Building with LLMs

Thanks to the recent wave of large language models (LLMs), anyone can now build generative AI applications with impressive communication and reasoning capabilities. If an organization wants to stay competitive in this climate, it needs to start thinking about how to use LLMs to deliver the best possible user experience.

As a result, AI teams across industries are in a race to incorporate LLM capabilities into their products. This guide is designed to help developers and data scientists understand the workflow involved in bringing a state-of-the-art generative AI system to production. It introduces basic concepts from the world of LLMs in accessible language, and presents the tools to design, build, and maintain an end-to-end application that's driven by large language models.

deepset

Generative AI, retrieval augmentation, semantic search engines, and much, much more: **LLMs** are here to make all of our lives easier. But while the tools to set up an LLM-powered prototype for virtually any kind of use case are all out there, overseeing a project that implements a system end to end – from its inception to the final product that your users interact with – can seem like a daunting task for developers.

Large language models gained fame well beyond the usual NLP (natural language processing) bubble through the introduction of the ChatGPT user interface. While many other types of models are technically large language models, the term has become nearly synonymous with a particular type of generative language model such as GPT models, Llama 2, and similar. These are language models that leverage the Transformer neural network architecture, like all state-of-the-art models, and that are trained on huge amounts of textual data. Unlike smaller Transformer models such as BERT, LLMs are called large because they have billions or even trillions of parameters.

Our previous ebook ("NLP for Product Managers") provided an in-depth introduction to the topic of natural language processing in general, and to the task of managing a team looking to implement a system powered by language models in particular. We recommend checking it out if you want to get up to speed on concepts like language modeling or Transformer models. This book, on the other hand, looks at the process of building with LLMs from a practical angle – focusing on the process of designing, improving, and shipping the system itself.

If you're an ML engineer or a data scientist tasked with implementing an LLM-based system, you'll likely have lots of questions:

- Do I need to understand all the intricacies of large language models?
- How can I work around the limitations of the context window?
- How can I decide on the best model for my use case? What are the other things I need to pick and refine?
- How do I build a prototype that can be scaled to a production-ready system?
- How can I compare different pipeline designs and **prompts** to find those that work best for me?
- How can I collaborate with DevOps and back-end engineers on my team towards the final deployment of the LLM system?

If you have ever asked yourself any of these questions, then this book is for you.

Prompting

During training, LLMs have learned to complete a prompt in the best possible way. This is how they work during inference (querying) too: given a prompt, they try to generate the output that best completes it. That's why, to gain the best results, it is important that you prompt your model in the most helpful and systematic manner – so important, indeed, that it has given rise to a new discipline in machine learning, dubbed prompt engineering.

As the company behind Haystack and deepset Cloud, we at deepset are determined to enable the adoption of LLM-based products in all organizations that work with text data in some form. We see it as one of our core tasks to educate and empower the different people involved in the workflow of building with LLMs. By giving developers the tools to set up their own systems, we hope to see more organizations benefit from the incredible advances that the field of language modeling has undergone in the past couple of years.

In this book, we will go over the different processes that are essential to successfully setting up an LLM-powered system in production. After a brief introduction to the LLM ecosystem, we will look at the entire software development lifecycle from a high-level perspective, and finally dive into each aspect separately – from designing the pipeline and refining the prompt to deploying the LLM-powered pipeline to production.

Contents

deepset

01 THE ECOSYSTEM OF AN LLM APPLICATION

People just learning about LLMs are sometimes intimidated by the complexity of these large, Transformer-based language models. But when we talk about building with LLMs, we can often ignore the complex math and advanced architectural decisions that go into training these models. That's because for most use cases it is not needed to train an LLM from scratch.

Rather, building with LLMs requires choosing from a vast and ever-growing selection of pretrained models, some of which are proprietary, while others are open source. Thanks to model sharing, you can go to centralized locations like the Hugging Face model hub, where tens of thousands of pre-trained models are freely available. You can also use an interface like OpenAI's API to access their language models without even leaving your IDE. Due to this ease of sharing, everyone can benefit from the huge leaps that LLM research and development have made since the inception of the Transformer architecture for language modeling.

But an LLM alone does not make a production-ready application. Depending on the nature of your application, you'll also need data stores, smaller embedding models, prompts, and the hardware to run your system in production.

That's why frameworks for building with LLMs assist their users with everything needed to set up and deploy a system that embeds LLMs, such as:

- Developing demo-ready prototypes without model lock-in, making it easy to update your system with the newest and most suitable LLMs if you see a decay in performance of your system in production.
- Connecting to different (vector) databases, offering embedding models for **RAG** applications, and making it easy to compare different models and prompts.
- Flexibly letting you adjust your pipeline to contain additional components, such as a prompt injection classifier or a model to detect hallucinations.
- Providing the building blocks to set up modular, customized LLM pipeline infrastructure.
- Offering the tooling required for evaluating and improving your prototypes and deploying them to production.

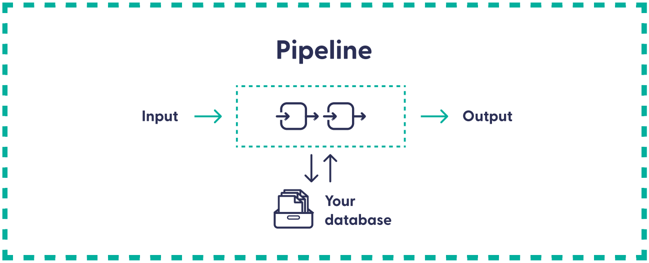
Retrieval augmented generation (RAG)

LLMs learn a faithful representation of their training data. However, that data has a cutoff date and overall doesn't contain all the specific knowledge needed in many enterprise applications. Luckily, we can provide additional context to the LLM in our prompt in the shape of our own company-owned data, and instruct the model to base its responses on that. In retrieval augmented generation (RAG), we delegate the context-finding step to a retrieval module. Its task is to identify the documents that are most likely to contain the relevant information and include those in the prompt together with the user's query.

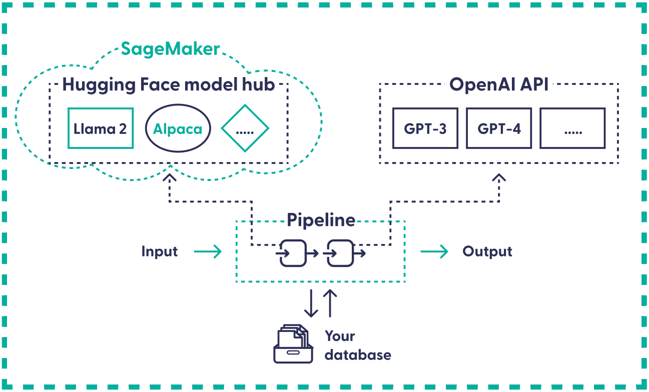
In the context of an LLM application, pipelines are composite units that are used for indexing (writing documents and their vectors into databases) and for inference (querying).

Indexing pipelines are particularly relevant in a RAG context (retrieval augmented generation) or for other applications, such as semantic search or file similarity – basically, whenever you want to work with your own data, you need to index that data into a database to make it available for your LLM-powered application. The indexing pipeline helps you achieve this by preprocessing and cleaning the raw files, before turning them into documents, then into indexable vectors.

Query pipelines handle the logic of inputting a natural-language query, transforming it in one or multiple steps, and outputting the result. The great benefit of pipelines is that they allow developers to compare different setups, build and test prototypes quickly, and later hand those prototypes over to a back-end engineer for final implementation.



In addition to different readymade pipeline designs for indexing and querying (as well as the option to adapt those, or design your own from scratch), LLM frameworks also offer interfaces to the various locations where pre-trained models are hosted, such as the APIs of different proprietary LLM providers or the Hugging Face model hub, where open source models are shared. This means you can simply plug different models into your query pipeline and look at the results to find out what works best for you.



Besides open source libraries and frameworks, there are also managed solutions for implementing projects with LLMs. Such solutions provide a more approachable and user-friendly way to set up a working system in production. The user interface makes it easy to implement and compare different system configurations, to collect feedback from end users early in the process, and to serve and deploy the final LLM application.

Let’s now have a look at the implementation cycle of an LLM-powered system as a whole.

02 THE IMPLEMENTATION CYCLE FOR LLM-POWERED SYSTEMS

The NLP implementation process consists of two parts. In the prototyping phase, the developer sets up a working prototype pipeline and experiments with different configurations. When building with LLMs, we usually opt for *rapid prototyping*: a workflow in which an early system gets developed and deployed quickly, to make sure that we can test it with real users and collect their feedback very early on in the process. This allows us to iterate through prototypes quickly, constantly improving and refining our system – and prevents us from spending months on building a complex system that turns out not to solve our problem.

Once the cycle of prototyping, deployment, and user feedback has produced a satisfactory prototype system, the AI engineer or data scientist who developed the system hands it over to an MLOps (machine learning operations) engineer. MLOps engineers concern themselves with how to deploy a system to production in the most robust, reproducible, and cost-saving way. Plus, once the LLM-powered application is running in production, MLOps makes sure that it is monitored continuously, taking care of fixing and updating the system when its performance plummets. For instance, in the fast-paced world of LLMs, a more powerful new model might quickly arrive on the scene, making it necessary to swap an existing model for it.

Both the prototyping and the MLOps phase therefore heavily feature the testing and evaluation of pipelines, both quantitatively and qualitatively. Only by periodically testing your system on real-world data and **with real users** can you make sure that it remains up to date.

User feedback

Rather than following the sequential data science model of finalizing a system before deploying and sharing it with end users, modern machine learning projects make **user feedback** an essential element of the development lifecycle. In the old model, it could very well happen that after months of development, you would realize that your final product wasn't solving your users' actual problem. By involving an example group of end users early on, you can analyze their feedback to improve your system – for instance, by refining your prompt, changing your LLM, or improving your model for retrieval.

Let's take a bird's-eye view of the different phases of the implementation process, before we move on to look at each of the following topics in more detail.

Prototyping

- **Design your pipeline.** Here, you think about what you want your system to achieve. Does it need to generate answers based on the documents in your database? Is it crucial that it returns the most relevant document as its first result? Consider potential future use cases as well, so that the pipeline will be flexible and capable of scaling up with your project.
- **Choose the models and your prompt.** The design decisions made in the previous step determine which kinds of LLMs are suitable for your system. You can pick different proprietary and open source models and see how they compare to each other. To come up with an initial prompt, be sure to follow best practices for prompting. You're likely to refine your prompt as you proceed through your project.

- **Build the prototype.** Instantiate the pipeline with your preliminary selection of models and connect it to your database. Now you already have a running system! If you're looking to experiment with different language models, then you'll likely build several pipeline prototypes at this stage.
- **Experiment, evaluate, and test.** This phase serves to find the best setup for your use case and the resources you have available. Is the inference fast enough? Accurate enough? What do your users think about it? You need to deploy a preliminary system, get it into the hands of your end users, and collect their feedback. This is particularly important considering the generative nature of LLMs because **it's hard to evaluate them** in a quantifiable way. If you're leveraging document retrieval in your pipeline, you'll want to evaluate it using a representative evaluation dataset. The more energy and resources you pour into experimentation, the better your system will become.

Evaluating LLMs

Some language models are harder to evaluate than others. For example, if a sentiment classifier labels a text as *positive* while the correct label is *negative*, then we can safely say that it made a mistake. However, when it comes to generating or extracting text, it isn't always so easy to say what's wrong and what isn't. Your generative LLM may output a summary that's vastly different from the "correct" text – but it can be just as good, or even better. That's why it's important to evaluate LLMs not only quantitatively, but also qualitatively – by testing them with real-world users.

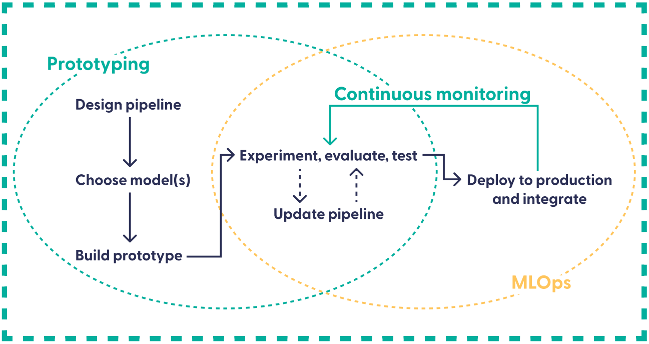
- **Fine-tune your prompt and models.** If the LLM isn't generating the kind of answers that you're looking for, you could evaluate and fine-tune your retrieval model (if you're using one) and tweak the prompt using different methods – more on that in chapter 5. You could also fine-tune your generative or extractive language models.

- **Repeat.** Having adjusted your prompt, your retrieval setup, and potentially even your language models, you can follow this step with another iteration of experimentation, evaluation, and testing.

MLOps

- **Deployment to production.** At this stage, you hand your prototype pipeline over to the DevOps team for deployment to production. This requires setting up the indexing and query pipelines and setting up the workflow for data integration. How does data – both the documents and user queries – flow into the system? How is it processed? And how is existing data updated? These processes need to be properly defined before the pipelines can be deployed.

- **Monitoring your system.** Machine learning systems in production are a bit like living things that benefit from regular health checks. So you need to make sure to monitor the deployed product, periodically checking the system's performance against updated evaluation data sets and continuing to collect feedback from real users. It's important to make sure that your system remains up to date and doesn't decay. If you notice a dip in your system's quality, you'll likely want to go back to refining your prompt, checking if retrieval works as intended, and whether it's time to stick a new LLM into your pipeline. So, in a way, this final phase consists of periodically repeating the steps outlined earlier – only now, you're performing them on a system that's already been deployed to production.



The above diagram illustrates why, contrary to most people's perceptions, building an LLM-powered system is about much more than simply using an API and building a front end. Rather, the hard work in making these systems production-ready is about ensuring that you have all the parts you need: the right models and a way to swap them in and out, a framework to compose pipelines, and a way to observe and evaluate your system in practice.

While the process in its entirety may seem a bit overwhelming, it helps to break it down into neatly defined steps. So in the next chapters, we'll have a look at each of the various stages of the implementation process in detail. But first, let's talk about language models.

03 A LANGUAGE MODEL OVERVIEW

It can be easy to lose track of what's happening in the field of language modeling. Even the names we use to refer to it change rapidly: AI, NLP, or LLMs. To make sure you don't get lost, we'll provide a quick overview of the main branches in language modeling today. Large language models may have stolen the limelight for now, but there are at least two more categories of LMs worth keeping on your radar.

LLMs

Large (generative) language models are much bigger than the previous generation of language models, hence their name (their size describes the number of trainable parameters). In fact, they are so big that it's mostly not feasible to run them locally, making it necessary to run them on an external cloud provider. This applies even when dealing with an open source model; but other models, like GPT, are proprietary anyway, meaning that you can only communicate with them via a fee-based API. It's important to keep in mind that when we talk about LLMs, we almost always mean generative models – that is, models that generate well-formed, conversational content in response to a prompt. This includes writing summaries, translations, and many other output formats.

Encoder models

Other, smaller language models typically perform more specific tasks. Described as Transformer models, language models, and sometimes even LLMs, a model like BERT for instance doesn't *generate* content. Rather, it acts as a language-aware classifier that can be fine-tuned in different ways.

Programmatically, what distinguishes this class of models from the generative group is that they only have an encoder part, not a decoder.

A popular category of BERT-based models can do extractive question answering, which means identifying an answer passage in a document in response to a query. These models are always free and open source, and can be downloaded and fine-tuned locally, on your own machine.

Embedding models

Finally, there's a family of modest language models that often flies under the radar, even though its members provide value daily for billions of people: namely, embedding models. These Transformers embed text passages as dense vectors. This lets them encode the semantic information of those texts – that is, their meaning rather than their lexical form. The vectors are stored in a database along with the original document, for later retrieval in an application that leverages semantic vector search – for instance, in a RAG scenario.

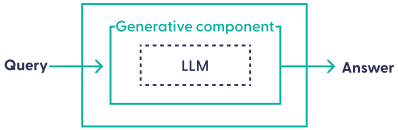
04 DESIGNING THE PIPELINE

As mentioned earlier, pipelines are a foundational paradigm for building effective applications with LLMs. They allow engineers to architect the correct flow of data from your database, all the way up to an LLM, so as to achieve the desired outcome. How, exactly, do pipelines accomplish that?

The pipeline paradigm

As we have established in chapter 1, pipelines are system architectures that combine various components – which are powered by language models – in a sequential order. For an example of a minimal pipeline design, consider a simple pipeline for answer generation. It consists of an input interface for user queries, a component that provides an interface to an LLM and prompts the model with the user's query, and an output interface that returns the model's answers to the user's query. In this setup, the LLM does not have access to any external data, only the information it has memorized during training.

Simple generative pipeline



While useful for illustration purposes, such an overly simplified setup is not very useful in practice.

What are the advantages of using pipelines?

Most LLM-powered systems in reality are much more complex than our example above. That's because LLMs are great at generating answers, but prone to **hallucinations** – plus, they don't know anything about your confidential data or events that happened past the cutoff date of their **training data**. In fact, an LLM isn't particularly useful on its own for most enterprise applications. Rather, it needs to be embedded in a complex system that can leverage additional data – and that is where pipelines really shine.

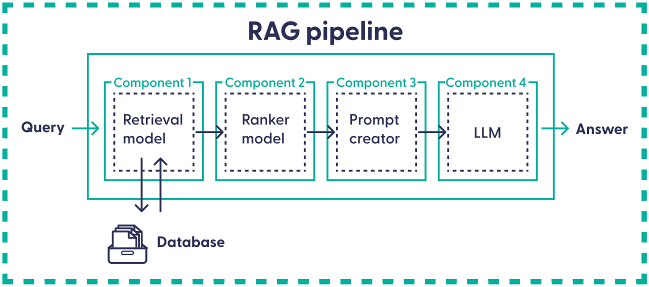
Hallucinations are "made-up" answers by an LLM that are not based on facts and often contain inaccuracies and factual errors. They happen because the LLM has been tuned to output a response and it tries to comply even when it doesn't "know" an answer. The best techniques to combat hallucinations are RAG, clever prompting, and using another model to check whether the LLM's answers are grounded in the documents it has seen.

Data in NLP

As a sub-discipline of machine learning, and, more specifically, deep learning, modern NLP produces systems whose quality largely depends on the training data. That's why data-centric practices like creating high-quality data sets, regularly monitoring the data even after deployment, and performing qualitative error analyses are vital tools for the success of a project that leverages language models. The fact that many for-profit LLM providers use publicly available data from the internet to train their models has also raised ethical questions. The unclear provenance of that data means that it can contain biases and falsehoods. There are also unresolved legal questions with regards to data that is publicly available but whose license may not permit that data to be used for training language models.

Pipelines allow you to build an architecture around the LLM that supplies it with the relevant data before generating a response.

The main advantage of the pipeline structure is that this architecture is customizable to virtually any requirement. So, in our next example, we're going to look at a more involved pipeline design for *retrieval augmented generation* (RAG). It leverages document search to retrieve the right documents from a corpus, which it then passes on to the LLM as a basis for its answer.



1. They're easy to build

Our RAG system combines four components: one for retrieval, one for document ranking, one for prompt creation, and one for the LLM itself. The pipeline takes care of routing the query to the first component, whose output then serves as input to the next component, and so forth. It also allows you to set all of the components' parameters – such as how many documents to return, the prompt template, and the LLM to use – when you first define the pipeline. By bundling all of these parameters in one place, the pipeline becomes much more manageable than if you had to juggle each step individually.

2. They're easy to reason about

That brings us to the second huge advantage: the pipeline paradigm allows you to reason about your system as a self-contained unit, just like the way a complex section of code can be abstracted away into a function or class and treated as a single item.

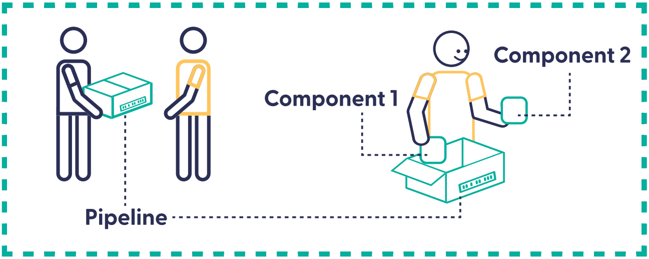
When managing your RAG pipeline as part of your final product, for example, you don't always need to remember that it uses a retriever, a ranker, and a custom prompt generator, or which of those components connects to the database. All you need to know is that the pipeline receives a query, and, in response, generates a response based on your data.

3. They're easy to share and document

Like a function or class, a pipeline is also easier to share with other people in your organization. Most LLM frameworks have a way of exporting pipelines in some basic file format. If you want to move your system to production, for example, all you need to do is export it to a file and hand it over to the respective engineers. This allows you to reproduce, reuse, and version-control any pipeline setup.

Black box versus individual components

But while it can be extremely useful to think about pipelines as self-contained units, it is often necessary to open the black box and look into the system components separately: during debugging, model evaluation, or testing. Pipelines make it possible to isolate each component and inspect its outputs, so that you remain in full control of your system's architecture.



Choosing the pipeline design

Your pipeline design should serve your use case. While many frameworks offer ready-made pipeline architectures, it's usually just as simple to set up your own system by connecting components within a default pipeline object. For example, you could use a custom design with two branches that lead to different retrieval methods such as keyword retrieval or embedding retrieval, or you could even build out your own custom component that performs translation at the right step of the pipeline.

To better understand the different ways in which building with LLMs can benefit organizations in practice, let's have a look at the following three model use cases. Although, strictly speaking, these are hypothetical examples, they are inspired by real-world use cases that we've encountered over the years.

Use case 1: Combining the power of generative AI and your own data to create a virtual domain expert

A legal publishing house offers a custom, RAG-powered chat interface for its clients. Lawyers and other legal professionals can speed up their research process many times over through the offering, which generates its responses on the basis of millions of curated, regularly updated documents from the law domain.

Use case 2: Building a QA system on top of a database of technical manuals

An aerospace manufacturer has amassed thousands of manuals for their products over the years. With that much textual information in private silos, it's hard for pilots in training to find the answers to their problems quickly (they can't just semantically scan the data set using internet search engines). Therefore, the company builds an **extractive question answering (QA)** interface.

Their setup uses a semantic retrieval model from the Hugging Face model hub, plus a QA model adapted to the technical domain.

Extractive question answering uses encoder models that can extract answers from documents in response to a query. These models do not synthesize answers from scratch like generative LLMs do – they only mark the passage in the document that contains the answer. Like their LLM counterparts, these smaller Transformers are often combined with a retrieval module.

Use case 3: Extracting information from business reports

A federal institution assesses financial risk factors for companies based on their business reports. They run each report through a pipeline for information extraction, which highlights the relevant sections using a model that's been fine-tuned to financial jargon. While low-confidence results still need to be checked manually, the system allows their officers to assess individual companies much faster.

Once you have decided on the structure of your pipeline design, you can start thinking about what language models and prompts to populate individual components with.

05 CHOOSING THE RIGHT MODEL AND PROMPT

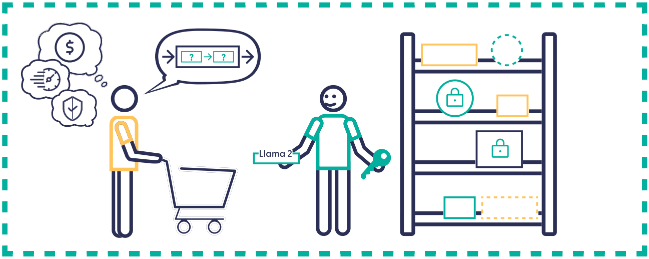
It is very rare that those building and deploying modern, LLM-based systems will have to train or **fine-tune** their own models (although this may be needed in some very specific use cases). Instead, working with LLMs is often all about:

- Choosing the right models for your use case
- Prompting the LLM to complete the task of your application
- Evaluating the model
- Monitoring and updating the retrieval model

Fine-tuning
A pre-trained model is characterized by its weights – the parameters that have been set during training, and that determine the model's behavior during inference. When you fine-tune a model, the weights are initialized to the pre-trained values, and you then adapt them further to your data by running additional training steps. Fine-tuning is a well-established practice for smaller encoder models, as it can teach them how to perform specific tasks or understand domain-specific language better. It is less common with LLMs – these models are much more powerful, and rather than fine-tuning them, we engage in "steering" techniques like prompting to harness their power. For very specialized use cases, you may still want to fine-tune an LLM, however – and some providers allow their users to do so via an API.

In this chapter, we'll look at the first step of choosing the model. Because of the huge – and steadily growing – selection of pre-trained LLMs out there, it helps to be clear in advance about what you require from the model.

These include considerations about your ideal LLM's latency (speed), quality of responses, cost, and privacy. Having clarity about these factors will help you narrow down the space of candidates when you go model-shopping.



What do you need from your language model?

Of course, your project will dictate the main properties of your language model, such as the language itself and the specific task. LLMs work well out of the box for many of the world's major languages like English, Spanish, and Chinese. Similarly, many text embedding models for document retrieval are able to process documents in multiple languages.

Leaderboards like the Chatbot Arena or Hugging Face's Open LLM Leaderboard can help you get an idea of the quality of different models' output and help you preselect a few candidates. For a comparison of different embedders, have a look at the MTEB Leaderboard on Hugging Face. Because these models' results can vary widely between use cases, it's best to try out the models for yourself, and on your own data, by plugging them into your pipeline.

In terms of cost, consider that even open source models will likely have to run on external servers – and that cost often adds up to more than the fees charged by third-party model providers.

This brings us to the issue of security and legal concerns. Many of the most high-performing LLMs today are proprietary models served through third-party APIs such as OpenAI or Cohere. Unlike open source models, these LLMs cannot be hosted on your own infrastructure, which makes them unsuitable for teams with stringent security requirements and legal constraints.

Teams with security constraints that don't allow them to make use of proprietary models often choose to host models themselves. This comes at an effort cost: deployment, maintaining servers, and ensuring availability. Alternatively, **services like SageMaker and Azure** allow you to use their hardware to deploy and host your own model.

Note that there's no reason to limit yourself to one model. Thanks to the option of fast prototyping offered by pipelines, you'll be able to try and test many different models before settling on the model to use in production.

Hosted inference
LLMs are too large to run locally on your machine – you would need several GPUs for that. Closed-source models are served via third-party APIs, requiring you to send your prompt to the provider and receive the model's output in return. Open source models, on the other hand, can be deployed on a hosted inference service like SageMaker or Azure. This transfers the heavy lifting of running an LLM to an external service, while leaving the control over which models are running and how they are accessed in your hands.

Model prompting

The triumph of LLMs has given rise to a new discipline in AI: prompt engineering. Skillful prompting has become an increasingly important design step in LLM applications because it has a stark effect on their performance. In some cases, the choice of prompt can even influence which LLM you end up using in your application. But what exactly is a prompt?

Simply put, a prompt is the instruction we send to an LLM, in natural language. Most – though not all – LLMs are designed to follow instructions.

Besides containing a placeholder for the actual user query, prompts allow us to outline the desired shape, tone, length, etc. of the LLM's response. They are so flexible that, depending on the model's abilities, you can use prompting to morph your LLM pipeline into anything from a summarization system to a question answering one.

Model prompting is one of the key steps in a retrieval augmented generation (RAG) pipeline. Let's have a look at two example prompts for RAG systems. The first one is suitable for a RAG pipeline that summarizes the context provided by the retrieved documents:

```
"Create a summary of the following context.
Context: {documents}
Summary:"
```

The following prompt, on the other hand, instructs the LLM to answer a query based on the provided context:

```
"Answer the question based on the provided context.
Context: {documents}
Question: {query}
Answer:"
```

As these examples illustrate, the prompt creation component can act as a template with placeholders that get filled in by other components in the pipeline. For example, `{documents}` could be provided by a retriever component, while `{query}` could be user input.

Each LLM has a context window, which is the number of tokens it can attend to. The length of the context window depends on the model and determines how many documents the LLM can process at once. Some models are designed to accept very long prompts, while others can only parse a more limited context. The shorter the context window, the more important it is for retrieval to select the most relevant documents.

06 PROTOTYPING

After designing your pipeline architecture and choosing your model candidates, it’s time to start prototyping. Fast prototyping, where you iterate through different system configurations by evaluating, testing, and adapting pipelines quickly, is an essential element of the LLM-powered implementation cycle.

A key advantage of using pipelines over setting up a system from scratch is that they allow for fast prototyping. All you need to do is instantiate the pipeline, add the models to it, and connect the pipeline to your underlying database. With the right LLM framework, the entire process won’t require more than a few lines of code.

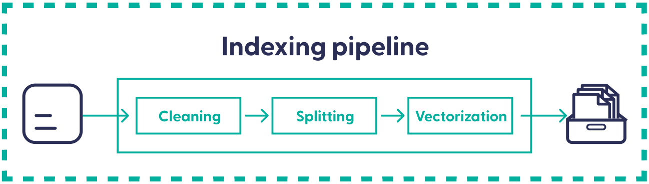
Working with documents

Text documents can come in many shapes and formats, and their initial form may not fit the language model you’re going to work with. For instance, how many tokens a prompt can pass on from the retrieval component to the generative component depends largely on the LLM you use, while language models for extractive QA benefit always from shorter documents.

Furthermore, if your pipeline uses a component for document search with text embeddings – as some **retrieval methods** do – you need to create those embeddings first, before you can add them to the database – a task known as indexing. The choice of embedding model will also have an impact on the length your documents need to be.

Keyword vs. embedding retrieval
Document retrieval describes the task of selecting documents from a large corpus in response to a query. You can use a keyword-based method like BM25, which is language- and domain-agnostic and runs very fast. However, if you want your retrieval module to be aware of semantics, you'll need to use an embedding model that leverages Transformers. Such a dense model may perform poorly in an out-of-domain setting, however.

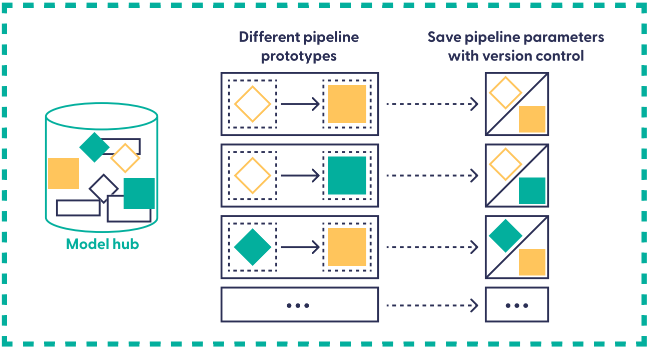
Once more, you can use a pipeline to streamline the task of preprocessing and indexing your documents.



The indexing pipeline pre-processes your documents – by cleaning and splitting them. It then transforms them into vectors and indexes them.

Setting up the query pipeline

Set up the query pipeline by instantiating it together with the models that you want to use. If you're working with several prototypes, make sure to keep them separate and identifiable. Ideally, do this using a version control system such as Git; alternatively, give them descriptive names and save their parameters in separate files.


Different pipeline prototypes
Save pipeline parameters with version control
Model hub
•••
•••

The pipeline is then connected to the indexed database. Now you can start querying your system – and, even better, let other people do the same.

07 TESTING AND EVALUATION

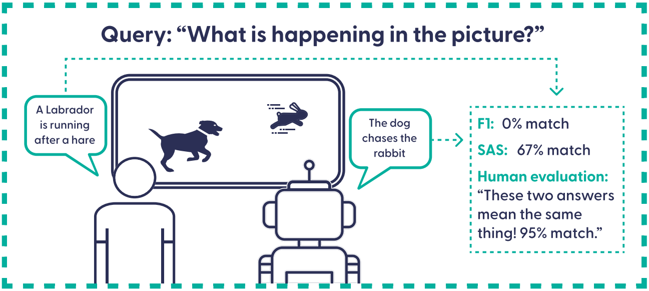
To understand the quality of your pipelines – and to be able to compare different systems – you'll need to evaluate and test them. For a quantitative analysis, a representative evaluation data set is run through the model to compute a number of performance metrics (see below).

LLM-powered systems aren't as straightforward to evaluate as other machine learning models, because it's not always easy to define when language is "correct" (see infobox "Evaluating LLMs"). That's why, to successfully evaluate an LLM system, it's necessary to include qualitative judgments in addition to the quantitative measurements.

Evaluating an LLM pipeline

When you evaluate an LLM pipeline, you can check the prediction quality either of the entire system or of individual components. The metrics you use then depend on a component's task. For example, to evaluate a retrieval component, you'll be using a metric like recall, which measures the percentage of correct documents retrieved.

For the evaluation of extractive models, you'll be looking at metrics like the F1 score, which measures the lexical overlap between the expected answer and the system's answer. However, the shortcomings of such metrics are apparent when we talk about generative, semantics-driven AI systems, whose entire purpose is to abstract away from individual words and focus on the meaning of a text.


Query: "What is happening in the picture?"
A Labrador is running after a hare
The dog chases the rabbit
F1: 0% match
SAS: 67% match
Human evaluation: "These two answers mean the same thing! 95% match."

Two people could come up with vastly different responses to the same query, and both could be equally appropriate. To capture that property, some frameworks have proposed Transformer-based metrics like the semantic answer similarity (SAS), which measure the semantic rather than the lexical congruence of two texts. Some other approaches have included using natural language inference (NLI) models that check whether a statement is entailed within the knowledge base, allowing us to detect hallucinations. However, as such methods are only slowly gaining wider traction, complementing your quantitative analysis with a qualitative one remains absolutely necessary.

Qualitative evaluation

Even if the data you use for your evaluation data set is relatively recent, nothing beats evaluating your system using real user feedback. Thanks to the easy prototyping offered by LLM pipelines, you'll be able to demonstrate one or more system prototypes to your audience early in the implementation cycle.

Share a simple browser-based prototype with a representative group of users and have them submit queries as they would to your final system. You can then ask them to evaluate the answers they received from the system.

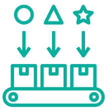
Aggregated, these judgments are valuable numbers that will help you decide which system to use in production, or which aspects of your system need improvement.

Testing your system with real users has a second benefit: it will alert you to discrepancies between your own preconceptions of how your system will be used and how it's actually used in the real world. Perhaps the demographic of your users has shifted, and they talk about different topics than they did when you collected your data. Perhaps they use words or abbreviations that your system has never seen before. You'll only know by investigating the data produced by your actual users.

You might also discover that the data you provide to your system for retrieval is outdated or lacking in some other ways. In that case, you could look into techniques for data collection and curation. A simple but effective technique for improving retrieval is to combine a keyword (sparse) retriever with an embedding (dense) one. Such a hybrid retrieval setup benefits from adding a **ranker** to the pipeline.

Rankers

In most applications that use retrieval, the order of the retrieved documents is crucial – for instance, when you only pass on a subset of those documents to the next component, or when you need to combine the results from two retrievers (hybrid retrieval). Rankers use small, fast Transformer models that rank the retrieved documents according to different criteria, like their relevance to the query or their diversity.



08 DEPLOYMENT AND MONITORING

Once you've updated your datasets for retrieval, refined your prompts, and evaluated your prototypes in terms of both user feedback and evaluation metrics, it's finally time to deploy the system to production and release it to the world.

But, a bit like a garden that requires constant tending to, your ML models in production still need regular check-ins and care from their developer. So, rather than handing off your system to a back-end engineer and being done with it, MLOps requires that you as the model maintainer stay in the loop. Not only do you need to take care to deliver a deployable and maintainable system, but because of the unique perishable nature of ML models, you also need to ensure that the downstream systems that use them remain up to date.

So, in this chapter, let's look at the steps involved in bringing your system to production – and maintaining it. Then we'll talk about how managed tools can assist you in streamlining a process that may, at times, seem quite overwhelming.

From prototypes to a system in production

While the underlying models are the same, the difference between running a prototype pipeline locally and deploying that same pipeline to production is like day and night. That's because the requirements are quite different. A pipeline in production needs to run reliably at all times, be able to accept and process user queries quickly, and communicate with many different clients.

In the bigger picture, the LLM-powered pipeline is just one component of the larger business application it is embedded in. Such an application is based on a complex infrastructure, comprising servers, process management tools, and the database.

Such a setup has to handle many requirements. For example, it needs to:

- Handle the scaling up and down of compute resources depending on system load.
- Provide copies of your data in the case of hardware outages or network problems.
- Duplicate pipelines in the case of many parallel requests.
- Handle user authentication.
- Limit the rate of queries to prevent abuse of the system.
- Protect sensitive user data.

Pipelines bring the principles of modularity and composability to LLMs, in the same way composable stacks did for other web apps a few years ago. Building your system out of modular components in a pipeline offers easily examined, granular control for use in installation, debugging, and analysis.

By packaging your composed system in a pipeline, you have created a portable piece of software that handles everything from raw data to inference, allowing a smooth transition to a production environment.

Integrate your LLM-powered system into your final product

In order for your pipeline to seamlessly integrate into the business application, it has to fit the overall API-driven application architecture.

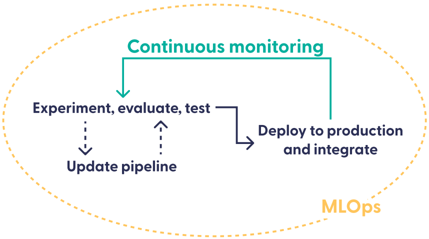
In the last decade using an HTTP RESTful API has become an industry standard for inter- and intra-application integration. This approach is widely used to instrument the common entry points to the application, where on a technical level the client requests and the application responses flow back and forth in a predictable manner.

If your pipeline isn't wrapped in an API layer, it's going to be extremely hard for your peers in back-end and front-end app development to integrate it into the actual business application. In many cases, the lack of a standardized, properly implemented API layer becomes a huge obstacle to actually using a pipeline in production.

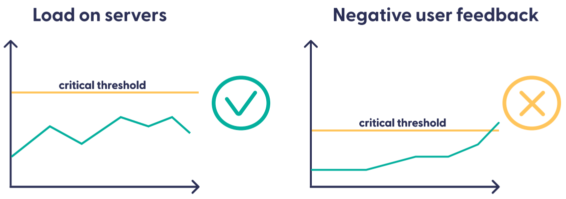
Monitor your LLM application

A central goal of MLOps is to account for the decay likely to occur in machine learning models. “Model decay” describes the phenomenon where ML models in production become outdated – and therefore less useful – over time. That’s why it’s so crucial to monitor your deployed system, and update it once it shows signs of decay.

As outlined in chapter 2, this part of the production process for LLMs resembles a cycle: on a high level, you go back to evaluating and testing your models, and, optionally, to refining your pipeline. This could be done by, for example, adapting the prompt, swapping the LLM for a better one, or improving your retrieval setup. You then have to redeploy the updated pipeline to production.


Continuous monitoring
Experiment, evaluate, test
Update pipeline
Deploy to production and integrate
MLOps

While some steps of the monitoring process – such as collecting user feedback and refining the prompt – involve manual work, others can and should be automated. To be able to monitor your system, it's useful to devise metrics that can measure relevant aspects of your system's performance and quality. Those measurements can be calculated periodically, and their results can be integrated into some visualization, like a dashboard. In addition, you might want to define thresholds which, should your system reach them, will alert you to any critical states.



How about those aspects of the MLOps cycle that can't be fully automated? For example, both the selection of representative users and the evaluation of their feedback require some manual work, as does the collection of new training data and the retraining of models.

Both tasks – feedback and retraining – are made considerably easier by managed LLM platforms, which are designed to assist AI teams in organizing their workflow, demoing prototypes, visualizing workloads, and performing version control.

In addition to providing you with the tools to manage, monitor, and maintain your LLM system in production, managed tools also take a lot of work off the back-end engineer's shoulders – for example, by handling the deployment, automated scaling, and authentication of your system in production. And, of course, most managed solutions for serving machine learning systems in production include tools for monitoring those systems in a fully automated manner.

Building, implementing, and maintaining an application that's powered by the latest LLMs might not be the easiest task for a developer – but thanks to model sharing, LLM frameworks, and hosted solutions with powerful back ends and intuitive user interfaces, it can turn into a highly rewarding job.

It's certainly worth it: businesses across industries are benefiting from the incredible abilities of large language models and their sheer endless customizability. Paired with safeguarding and quality-improving techniques like prompting and retrieval augmentation, these systems are poised to change the landscape of data management as we know it. Providing a sleek, natural-language interface powered by generative AI to unlock the information hidden in your data storage isn't only a surefire way of building a satisfied user base – it's a must if you want to stay competitive.

Get started

At deepset, we think a lot about how to make the most exciting technological advances of our time available to a broader range of people. We're certain: it's time to bring the impressive results of the last couple of years in natural language processing – which culminated in the development of LLMs – to every application and every service that uses natural language.

Haystack, our open source LLM framework, lets you build your own natural language interfaces for your data.

Check out our LLM platform deepset Cloud for a fully managed solution that assists AI teams in building their own applications powered by large language models. deepset Cloud offers all the tools needed for fast prototyping, deployment, and collecting user feedback – all while implementing best practices from the MLOps cycle.

Finally, we'd be happy to welcome you in our online community on Discord, where you can chat to other developers that are building LLM into their products – or directly to our team.

09 LEARN MORE

deepset website

deepset Cloud documentation

Haystack website

Hallucination detection

Retrieval augmented generation

What is an LLM?

Hugging Face model hub

Beginner's guide to prompting

Haystack on Discord