in the preceding paragraph.

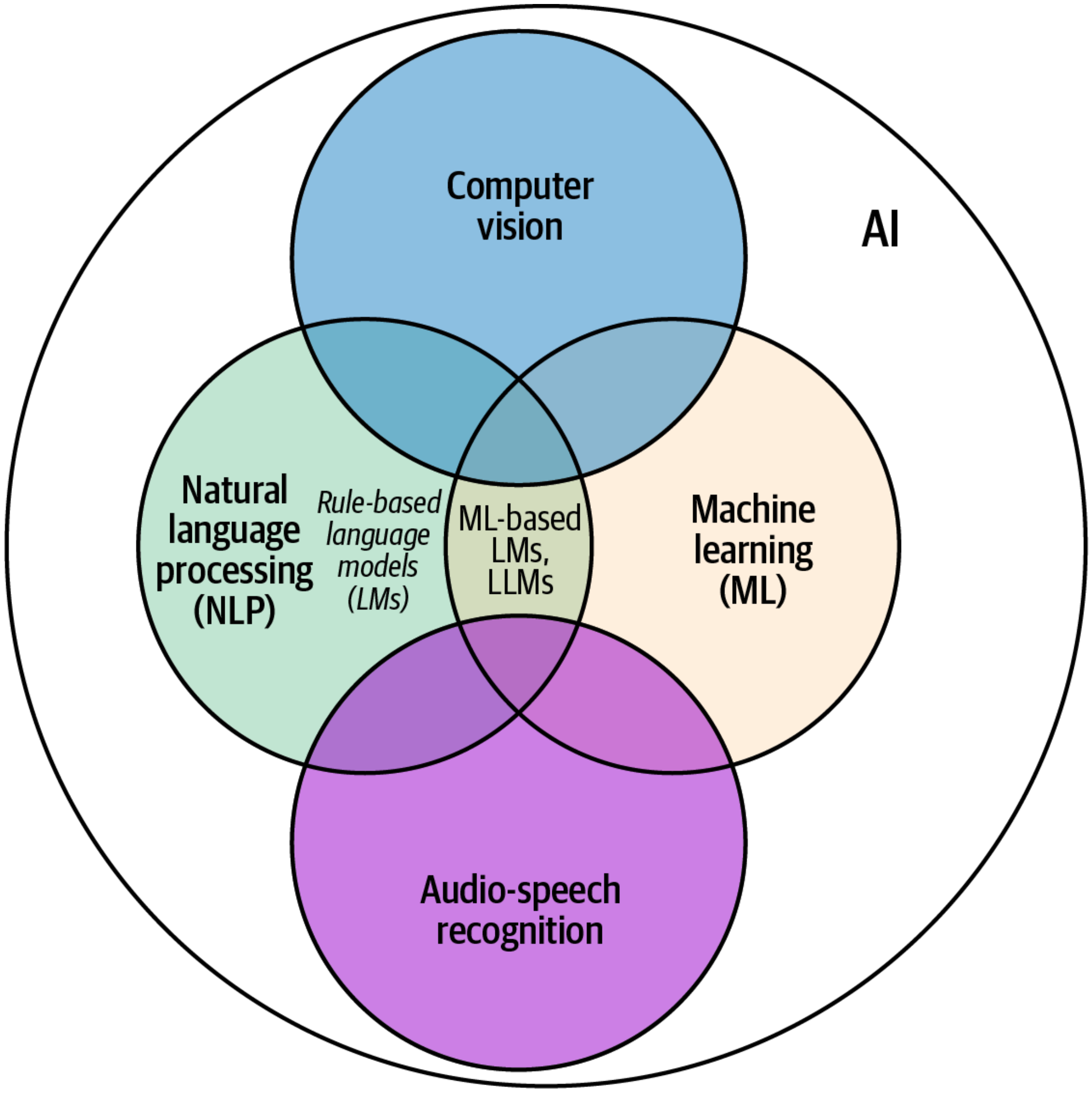

Figure 1-1. A Venn diagram explaining the correlation among AI, ML, and LLMs[1]

LLMs do this by using a large number of parameters (variables that store input-output patterns in the data to help the model make predictions): LLaMA, an LLM developed by Meta, contains 65 billion parameters; PaLM, by Google, has 540 billion; and GPT-4, developed by OpenAI, is estimated to have about 1.7 trillion. These parameters allow them to capture massive amounts of linguistic and contextual information about the world and perform well on a wide range of tasks.

As such, LLMs can be used in chatbots, coding assistance tools, and many other kinds of applications. Essentially, any information storage plus retrieval task that requires users to interact with a machine via text can employ an LLM. While some industries have been quick to adopt LLMs into their ML stack, others are reticent due to industry-specific challenges such as costs, evaluation, and regulatory compliance. We will look further into each of these problems in Chapter 3.

As I write this, technologists are working hard to integrate AI into every nook and cranny of the world around us—yet, less than 5% of the currently developed LLM prototypes have made it to production, based on an experimental survey.

Through this high-level report on LLMOps, we explore the potential and top five use-cases of LLMs in this first chapter. Chapter 2 introduces the LLMOps framework to deploy safe, scalable, and reliable LLM-based applications to production. Chapter 3 looks at the application of LLMOps through the entire LLM pipeline.

LLM Applications: Breakthrough or

Hype?

But first, let’s consider the question everyone is asking: Are LLMs truly innovative, or are they just hype?

AI and ML have been around for quite a few years now, and since the late 2010s and early 2020s, the engineering stack around them has developed quite a lot. However, until recently, building an ML or AI model required a lot of intensive expertise. In 2019, gathering task-relevant data to iteratively develop and then deploy models in production took a team of four experts weeks, if not months, of work. In 2024, however, any software engineer can develop an LLM-based app in an afternoon using just an API key, without any experience or knowledge of how to train an ML model.

This is a very real and very important change. McKinsey predicted in July 2023 that by 2030, 30% of United States workers’ working hours will be automated by generative AI. This includes 12 million jobs affected across industries, in roles such as customer service, law, writing services, data analytics, and even healthcare.

The Looming Transformation: Five Key

Applications of LLMs

It’s hard to imagine an industry that *won’t* be affected by generative AI and LLMs. The possibilities seem endless. LLMs are no longer applied only to text. They’re now being integrated with a vast variety of data types, like audio and images, and can quickly generate contextually rich responses.

This chapter looks at five top applications of LLMs:

- Knowledge retrieval
- Translation
- Audio-speech synthesis
- Recommender systems
- Autonomous agents

Knowledge Retrieval

Knowledge retrieval deals with querying and processing information to create structured, consistent output. Broadly speaking, it can be broken down into two subcategories: data retrieval systems and information retrieval systems. *Data retrieval systems* are commonly used in database management systems, whereas *information retrieval systems* are used in web search engines.

Since 1995, search engines have been the public's go-to method for discovering information online. However, with the exponential growth of data on the internet, ranking algorithms don't perform as well as they used to. Thus, knowledge retrieval and analysis is the most popular use case for LLMs, with offerings like ChatGPT, Bing Copilot, Google Gemini, Perplexity AI and You.com at the forefront of the race.

LLMs' personalized, conversational approach to information retrieval has radicalized users' search experience. Users can now set parameters for tonality ("Explain it to me like I'm five"), intent ("Analyze only the positive reviews of the game on this page and summarize them in one paragraph"), and format ("Give me a JavaScript version for this Python code").

On the data retrieval side, Google and Microsoft have integrated LLMs within their office software suites that allow users to ask questions in text documents and spreadsheets. This adds a new dimension to interacting with textual data that surpasses any previously available ML models or search engines.

Some LLM-based applications can now integrate with an organization's internal data and software systems to execute specific tasks, saving time and resources. Researchers are also exploring using LLMs as database administrators and to create knowledge graphs and better retrieval and ranking systems.

Translation

LLMs also show promising results on language translation tasks. Conventional ML-based translation engines often struggled to translate between languages with small datasets. It required a lot of data to teach an ML model about the statistical relationships between two languages. LLMs, on the other hand, are capable of *zero-shot translation* (translating without any prior examples) and *few-shot translation* (translating with very little data).

It has potential applications in literature, film, and music: imagine subtitling films or translating song lyrics into a new language without requiring large, labeled datasets for each language. In fact, several papers have provided empirical results showing that LLMs trained on multilingual datasets are better able to adapt to and perform in new languages than models trained on a wide variety of tasks in a single language. This opens up new possibilities for languages with sparse resources.

Programming

LLMs show remarkable performance at programming and coding, due to their capacity for context understanding and syntax awareness. Programming is a highly autoregressive task (i.e., it looks at the previous word to predict the next one), such that code completion is based on previous code elements in the sequence. LLMs, being naturally autoregressive, perform well at autocompleting code snippets to generate syntactically accurate code, drawing on patterns learned from vast repositories of code examples from GitHub or otherwise. These models are also language-agnostic, offering great flexibility for working across multiple programming languages.

LLMs' natural language capabilities allow them to learn *semantics*, the inherent and inferred meanings that connect English and various programming languages. This allows them to generate and debug code from natural language prompts. OpenAI's codex and code interpreter, which can generate code and charts using natural language queries only, has seen massive adoption, with 1 million new added users from October to December 2023.

Audio-Speech Synthesis

LLMs such as GPT-3 are effective for audio-speech synthesis due to their proficiency at understanding and generating human-like language. When trained on massive datasets of text and audio, they can even generalize and learn statistical relationships between text and sound. This allows them to generate more realistic transcriptions and captions than traditional speech synthesis systems, without being explicitly trained to do so. It makes LLMs well suited for a wide range of applications, from virtual assistants to narration, gaming, and even educational content. This means they can reduce development time and costs by automating several tasks, saving businesses time and money.

The large number of *embeddings* (mapping between words and phrases for the models to understand the relationship between different items of text and generate new responses) also allow the models to learn difficult nuances of human speech, such as intonation, rhythm, and stress. This would allow people around the world to get good results from speech recognition through home devices such as Siri and Alexa, without the need to explicitly train those models on massive accent datasets.

Recommender Systems

Research suggests that LLM-powered recommender systems can enhance and personalize the user experience for the ecommerce applications and aggregators. Because LLMs have access to multiple data sources including social media, online reviews, and blogs, and can retrieve context from user engagement history, product descriptions, etc., they can incorporate a user's preferences, recent activities, interests, and ongoing conversations to tailor real-time recommendations to that user's current needs and interests.

Recommender systems often struggle when given ambiguous queries or requests; LLMs, with their contextual understanding and ability to infer meaning from context, can navigate ambiguity more effectively, providing meaningful recommendations even with imprecise input.

Since LLMs can also process multimodal inputs, users can provide nontext data types as well. For example, not only can a user describe a product or service in natural language, they can also provide an image. The model can consider both modalities to generate more relevant recommendations. For example, “recommend shoes that go along well with this dress.”

Beyond text, audio, images, and video, LLMs can also process metadata such as time stamps, geolocation, and user engagement metrics to offer recommendations that consider the user’s temporal and spatial context. In some LLM-based recommender systems, users can even ask *why* the model has recommended a particular item, allowing the user to debug its chain of thought. This can help them generate better recommendations and establish trust with users by making the model’s logic more transparent.

Autonomous Agents

If you enjoy Marvel movies, you might be familiar with J.A.R.V.I.S. (Just a Rather Very Intelligent System), Tony Stark’s AI assistant in *Iron Man*. Engineers and organizations alike express a lot of interest in building autonomous agents along the lines of J.A.R.V.I.S. Any *autonomous AI agent* is an LLM-based application that can devise a plan for accomplishing a defined objective. It can communicate with other agents (also called *services*) through a process called *chaining* to execute that plan: this is known as the *planning/communication/action triad*.

Because LLMs can understand context and semantic nuances within language, they can discern relationships and dependencies between different elements of the data provided to them. This lets them understand the underlying components of multifaceted tasks and structure them systematically. When prompted, they can also explain their reasoning for each step. Most importantly, these applications can also be prompted to behave in a defined manner that's appropriate to a specific persona or job role. This provides them with an advantage in handling complex scenarios in procedural settings. Numerous companies have been exploring using AI agents to order food, make a travel booking, or interact with computers. They can even function as microservices.

As we dive deeper into the landscape of autonomous AI agents, understanding their architectural diversity and how to design an agent tailored to the use case specifically—whether it requires sequential decision making or batch-decision making—becomes the pivotal factor in choosing between action agents versus plan-and-execute agents.

Conclusion

These applications only scratch the surface of LLMs' potential as tools for encoding, processing, retrieving, and interacting with the massive amounts of data we generate by the minute. Their ability to process diverse sources, formats, and modalities makes LLMs incredibly versatile. Leveraging these models will allow organizations to increase the efficiency and generalizability of their ML systems. This is in part due to LLMs’ ability to process diverse sources, formats, and modalities, making LLMs incredibly versatile in their utility. This report sums up the landscape of LLM applications.

In Chapter 2, I will introduce you to the LLMOps framework and explain why we need it.

[1] Inspired by: Balaram Panda, “What Is LLM (Large Language Model)?”, *Medium* (blog), August 14, 2023.

Chapter 2. Introducing LLMOps

In June 2023, Nvidia CEO Jensen Huang told the world to "get ready for Software 3.0", in which humans and machines work together to create smart systems that can switch effortlessly between natural language and code. The way we write AI applications has changed. And LLMs aren't just doing the usual behind-the-scenes algorithmic functions like their older ML-model siblings—they're changing how we see and interact with software at scale.

It's a big shift from the Software 2.0 era, where data scientists and ML engineers collected tons of data and extensively feature engineered them to create in-house models to generate predictions and classifications. Now the LLMs are frontend stars, acting as both connectors and orchestrators for the massive information sources around us—both static sources (e.g., documents), and dynamic sources (e.g., websites and APIs).

Operationalizing LLMs

But any tech is only as good as the way you handle it. Without proper implementation and management, even the most advanced technology can falter and LLM-based applications are no exception. If you are doing LLM engineering, you may already be implementing some LLMOps principles, albeit ineffectively.

First, do you have a well-defined strategy encompassing new tools, design patterns, and operational practices to help ensure that everything runs smoothly and reliably and can scale up without having to reengineer the whole infrastructure stack? This approach is called *operationalizing*.

Second, how are you dealing with silos within the organization, so that everyone is on the same page and there are no unwelcome surprises, ensuring collaboration within the teams and the ability to troubleshoot, resolve, and postmortem problems quickly when issues arise?

And finally, how are you monitoring for indicators such as streamlined development process, efficient scaling capabilities, and proactive risk mitigation so as to ensure the application works smoothly and adapts to the user needs effectively in real time?

Here are a few signs that a team could benefit from LLMOps engineers:

- Your systems experience frequent outages or reliability issues.
- Your models are generating information that’s unreliable, unethical, or bad business practice.
- You are building large-scale, distributed systems with complex dependencies and frameworks that add needless complexity and introduce additional supply chain vulnerabilities.
- Your ML teams spend a significant amount of time on manual operations and repetitive work instead of churning out new features

and R&D milestones.

- You anticipate rapid growth or increased demand for your services in the future.
- You want to include cross-functional executives on strategic decisions for adopting or retiring a feature, model/model-provider, tools, and frameworks used by the organization.

To integrate LLMOps expertise in your team, you need a game plan.

If you want to make sure things are smooth, won't crash, and keep growing without incurring technical debt or financial liability, you need a system. This is where operationalizing kicks in—it's the secret sauce. With an operational framework in place, you're not just following the rules, you're tweaking things based on what works and what doesn't, keeping development in sync with what the company wants to do.

This is true for any new technology, in fact. The DevOps movement for managing IT operations and software development started around 2007 and is now common across organizations. From that sprang MLOps, around 2016, a framework for managing data collection and ML model life cycles. It quickly became clear that the companies successfully integrating ML into their products were often the ones using MLOps, and now, to deal with our resource-hungry, hard-to-evaluate language models requiring multidimensional abstractions, we need LLMOps.

So LLMOps isn’t just tech jargon; it’s a game plan that helps you develop and implement a systematic approach to operationalizing LLMs. It helps you decide which tools you need, what design patterns best fit your needs, and what cultural principles are important to you, so the LLMs can do their work efficiently without causing any chaos.

LLMs are no longer merely the parts of software applications as ML models but at the very core, serving as the connecting tissue that links to various facets and functionalities within and beyond the app ecosystem. To effectively leverage their capabilities, LLMOps strategies include not only the model life cycle but the broader application or product development life cycle. And since consumers expect LLM-based applications to have real-time or near-real-time latency no matter how popular they are, an LLMOps strategy helps you make sure your infrastructure can adapt to varying scales of operation.

As a discipline, LLMOps draws from DevOps and MLOps, but also from the fields of cybersecurity and user-interaction design. Its key goals are to make LLMs and LLM-based applications safe, scalable, and robust. In this chapter, I will look at each of these three goals in detail.

The goal of an LLMOps team is to integrate development, operations, and quality assurance as a cross-functional unit that is positioned between the product, data, engineering, security, and customer care teams. This LLMOps team aims to automate manual processes, mitigate risks, ensure regulatory compliance, optimize development workflows, and enhance the product and engineering processes through interactive feedback loops.

On a day-to-day basis, LLMOps teams work toward defining and achieving service level objectives (SLOs) that define the level of service an LLM-based system or application aims to provide to its users across different metrics such as availability, latency, throughput, error rate, data consistency, data freshness, capacity planning, response time, recovery time objective (RTO), etc.

LLMOps, as a discipline, is still in its infancy. As I write this in early 2024, there are very few mature tools and resources available for LLMOps teams. The skill sets required for operationalizing LLMs aren't as established, leaving a lot to be developed and figured out over the next few years.

Challenges

LLMs are ML models, but they come with their own unique challenges. Their size, memory requirements, and architectural complexity make it hard to optimize these systems. Doing so requires a systematic and specialized approach.

Furthermore, working with the ambiguities of natural language can be challenging even for human translators. It's important to evaluate the output for factual correctness and other issues. LLMs can generate inaccurate information (known as *hallucinations*) and biased material, and they can even violate copyright if their training data includes copyrighted material.

You need to monitor users’ experience and retrain your models if problems arise. Establishing a robust setup to monitor and retrain models is a big job that requires a strategic approach. Identifying potential issues ahead of time and optimizing your models can help you ensure a seamless user experience.

There are also significant costs to operating these massive models in production, including computation power and infrastructure. Without an operational framework, not only will you incur technical debt, but you’ll also run a cost deficit from developing noncompliant products.

Let’s look next at how to address these challenges.

The Goals of LLMOps

LLMOps has three key goals for optimizing LLM app deployments: safety, scalability, and robustness. This section examines each goal.

Safety

Safety is about making sure that your model plays by the rules. Imagine having this supersmart mammoth model doing its own thing and processing tons of information. You wouldn’t want it to go rogue or produce weird results, right? Safety includes making sure your LLM does not have access to users’ personally identifying information (PII) and can’t leak information from one user or one enterprise to another. It’s about how you deal with data handling, resource management, and the implementation of guardrails, and how you evaluate your model.

Deploying LLMs in production involves several kinds of safety risks. Three of the most critical are:

Compliance and regulatory risk

Organizations that break laws and regulations set by governments and other institutions (such as the European Union) can face large fines or other punitive measures. For example, the EU AI Act, which was undergoing final review at the time of writing, can enforce fines of up to 10 million euros or 2% of global profits, whichever is higher, on organizations that breach its rules regarding data privacy, security, and other AI-related issues.

Reputational risk

AI systems sometimes exhibit socially unacceptable behaviors, such as using slurs or making derogatory remarks. Such behavior in public or toward customers can do serious harm to users. This is not only bad in itself but can be terrible public relations—not just for the organization, but for the concept of AI and ML systems as a whole.

Operational risk

> Many organizations now use AI and ML tools to inform their operational decisions. If a model generates outputs that contain factual errors or hallucinations, executives could use that data to make erroneous operational decisions, like a bad investment, with costly outcomes.

An efficient LLMOps infrastructure accounts for all three types of risk.

One of the earliest attempts to deploy an LLM-based app was in 2016. Microsoft created an AI chatbot, trained it on clean data, and named it Tay. Tay was designed to engage with and learn from people on Twitter. But when “trolls” began interacting with it, Tay internalized their language. Within 16 hours, Tay started making racist remarks and engaging in other toxic behavior. Microsoft tried to fix the model, but it soon became clear that Tay was not safe to use. It was shut down within 24 hours. The experiment was a PR disaster for Microsoft. Even seven years later, in 2023, a *New York Times* op-ed mused how “Microsoft let its principles go”[1].

Tay’s little adventure is a clear example that it can be risky to deploy a powerful model at scale without a reliable framework for testing, evaluating, and monitoring its performance. Additionally, evaluations developed on clean data may not perform well on rogue data and needs to be reflected on in the data collection stage as well as the model-training/fine-tuning process.

Now, imagine the risks associated with incorrect, false and made-up information in output from an LLM model used for strategic decision making or in legal research. That’s why it’s so important to have a strategy in place, as well as a reliable framework for testing, evaluating, and monitoring the performance of LLM applications at scale. LLMOps does exactly that.

Scalability

Scalability is like the superhero that helps your LLM handle the big leagues. If a model service goes down, no problem—you can switch to another one seamlessly. Getting more traffic than usual? The load balancer is already integrated and tested and its costs are already being monitored.

With customer-facing models, there is no forgiveness when it comes to inference speed, latency, and throughput. Can your application handle the pressure and integrate with another service quickly without major engineering time and effort? Scalability is about making sure that your data warehouses and models don’t freeze up or crash when demand skyrockets. Optimizing for scalability is an important part of LLMOps. Not doing so can lead to performance degradation, insufficient resource utilization, application crashing or becoming unresponsive under heavy loads, thus disrupting service availability etc. These can be addressed using distributed computing microservices architecture, caching and optimization, load testing and performance testing, etc., which are some of the key considerations for LLMOps engineers.

Robustness

Robustness means that your model‘s behavior and performance stay consistent and reliable over time. This means developing model checkpoints and having a rollback strategy in place in case the model’s performance starts to degrade quickly. This becomes especially important if you are using a proprietary model like GPT-4, but also if you are hosting your own model. In a 2023 paper, researchers studied how ChatGPT’s performance degraded after its launch. Since most model providers have relatively opaque update processes, this kind of degradation (also known as drift) is a big concern.

Robustness is about making sure that your LLM-based applications remain performant. You don’t want the model playing hide-and-seek or giving you a dreadful “poem” because the server is down. Thus, monitoring for robustness comes down to four basic factors:

Data drift

Data drift is when the statistical properties of the input data change over time, leading to a decrease in performance. This can be caused by changes in user behavior, the data source, or the environment. For example, if a company’s customer demographics change quickly, its historical data may slowly become less relevant to the organization. Overcoming this requires a robust data-management system that can detect and correct data drift.

Concept drift

Concept drift is when a model’s performance degrades over time due to changes in learned features. For example, before the COVID-19 pandemic, the word *Corona* was most widely associated with a beer brand. Today, it’s just as likely to refer to the novel coronavirus. Advanced retrieval-augmented generation (RAG) pipelines (see “Step 4: Domain Adaptation” in Chapter 3) can be highly effective for automatically updating your model based on recent news events and happenings from the internet past the model’s cutoff date (the point in time at which model’s training data ends, which means the responses are based on knowledge available up to that date) without having to retrain it from scratch.

Prompt drift

Prompt drift is when a model’s behavior changes due to updates in the fine-tuning or retraining processes. In other words, prompt drift manifests as a shift in the model’s understanding and generation of responses, which may deviate from its previous behavior. This usually occurs due to changes in data distribution, introduction of new training examples, bias, or updates/adjustments to hyperparameters or modification of the training objective. During early deployments using proprietary models like GPT-3, companies that developed elaborate prompt pipelines (sequence of prompts) suffered from performance drift in their applications when the model was updated. Addressing prompt drifts through constant monitoring, testing, and retraining is essential to maintain the reliability and consistency of LLMs, especially with dynamic models.

Unreliable dependencies

LLMs often depend on external libraries, services, data sources, APIs, and so on. However, the integration of such dependencies introduces complexities, vulnerabilities, and potential points of failure into the model’s ecosystem. To navigate these challenges effectively, adopting robust infrastructure and management practices becomes paramount. Using containerized deployments, version systems, and dependency-management tools such as MLflow can help with experiment tracking and reproducibility, as well as keeping the configurations consistent and updated.

The Model Life Cycle

LLMOps provides a strategic and thoughtful way to develop LLM-based applications through all three stages of the model life cycle: development, deployment, and serving.

Development

In the *development* phase of the model life cycle, LLMOps provides a structured approach to the iterative process of data curation, model training/fine-tuning and evaluation to achieve optimal performance. On the technical side, LLMOps teams employ a range of tools and techniques to streamline the development process and increase developer productivity. This may include version control systems for tracking changes to the model code and configuration; feature stores for storing prompts and cache; and using code reviews, tests, and automated continuous integration/continuous deployment (CI/CD) to maintain code quality and accelerate the development cycle.

One of the key differentiators of strong LLMOps teams is the use of a comprehensive evaluation tracking, horizontal scaling, and data management platform so as to optimize training costs, track performance, and maintain inner and outer model alignment.

Deployment

As you reach the *deployment* stage, LLMOps can help you choose a deployment strategy. You’ll also need to deal with the challenges of integration, including data integration and data flywheel, load balancing, security, API design, versioning, monitoring and logging, error handling and recovery, resource allocation, integration testing, documentation, compliance, failover and redundancy, performance optimization, and cost management.

Serving

At the *serving* stage, you need to determine how you'll deliver your application or service to users or other systems. LLMs can use a monolithic architecture (one large, unified model, such as GPT-3) or a microservices-based serving architecture (a collection of smaller, independent deployable services, such as Gemini). Which you choose should be based on your use case and scaling decisions. You can also use a *multinode* deployment to run multiple instances of your app on separate nodes or servers. This comes down to infrastructure design decisions about how to use GPU resources efficiently, how to perform maintenance and updates, and whether you intend to use different technology stacks for different services.

Conclusion

In closing, LLMOps involves overcoming software challenges but also strategically considers all the choices and challenges with the model life cycle. From development to deployment to serving, each stage presents unique choices and challenges that carry significant implications for costs, infrastructure design, maintenance, and scalability. While going into the complete landscape of LLMOps decision and design patterns is beyond the scope of this report, in Chapter 3, we will look into the application of LLMOps specifically through the LLM life cycle.

[1] Reid Blackman, “History May Wonder Why Microsoft Let Its Principles Go for a Creepy, Clingy Bot,” Opinion, *New York Times*, February 23, 2023.

Chapter 3. Automating the LLM Life Cycle

Developing and deploying an LLM-based application is an iterative endeavor and a practice of constant experimentation that can lead to significant technical debt. Across the model life cycle, LLM teams create several versions of their data, model, and pipeline stack. Not all of those changes get documented. And then there are the tools and dependencies! Across the model pipeline, there are some similarities with operationalizing deterministic ML models (aka MLOps); however, there are several new challenges as well. Namely, where LLMOps differs substantially from MLOps is that these models are generative in nature, which can make evaluating and debugging your model's performance much harder. Additionally, conventional feature engineering is no longer relevant for large language models and given the large size of these models, a big focus for LLM teams is performance optimization, which includes dealing with data and model parallelism complexity, load imbalance, memory and resource management, etc.

LLMOps helps automate and streamline processes to resolve data communication and synchronization challenges across the entire LLM life cycle so that teams can prototype models quickly, consistently, and effectively in the production environment.

This chapter goes through the eight steps in the LLM life cycle, pictured in Figure 3-1:

1. Data engineering
2. Pretraining
3. Base model selection
4. Domain adaptation: prompt engineering, RAG, and fine-tuning
5. Model evaluation
6. Integration and orchestration: CI/CD
7. Security and reliability engineering
8. Deployment and model monitoring

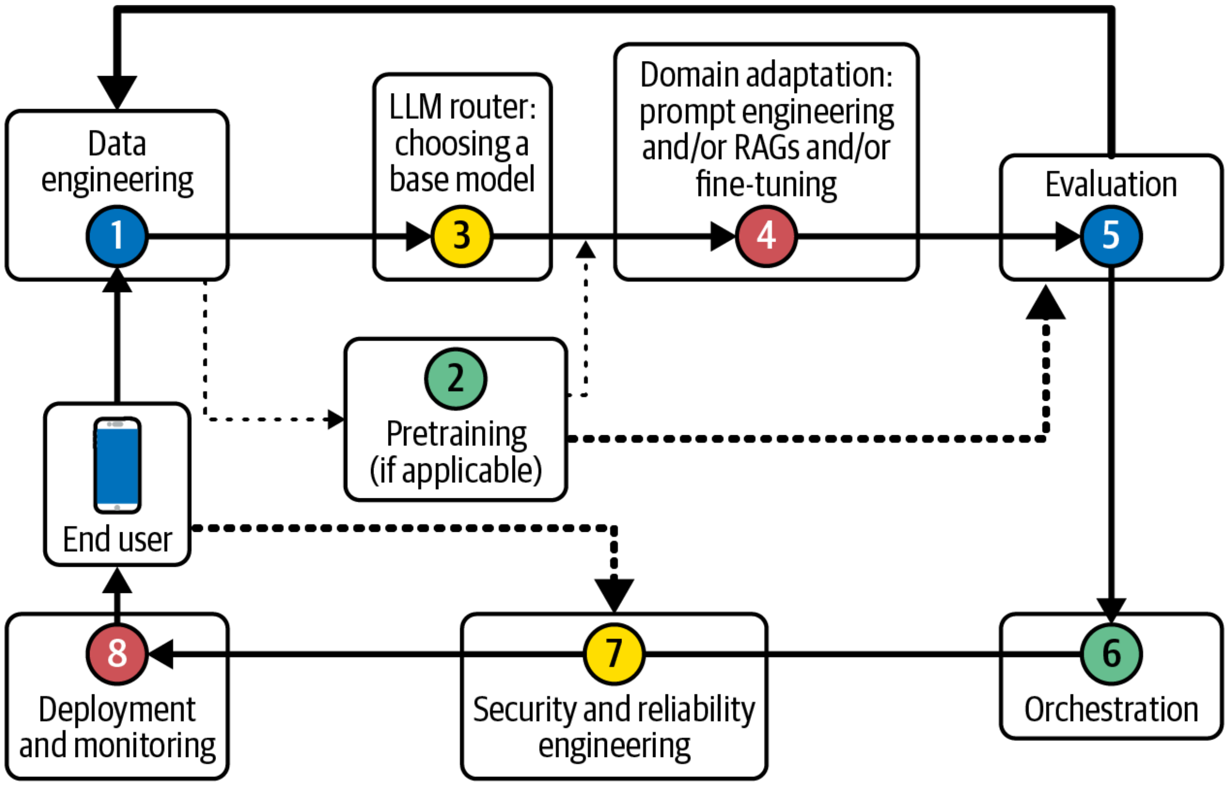
Data engineering
1
LLM router: choosing a base model
3
Domain adaptation: prompt engineering and/or RAGs and/or fine-tuning
4
Evaluation
5
2
Pretraining (if applicable)
End user
8
Deployment and monitoring
7
Security and reliability engineering
6
Orchestration

Figure 3-1. The LLM life cycle

Adopting LLMOps principles helps guide LLM teams in automating and streamlining the large language model life cycle at every step. Using automation tools, CI/CD practices, and systematic orchestration can improve safety, scalability, and robustness of large language models in production. It reduces chaos and ad hoc practices, thus constantly adapting, adopting, and automating the best practices.

Step 1: Data Engineering

Data engineering for LLMs involves the design, development, and management of data pipelines and infrastructure to support the training, evaluation, and deployment of these models.

Regardless of whether you want to use the base model as is or do further training to adapt it to the business/domain, this remains a pivotal part in performance optimization for LLMs. For organizations that need domain adaptation, acquiring access to data is a pivotal aspect of LLM model development. LLMs' performance is influenced not only by the quantity of data but also its quality: as the popular ML adage goes, "garbage in, garbage out."

The following section looks at some things to consider as you seek out the right training datasets for your LLM.

Data Collection

If you're collecting data to train your LLM, define what you want the model to do. Will it power a question-answering application? An email-writing assistant? A programming buddy?

Next, determine the data sources available to you, paying careful attention to each source's policies regarding data scraping, privacy, request limits, and legal use. Depending on the task, your options may include:

- Publicly available datasets in your domain from Hugging Face, Kaggle, or Google Datasets
- Web scraping from websites, forums, and other online platforms
- Internal data within your company, such as documentation and code
- Customer interaction data such as customer chats, web forms, etc.

The goal here is to automate collecting new data from multiple sources, integrating it into the model, and ensuring a continuous flow of recent and relevant data. Good LLMOps practices require automating data ingestion using tools like Airbyte, Striim, or Voltron Data.

The LLMOps stack also includes tools such as Data-Juicer, which checks whether a dataset is heterogeneous enough to reduce bias and overfitting. If you need a synthetic dataset to work with, Faker is a useful tool for generating such data.

Data Preprocessing

Preprocessing raw data means cleaning, structuring, and preparing it before it’s used for training. It ensures that the model learns from high-quality, optimized datasets. Tasks in the data processing pipeline include:

Noise management

Identifying and removing irrelevant and low-quality data, using tools such as Unstructured

Data augmentation

Enhancing the diversity of the data

Tokenization

Breaking text down into manageable chunks for the LLM

Toxicity and bias mitigation

Identifying toxic or biased content

Input prompt regulation

Automating regulation of input prompts, ensuring consistency and preventing SQL injection (SQLi) attacks

Deduplication

Removing duplicate entries so that the model learns from unique examples

Data sanitization

Permanently removing PII and sensitive information from storage to maintain data security and privacy

By integrating these preprocessing strategies into an automated data pipeline through LLMOps, organizations can achieve consistency, efficiency, and reliability and reduce risk in training LLMs.

Data Storage

Data storage is one of the most critical components in the data engineering stage of the LLM life cycle. While you can use a nonrelational database (e.g., MongoDB) or a graph database (e.g., GraphQL) to store embeddings, vector databases such as Faiss or Chroma particularly stand out for their use in many successful LLM applications. They don't just store or index but are optimized to work with vector embeddings and can also integrate easily with your existing stack. This article summarizes some of the leading offerings.

Factors to consider in choosing a database include performance, ease of local use for development, managed cloud options, user interface, and the requirements of your use case.

Building data-processing pipelines can facilitate and guide the flow of information between vector databases and your model, so you can harness different data sources in a coherent and orchestrated manner.

Data Management

Data management encompasses cataloging, versioning, and governing data.

Data catalogs systematically capture and manage the metadata associated with each dataset, such as data source locations and names, formats, quality, and usage. Since LLMs are such complex models trained on very large amounts of data, data catalogs are important for debugging, retraining tasks, and maintaining data integrity. They also facilitate collaborative data exploration, discoverability, and lineage tracking, to help you understand how data flows through the models. This helps to create accountability, transparency, and traceability in the system. Secoda AI is one tool automating data cataloging as a service for LLM developers in enterprises.

Synchronizing different versions of your model with the corresponding versions of the training data helps ensure consistency, makes rollbacks easier, and mitigates the issues that can arise from changes to datasets, such as inconsistent outputs and reproducibility problems. Data-versioning tools in the LLMOps stack include DVC and Neptune.

Step 2: Pretraining

This step applies mainly to those interested in developing and training their own models in house.

This section will explore some of the key ways that LLMOps streamlines the pretraining process, covering versionized tokenizers, model versioning, and checkpointing. It also looks at some tools to automate this stage of the pipeline.

Tokenization

Tokenization is a fundamental step in processing text data into smaller chunks for LLMs.

When you supply a sentence to an LLM-based tool, a tokenizer breaks that sentence down into small *tokens*, which can be words, phrases, or even individual letters. The patterns in sequences of tokens can then be learned by the LLM so that it can successfully predict the next word in a sequence or even generate an entire cohesive paragraph.

Versionized tokenizers (like Hugging Face's Tokenizer library) ensure consistency and reproducibility: the same input text consistently produces the same output text. This lets LLM engineers maintain precise control over the tokenization process across different model versions and iterations.

Checkpointing

Checkpointing is the process of saving intermediate states of the model during training. If there is a system failure during training, or any interruption that causes the model to lose its progress, saving a checkpoint allows LLM engineers to resume training from a specific point. In LLMOps, checkpointing is vital for large-scale LLM training, which can take days or weeks. Tools like DeepSpeed offer efficient LLM checkpointing solutions.

Model Versioning and Logging

LLMs are trained in stages, with engineers evaluating each version and then retraining the model. Keeping track of these versions is important for tracing, debugging, reproducibility, and deployment. Frameworks like MLflow let you record and manage different model versions and create automated logs for each run of the model.

Using tools to automate versioning, checkpointing, and other aspects of the development process is important for maintaining a smooth and reproducible model-training workflow.

Step 3: Choosing a Base Model

Building and maintaining an in-house LLM in production requires a significant investment in infrastructure, data collection, and personnel, in addition to ongoing operational costs. Thus, most organizations instead choose to build on a *pretrained model* (also called a *base model* or *foundation model*): a version of the LLM that comes pretrained on large and diverse textual datasets. You'll need to decide whether to use an open source model or a proprietary model (e.g., GPT-4).

Open source models offer:

Customizability and control

Open source LLMs offer users with flexibility to customize the model architecture, training data and model parameters, thus allowing the users full control over the model's behavior. Also, open source models enable organizations to avoid vendor lock-in and maintain full control of their technology stack.

Transparency and security

Open source LLMs offer full transparency into the training data as well as model code, allowing users to identify potential vulnerabilities, biases, or ethical concerns, thus allowing them to improve the model's security.

License considerations

When adopting an open source LLM, organizations must consider the licensing terms and conditions associated with the model's code and dependencies. Not all open source models are equally permissive. A model released under the GNU General Public License (GPL) requires the users to release changes to the distributed source code under a GPL license, whereas the BSD-2-Clause or MIT license is far more permissive that allows one to use the code by simply including license text in distribution.

Cost optimization

Open source LLMs can significantly reduce the upfront licensing costs and ongoing subscription fees associated with the proprietary models such as OpenAI's GPT or Anthropic's Claude.

Whereas proprietary models offer:

Scalable cost structure

Proprietary LLMs allow organizations to scale their investment based on factors such as the volume of data processed and the number of active users, thus reducing significant upfront costs. These proprietary models, depending on the provider, may offer tiered pricing or usage-based billing month by month to accommodate varying needs and budgets.

Monitoring and maintenance

> These model providers often monitor smooth operation and performance decline by monitoring for performance metrics, troubleshooting technical issues, software updates, and security patches. This allows organizations using proprietary models to focus on the business instead of day-to-day technical issues.

Robust powerhouses

> These models are often developed by leading technology companies that invest heavily in research, development, and infrastructure to create high-performance models that deliver safety, scalability, and robustness as compared to their open-source alternatives.

Function templates

> Some proprietary LLMs may offer prebuilt function templates or APIs that simplify integration and deployment into existing workflows and applications, allowing organizations to use templated solutions accelerating time to market.

As with any technology, choosing a base model comes with several trade-offs. I strongly recommend you consider the following questions as you make your decision:

- Do you have access to the training data?
- How diverse and representative is the training data?

- Has the model been tested on a range of scenarios to gauge its robustness?
- What measures have been taken to address biases in the model's responses?
- How does the model handle out-of-distribution inputs?
- Is the model's architecture transparent and interpretable?

Depending on whether you choose a proprietary model, an open source model, or a combination, different software architecture styles come into play—and so do their trade-offs. Proprietary models may pose risks such as high load and potential service outages, whereas open source models may very quickly suffer from drift and performance degradation.

If you choose to build on a proprietary model, a centralized or microservices-based architecture might be a good choice for your organization. If you build on an open source model, you may prefer containerization, such as Docker, to streamline deployment, and thus a serverless, hybrid API, or event-driven architecture that can scale responsively in scenarios with varying workloads.

LLMOps can guide you in choosing software design patterns and architecture choices for base models, implementing a routing solution to leverage different specialized LLMs depending on your application's scalability needs, monitoring budget, cost flexibility, and security priorities.

Step 4: Domain Adaptation

While pretrained models such as GPT-3 and GPT-4 possess a general understanding of language and perform well on tasks like coding, they might not be directly usable for tasks in more specific domains, for example healthcare or financial questionnaire. Fine-tuning, RAG, and prompt engineering are three methods for bridging that gap.

Prompt Engineering

A useful method for improving performance when you don't have access to a lot of data is *prompt engineering*, the process of designing and creating clear, concise prompts (instructions) that are easy for the LLM to understand. Instead of giving the LLM hundreds of input data points, you provide the model with around 10 examples of the task and a blueprint for handling these tasks, allowing the model to produce a desired output.

There are three steps to prompt engineering:

1. First, *prompt selection*: choosing the prompt that is most likely to generate the desired output. You can do this task manually or automate it. There are several techniques for selecting prompts automatically.
2. Second, *prompt tuning*: prompt tuning is a way of teaching an old dog new tricks. It's a technique to help the model update the parameters stored in its embedding layer without having to retrain the entire model

from scratch. There are a few approaches. This can be done manually or automated; I recommend the reinforcement-learning library RLPrompt.

3. Finally, *logging* and *tracing* aren't limited to inputs and outputs. Tracing tools allow you to save contextual logs with detailed information in rich formats, capturing prompt variations, time stamps, model predictions, and more. These lots are useful for user-journey mapping as well as debugging and auditing.

Automated tracing and monitoring tools such as Weave by Weights & Biases (WandB) and PromptLayer are an excellent starting point, but far from mature. As such, most LLM teams develop in-house tools and pipelines to manage their prompt-tuning automations. LLMOps techniques for prompt-tuning include automating user feedback, sentiment analysis monitoring, dynamic prompt adjustment, iterative model updates, establishing audit trails, and implementing access controls for sensitive logs.

Retrieval-Augmented Generation

RAG is an approach for enhancing LLMs' capability on question-answering tasks. RAGs allow users to connect a model with external knowledge sources. This can make its predictions more accurate and context-aware. You can also use RAG to feed new data into the model without having to retrain it. RAG pipelines within the LLMOps framework involves orchestrating, monitoring, and autoscaling a series of data warehouses, vector retrieval, and generative components together.

RAG is particularly useful in cases of concept drift (e.g., updates in political scenes so the model can be more accurate instead of producing factually incorrect information). For instance, a model experiencing concept drift might answer the prompt:

```
> Who is the president of the United States?
```

with

```
>> As of my training cutoff in 2016, Barack Obama
   president of the United States.
```

This also extends to context windows (text input/output range to generate a response and comprehend the context of the query) as well where the short-term memory loss between different context windows will drift to the point of amnesia. LLMs perform token and sequence prediction and don’t necessarily utilize representational models of the data they have consumed. This is why it can generate text that seems plausible but is factually, logically, or semantically incorrect, also known as hallucination, as discussed in Chapter 2.

It is important in RAG applications to consider metrics such as results "relevancy" as key in determining solution performance. Building test harnesses that validate solution performance against these and other metrics are an important part of LLMOps and highlight one aspect of moving from a demo or proof of concept into production. One can also improve model performance by implementing rerankers in the RAG pipeline (see Figure 3-2). Reranking is one of the simplest methods for dramatically improving recall performance (meaning, "how many of the relevant documents are we retrieving"). Rerankers avoid the information loss of bi-encoders, but they come with a different penalty—time.

Frameworks such as Semantic Kernel, LlamaIndex, and Haystack have made significant progress in making RAG pipelines easy to use and have introduced more abstractions, such as query engines.

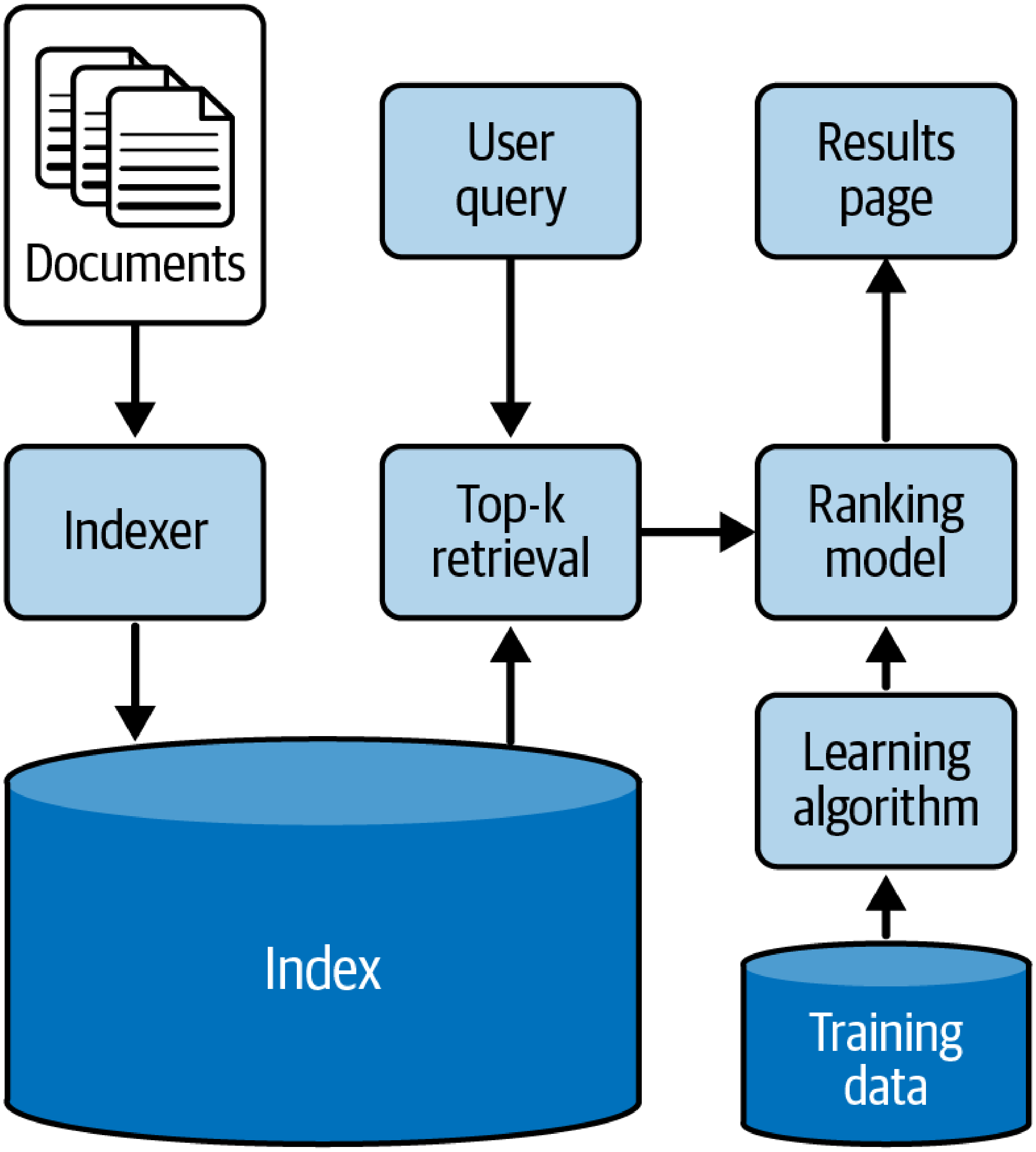

Figure 3-2. How RAG pipelines work in ML (adapted from Wikipedia)

Fine-Tuning

Fine-tuning is the process of giving a pretrained LLM further training for some specific task or problem domain. Fine-tuning a model is less computationally intensive than training a model from scratch, making it more resource-efficient and faster to get your model to understand and perform well for your use case. It also provides an opportunity to work on any security vulnerabilities or improve data heterogeneity, especially if you are using an open source, pretrained LLM.

Fine-tuning gives you more control over your model's behavior and output than prompt tuning and RAGs. Fine-tuning works by updating the parameters as you retrain the LLM on new, domain-specific data.

Fine-tuning methods include:

Adapter-based fine-tuning

Useful for changing the model's inherent behavior

Prefix tuning

Useful for task-specific training

Parameter-efficient fine-tuning (PEFT)

Updates only *some* model parameters, optimizing for memory usage, computational costs, and speed

Full fine-tuning

Updates *all* the model’s parameters (very computationally expensive)

Fine-tuning pipelines look a lot like pretraining pipelines: they share similar principles, tools, and best practices. LLMOps for fine-tuning encompasses automated data versioning, model checkpointing, hyperparameter tuning, experiment tracking, parallelization, and distributed computing tools and techniques, making sure that the pipelines can be automated and monitored without manual intervention.

Step 5: Model Evaluation

Evaluation is the most critical step in LLM development—and often the hardest. In fact, evaluating LLMs requires a whole new testing paradigm, and as of early 2024, its tools and techniques aren’t quite there yet. LLMs are so large and their architectures are so complex that, along with being generative in nature (which makes them hard to evaluate), ground truth or good metrics to act as proxies for performance often require a lot more thought.

Evaluation Challenges

Given their use across such varied domains and use cases, LLMs require a diverse set of evaluation metrics. You need to test not only your model’s in-context learning abilities, but also screen for out-of-context learning, such as bias and toxicity.

LLMs also sometimes hallucinate or produce factually incorrect information. This is an example of *silent failure*, and it can be very hard to detect without integrating fact-checking, cross-referencing, and/or human evaluators and moderators into the evaluation pipeline.

Human language is also complex and often subjective, making it very hard to quantify what constitutes “good enough” results. User feedback is important for this and for better personalization, but integrating this feedback into the evaluation and the fine-tuning process can be cumbersome and expensive in terms of time and resources.

Tools and Guardrails

Putting enough tracing and monitoring pipelines in place to evaluate your model’s performance across large datasets and many user interactions is particularly crucial as you deploy it at scale. LLMOps best practices support the integration of predeveloped and custom evaluation metrics and guardrails in the training pipeline to automate and streamline this process.

Some of the available tools include metric-based evaluation functions:

- Recall-Oriented Understudy for Gisting Evaluation (ROGUE)
- Bilingual Evaluation Understudy (BLEU) scores

- General Language Understanding Evaluation (GLUE)
- SuperGLUE benchmarks
- Big Bench
- OpenLLM
- Stanford HELM

Tool-based evaluations that can catch out-of-domain interactions, system failures, security (see "Step 7: Security and Reliability Engineering"), loggers, and the like include:

- Log10
- HoneyHive
- Traces by WandB
- Guardrails

Model-based evaluations use adversarial models for task-specific evaluations, while human-in-the-loop evaluators can deal with alerts and provide feedback, which can be integrated into and further enrich the retraining process. To optimize our LLMOps life cycle, a crucial principle involves automating and orchestrating these evaluators as essential components within our CI/CD pipelines.

Step 6: Integration and Orchestration

Integration and orchestration frameworks are tools for managing how data moves between LLM-based applications. LLM applications often involve a lot of consumer interaction. This means you need fast, dynamic ways to optimize response times, minimize latency, and deal with errors—faster and more dynamic than any standard ranking or prediction algorithm.

Integration refers to connecting components, tools, systems, and code to work together as a single, unified solution. For LLMOps, this includes integrating the user interface, data sources, different base models, RAG pipelines, loggers, and deployment environments to facilitate seamless data flow. Some of the leading integration frameworks for LLMs are Langfuse, LangChain, LlamaIndex, and Griptape.

Orchestration refers to coordinating tasks and optimizing workflows to automate and streamline tasks such as data collection, model training, and evaluation. Orchestration is critical to make sure all the tasks are executed in the correct sequence, guiding LLM engineers on how to scale and manage resources and parallelize or distribute workloads to use infrastructure efficiently.

LLMOps, in this step, requires automated testing, and CI/CD to streamline workflows. It also involves checking that updates to model architectures, tool integrations, and code do not cause incompatibility in dependencies or breaking changes.

Step 7: Security and Reliability Engineering

LLMs are exposed to double risk: they are consumer-facing applications, and their architecture exposes them to vulnerabilities such as memorization (discussed later) and data leaks. Also, the large input scale makes it very hard to test against the attack surface with a high degree of certainty.

As LLMs interact with users, they may interact with confidential or sensitive user data, such as PII, user passwords (especially in code), and communication logs. Exposing such data to bad actors or the public internet poses significant privacy and security risks, so safeguarding it is critical to LLMOps life cycle management.

Third-party integrations can also add to security risks (e.g., malware in core packages). When LLMs are used as AI agents, users must provide them access to other applications, such as flight-booking websites or note tools (e.g., Notion). If attackers gain unauthorized access to an LLM, they can misuse or tamper with the model. This can cause significant harm to users and organizations, such as data and privacy breaches. To prevent eavesdropping and distributed denial-of-service (DDoS) attacks, implement secure authentication mechanisms and use secure communication channels that involve HTTPS protocols.

LLMs are also susceptible to leaking data or inadvertently memorizing sensitive information from training data. LLMs may memorize data during fine-tuning. It is good practice to perform regular security audits to minimize the risk of a privacy breach.

Malicious actors sometimes perform adversarial attacks on LLMs. They may manipulate prompts to deceive the model into producing incorrect output or conduct prompt and SQLi attacks into connected databases. To reduce these risks, you should validate input and conduct *red-teaming exercises*, which is when internal and external experts simulate attacks on LLMOps systems to identify vulnerabilities in the system.

Developing in-house pipelines and protocols for comprehensive testing, validation, monitoring, and audits can help you identify emerging threats and act quickly.

Step 8: Deployment and Monitoring

The last step in the LLM life cycle is deployment and monitoring. Effective deployment and monitoring are essential components of the LLMOps life cycle. As the LLM applications evolve, deploying them efficiently while ensuring robust monitoring mechanisms becomes paramount. This means selecting a deployment pattern (how the application is deployed and where it is run) and a deployment strategy (how changes are introduced into the production environment) tailored to specific use cases. And to develop metrics and benchmarks for monitoring the performance, reliability and security of your LLM-based application.

Deployment Patterns

There are many ways to deploy an LLM-based application. Common patterns include:

- Cloud-based
- On-premises
- Edge
- Containerized
- Serverless
- API

The deployment pattern and deployment strategy chosen will depend on factors such as the application’s use case, the volume of data, your organization’s real-time requirements and security considerations, and the available infrastructure. Some LLMs operate more efficiently with small datasets, whereas others (e.g., OpenAI) require massive corpora and thus need distributed systems (specifically horizontal scaling or vertical scaling) or specialized hardware for optimal performance.

Here are some questions to ask as you consider which deployment pattern to use:

- How many requests will users make to the LLM?
- How big is the LLM’s context window?
- Do you need to maintain strong access control?
- Do you want to keep the model’s architecture and parameters private?
- Do you have real-time data processing demands that require low-latency responses?
- Is the model’s workload predictable, or will it need to add computational resources on the fly?

Model Serving

A critical aspect of the deployment stage is *model serving*: the process of deploying LLMs into real-world applications. This is called *productionizing*.

Determining how difficult deploying your model will be depends on your architecture. If you’re deploying a pretrained model (such as GPT-4) as is, use cloud-based serverless deployment because it reduces the deployment challenges to monitoring performance. More complex architectures (such as deploying a microservices-based architecture that combines several pretrained and fine-tuned models together as a cohesive solution) come with significant challenges. Some of these challenges are:

Computational resources

LLMs are computationally demanding; serving them at scale requires high-performance hardware that can be hard to procure and complex to maintain in-house.

Latency

The sheer size of LLMs can increase loading time and inference latency in serving, which can make or break any real-time application.

Scalability

Scaling LLMs to handle a large number of concurrent requests can be a challenge for traditional serving architectures, resulting in service disruptions, slow response times, and increased infrastructure costs. This is even more significant for LLMs with very large context windows.

Versioning and rollback

The model learning process is so interconnected that it can be difficult to roll back changes or introduce model updates without disrupting service.

Monitoring and interpretability

Monitoring the performance of LLMs in real time is a very intricate problem. Also, dynamic context handling in real time can pose challenges for maintaining coherence between user sessions.

Security and compliance

LLMs can inadvertently generate sensitive or inappropriate content that can very quickly cause security breaches or compliance violations.

Tools like BentoML, KServe, Kubeflow, MLflow, and Seldon's Core provide solutions for efficient model deployment, scaling, versioning, and security. However, the tooling in this space is far from mature.

Deployment Strategy

Next, you'll need to decide on a deployment strategy. Options include:

Blue-green deployment

This common strategy involves maintaining two separate environments: blue (with the current production version) and green (with the new or updated version). The switch between these two environments is instantaneous—all user traffic is directed from blue to green. It provides a relatively straightforward rollback mechanism.

Rolling deployment

In a rolling deployment, updates or new versions of the LLM are gradually deployed across the infrastructure. This is done by taking a small subset of instances offline and updating them before bringing them back online. This process continues iteratively until all instances are updated. Rolling deployments help ensure continuous availability during the update process, minimizing downtime and providing a smooth transition to the next version.

Canary deployment

Here, the updated version is released to a small subset of users before being deployed to the entire user base. This allows the engineering teams to monitor its performance, identify potential issues, and gather feedback before making a full deployment. If the canary deployment meets the key KPIs, the updated version is gradually rolled out. This strategy is particularly effective for risk mitigation and is a great fit for heavily compliant industries such as finance and healthcare.

Shadow deployment

In shadow deployment, you deploy the updated version of the LLM in parallel with the existing production version, but it doesn’t handle live requests. Instead, it operates in the background, processing inputs and performing the same tasks as the productionized version. Its results are not used in the actual application—only for monitoring and model evaluation—until it’s ready for a canary or blue-green deployment. Shadow deployment is valuable for detecting data drift and for risk mitigation.

You’ll need to consider the app’s criticality, the organization’s risk tolerance, and whether you need continuous model availability during updates. Some applications can afford to experiment, but those handling sensitive information demand a more conservative approach, to mitigate risk and minimize potential disruption.

People often think of monitoring as a distinct final stage of the LLM life cycle, but in truth, monitoring functions as an integrated component throughout the life cycle. At every step, monitoring plays a critical role in assessing the model’s performance and alignment with organizational goals and predefined benchmarks.

Monitoring tools like WandB, Prometheus, Grafana, and Datadog provide monitoring, visualization, and analytics dashboards.

Irrespective of the tool, every organization needs its own monitoring framework that aligns with its key goals and LLM usage. Nvidia has an excellent internal monitoring framework that guides its LLMOps pipeline.

Conclusion

The development and adoption of LLMOps practices is still in its early stages. The skill sets required of LLM engineers are still slightly murky as we adopt and adapt the design patterns that underlie the Software 3.0 era. As LLMs' real-world applications expand, the complexities of LLMOps continue to evolve.

LLMOps is not an afterthought: it's an essential practice in developing and deploying large language models.

Chapter 1 gave you a glimpse of the vast landscape of LLMs and their massive potential. LLMs are no longer just hype. Chapter 2 addressed the need for an operational framework for building these systems and introduced LLMOps as a specialized yet essential practice for developing and deploying safer, more scalable, and more robust LLM applications. Chapter 3 walked you through the fundamentals of operationalizing the entire LLM life cycle to optimize performance and security.

I hope that this high-level report has given you a clear sense of how implementing a systematic framework can enable building more robust solutions and, in hindsight, also offer more transparency and collaboration across large engineering teams and various stakeholders.

Even as developers and enterprises integrate LLM applications into all sorts of product offerings across different domains and industries, we’ve only scratched the surface of their vast possibilities. Many aspects of LLMOps are still uncharted. There are tools yet to be developed, nuances yet to be discovered, more powerful models yet to be created. In the next five years, I expect every challenge discussed here to be amplified tenfold with increasing model sizes, integration of more modalities, and more specialized cybersecurity attacks.

This brief report merely lays the foundations for understanding LLMOps and provides you with resources to learn more.

Thank you for accompanying me on this journey. I look forward to exploring these uncharted territories of data, models, tools, security, and their operations in greater detail with you.

In my upcoming book on the same topic (*LLMOps*, forthcoming from O’Reilly), I’ll go deeper into the nuances of these unexplored technical challenges, examining the interplay between LLMs and their operational ecosystems and providing a systematic framework on how to harness the full potential of these models.

About the Author

Abi Aryan is an independent consultant and a machine learning engineer. She has over eight years of experience in the ML industry, building and deploying machine learning models in production for recommender systems, computer vision, and natural language processing—within a wide range of industries such as ecommerce, insurance, and media and entertainment.

Previously, Abi was a visiting research scholar at the Cognitive Sciences Lab at UCLA where she worked on developing intelligent agents. She has also authored research papers on AutoML, multiagent systems, and LLM cost modeling and evaluations, and is currently authoring *LLMOps: Managing Large Language Models in Production* for O’Reilly (forthcoming in 2025).