


Guide

# The Practical Guide to Deploying Large Language Models

The background features large, overlapping abstract shapes in shades of cyan and blue, creating a modern, geometric aesthetic.



## Table of Contents

- 01 Introduction: LLM Applications and Components
- 02 The Challenges and Compromises in Production Deployment
- 03 Best Practices for Orchestrating LLMs in Production
- 04 Conclusion: Empowering your Business with LLMs
- 05 About Seldon

## Introduction: Unleashing the Power of LLMs in Production Environments

LLMs have emerged as a remarkable force and are reshaping how businesses communicate, analyze data, and innovate.

This guide offers a comprehensive exploration of the components of LLM applications, deployment tradeoffs, and pragmatic considerations that are essential for successfully orchestrating LLMs in production. The aim of this guide is to equip you with the knowledge you need to perform scalable, efficient, guided inference with considerations around monitoring and debugging.

Large Language Models represent an important evolution in enterprise AI capabilities. Having been trained on massive datasets, these models gain broad knowledge that serves as a strong foundation for various applications. Their versatility enables them to be easily adaptable to new use cases beyond their original training objective.

There's a variety of ways to apply LLMs effectively:

- Leverage the models' existing knowledge by asking them for predictions or inferences.
- Fine-tune the models by providing additional domain-specific training data
- Use the models for transfer learning by taking learnings and applying them to new tasks

With proper implementation, LLMs can enable robust and scalable AI systems that drive true business value. Curious about how LLMs can enhance your organization's capabilities and effectiveness?

Here are some examples of LLM applications:



### Customer Service Chatbot

Provide answers, product recommendations, and conversational search that answers your customer's questions.



### Document Understanding

Read documents, get summaries of key details, and discover user sentiment in those documents.



### Code Completion

Auto-generate code, explain and document code, and get suggestions on how to improve it.



### Content Generation

Make more content including blog posts, emails, ad copy, and outlines based prompts, faster.



### AI Assisted Search

Change the way you research by getting answers to questions using a large collection of texts.



### Translation

Change text from one language to another and adjust the writing style to change the tone or reading level.



LLM models are growing stronger every day, and regularly gaining new capabilities. LLMs allow for quick prototyping, but they can get quite complex as more components are added to improve the functionality and accuracy of the outputs.

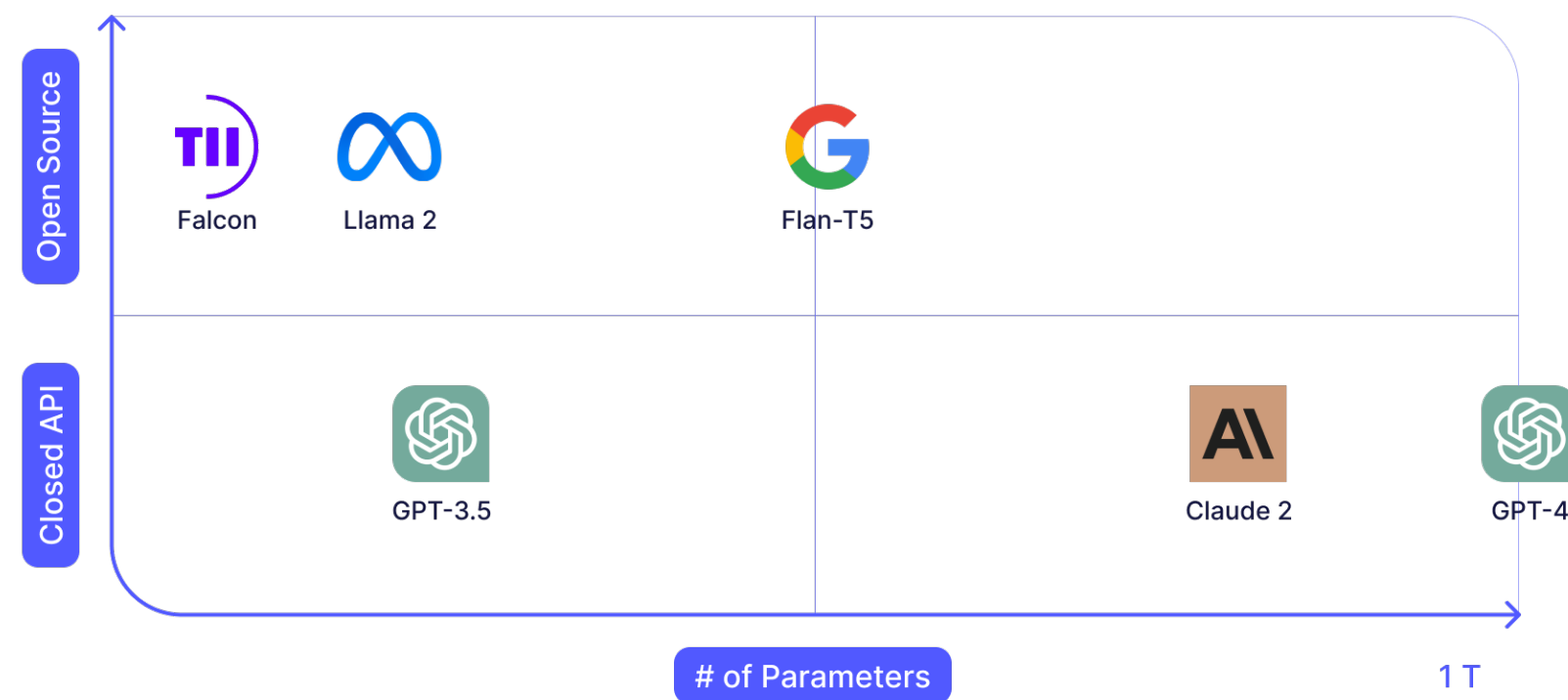
### LLM Components and Reference Architecture

The reference architecture for LLM applications has begun to develop recently. Depending on your particular use case, an application can be built by using the following components:

#### LLM Model:

Selecting an LLM model depends on a variety of factors, but the primary elements are the number of parameters (size of the model) and whether it is open source or behind an API. It's possible for models to have tokenizers and embedded models built in, while others may require you to run these steps yourself. Tokenizers break down input text into smaller chunks, while embedding models turn those chunks into numeric vectors that LLMs can understand.

OpenAI's GPT models are an easy starting point, because they're quick to get up and running due to how adaptable the model is. Sometimes, a long context window might be needed if performing a lot of in-context learning, which makes Claude 2 a great choice. However, using such large models can be expensive and time consuming.



Trade-offs between open/closed sourced LLMs and model size

GPT-3.5 cost much less for input and output tokens\* and runs significantly faster than other Large Language Models. These advantages incentivize developers to optimize prompts to maximize its accuracy. Prompt engineering has its limitations, so you might choose to fine-tune open source models like LLama 2 for specific use cases. Implementing this approach comes with more training and deployment complexity. However, when done correctly, it can reduce costs and latency while improving accuracy.

#### Prompt Template:

Prompt engineering is a key activity for LLM applications, much like the experimentation process in machine learning. There's a wide variety of prompting techniques that can improve results. The first technique is to provide clear instructions and write the steps needed to finish a task.

“Zero-shot” prompts work, however “few-shot” prompts with examples can tailor responses even further. Phrases like “show your work” or “step-by-step” make the model reason iteratively, increasing accuracy. Using this “chain-of-thought” prompting can be even more effective when examples of reasoning logic are used.

<p><b>(a) Few-shot</b></p> <p>Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?</p> <p>A: The answer is 11.</p> <p>Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?</p> <p>A:</p> <p>⊗ (Output) The answer is 8.</p>	<p><b>(b) Few-shot-CoT</b></p> <p>Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?</p> <p>A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.</p> <p>Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?</p> <p>A:</p> <p>⊙ (Output) The juggler can juggle 16 balls. Half of the balls are golf balls. So there are 16 / 2 = 8 golf balls. Half of the golf balls are blue. So there are 8 / 2 = 4 blue golf balls. The answer is 4.</p>
<p><b>(c) Zero-shot</b></p> <p>Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?</p> <p>A: The answer (arabic numerals) is</p> <p>⊗ (Output) 8.</p>	<p><b>(d) Zero-shot-CoT (Ours)</b></p> <p>Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?</p> <p>A: Let's think step by step.</p> <p>⊙ (Output) There are 16 balls in total. Half of the balls are golf balls. That means that there are 8 golf balls. Half of the golf balls are blue. That means that there are 4 blue golf balls.</p>

Examples of prompting techniques<sup>1</sup>

<sup>1</sup>\* Kojima et al. (2022)

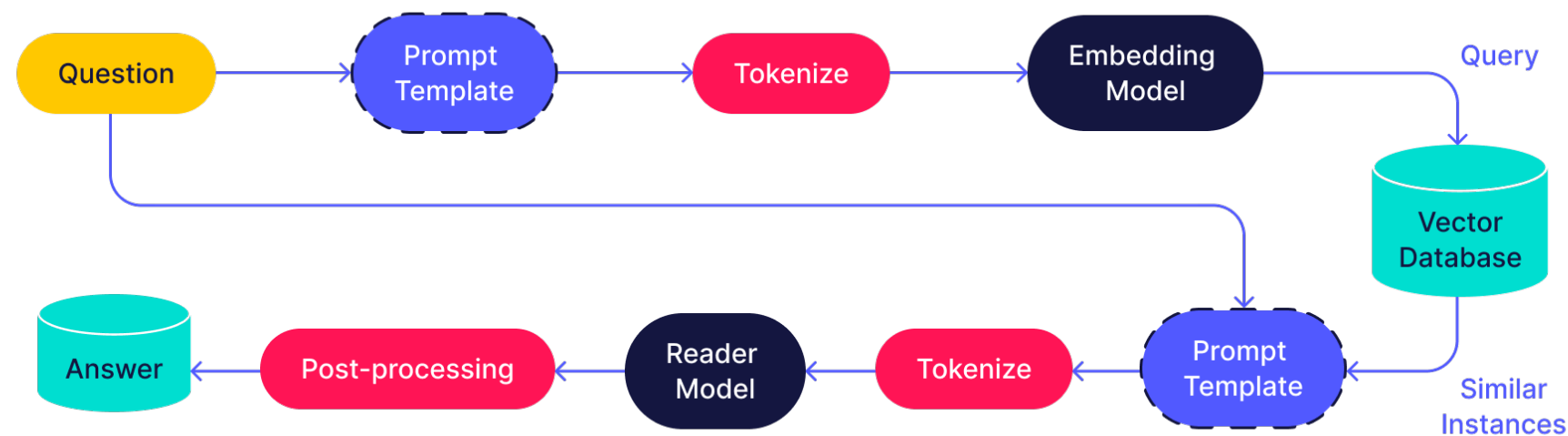


Templates can be created with your prompts and reused with the LLM application. These templates can be integrated into your code by using a simple f-string or str.format() or alternatively, libraries that offer more control like LangChain, Guidance, and LMQL can be used. Chat completion APIs rely on careful prompt engineering to perform well. First, the conversational task needs to be assigned clearly in a system prompt, so that the LLM understands its role.

Then you can give the LLM examples of user questions and high quality responses. Experiment systematically on these prompt templates to achieve improved model performance. The goal is to teach the model effectively. Evaluate model outputs by either using human feedback or automated scoring.

**Vector Database:**

Several use cases will require information the LLM hasn't been previously trained on. This could include a company's private knowledge base, current events, research papers, or other documents. A model's context window that it can directly access is limited, so for example, prompting them with an entire book's content will not work. Adding a supplementary external database can provide access to information needed. This technique is known as retrieval-augmented generation or RAG.



Simple retrieval augmented generation (RAG) flow

Vector databases are not a new concept, but have recently surged in popularity as they are used with LLM applications. In this technique, relevant external information is first divided into blocks of text. Then these blocks are tokenized, and run through an embedding model. Afterwards, they're added into a vector database, such as Pinecone, Chroma, Quadrant or pg vector.

When a prompt is given to the language model, it is also vectorized. The database finds similar vectors to retrieve relevant entries from the database. These entries are then provided along with the original prompt to give the model necessary information to respond coherently. The LLM may

cite the specific database chunks it used when generating its response. This adds a degree of trust and counters the risk of hallucinations.

**Agents and Tools:**

Large Language Models can enable powerful applications, but relying solely on them has some fundamental limitations. They aren't capable of prompting themselves, making external API calls, or retrieving web pages.

However, LLM agents can utilize additional tools to take actions beyond just generating text. For example, an LLM agent can execute code, search the web, lookup databases, or perform math calculations. The OpenAI framework allows agents to decide which tools or "functions" to use and return a JSON object with the arguments to call the function. This enables a whole range of new use cases like booking flights, generating images, or sending emails.

Some frameworks allow agents to complete tasks iteratively. They can break down tasks into subtasks and "self-ask" to gather more relevant information. Chains of actions can be executed to progressively move towards a correct answer. The concept of combining reasoning and using tools led to the emergence of the ReAct framework.

ReAct thinks, takes an action, observes the result, and decides the next step in a loop until finding a solution. This outperforms baseline LLMs, but evaluating performance and achieving reliability is still challenging. There are also security concerns with enabling LLMs to take actions like posting on the web.



ReAct REPL Agent <sup>2</sup>

<sup>2</sup> <https://peterroelants.github.io/posts/react-repl-agent/>



**Orchestrator:**

Orchestrators like LangChain and LlamaIndex tie together the various components needed for LLM applications. They provide frameworks to abstract and integrate LLMs, prompt templates, data sources, agents, and tools. Templating frameworks like Guidance and LMQL allow creating complex prompts that define inputs, outputs and rules. Orchestrators can also boost performance through memory management, token healing, beam search, session management, error handling, and more. This logical orchestration empowers developers to build differentiated, specialized, and hardened use cases.

**Monitoring:**

Given the unpredictability and rapid evolution of LLM models, you should monitor your LLM applications in production. Standard metrics like CPU usage, GPU, memory usage, latency and throughput should be tracked. Additionally, drift and anomaly detection can identify changing or unusual inputs over time. Logging requests and responses enables evaluation of potentially harmful outputs. With complex LLM pipelines, tracing data flow provides visibility into overall system behavior. Tools like LangSmith and Seldon Core v2 allow tracing prompt evolution through agent chains. Seldon Core v2 has a data-centric deployment graph that enables robust monitoring via drift detection and explainability features.

## The Challenges and Compromises in Production Deployment

So far, you've discovered the different applications LLMs can be used for, as well as what components make up the LLM architecture. Now, we're going to explore the challenges users experience when deploying LLMs within their own environments. You will learn the different techniques to overcome these challenges and make your deployments run smoother.

Customers can use third-party models like ChatGPT-4 in their applications, but relying on external endpoints has downsides. Users might have security and privacy concerns sending data to a 3rd party. The third-party model might also not fit their particular use case in terms of performance or deployment needs.

Instead, open source models like Llama2 and Falcon are now getting closer to proprietary closed source models in quality. This lets users fine-tune based on relevant data for their unique task. They can also optimize deployment specifically for their application's needs.

**LLM Inference Challenges**

LLMs have different inference characteristics than other kinds of machine learning models.

**A Sequential Token Generation**

LLM inference is autoregressive, which means that the current token generation is based on previous token generation. This sequential order causes high latency, especially when generating longer texts.

The variable output length makes latency unpredictable before execution. For example, answering "What is the capital of France?" will be much faster than summarizing a long document. This makes workload scheduling more challenging.

**B Variable Sized Input Prompts**

Users can leverage different prompt templates that are zero or few shot in order to get better results from their LLMs. The length of the different types of prompts will directly affect how much work the model needs to do to process the input. Longer prompts require more compute and memory due to the attention mechanism in transformers, so they scale quadratically as prompt length increases. Batching prompts of different lengths may also be challenging. One strategy is to include extra padding to match the longest prompt, although this isn't very efficient from a computational standpoint.

**C Low Batch Size**

Large Language Models usually have small batch sizes during inference, unlike in training and finetuning. Small batches cause low GPU usage and can create bottlenecks on IO. This isn't ideal as users want to fully utilize expensive GPUs.

**D Attention Key-Value (KV) Cache**

A common optimization for autoregressive token generation is caching previous tokens as input for the current token generation. This reduces recomputation but increases memory footprint, which can become substantial when dealing with multiple requests.

These characteristics create two main deployment challenges:

- LLMs have large memory requirements that can bottleneck deployments and affect hardware choices
- Scheduling strategies are important for user experience and hardware utilization. Depending on the application, different configuration choices will need to be made for the best optimization

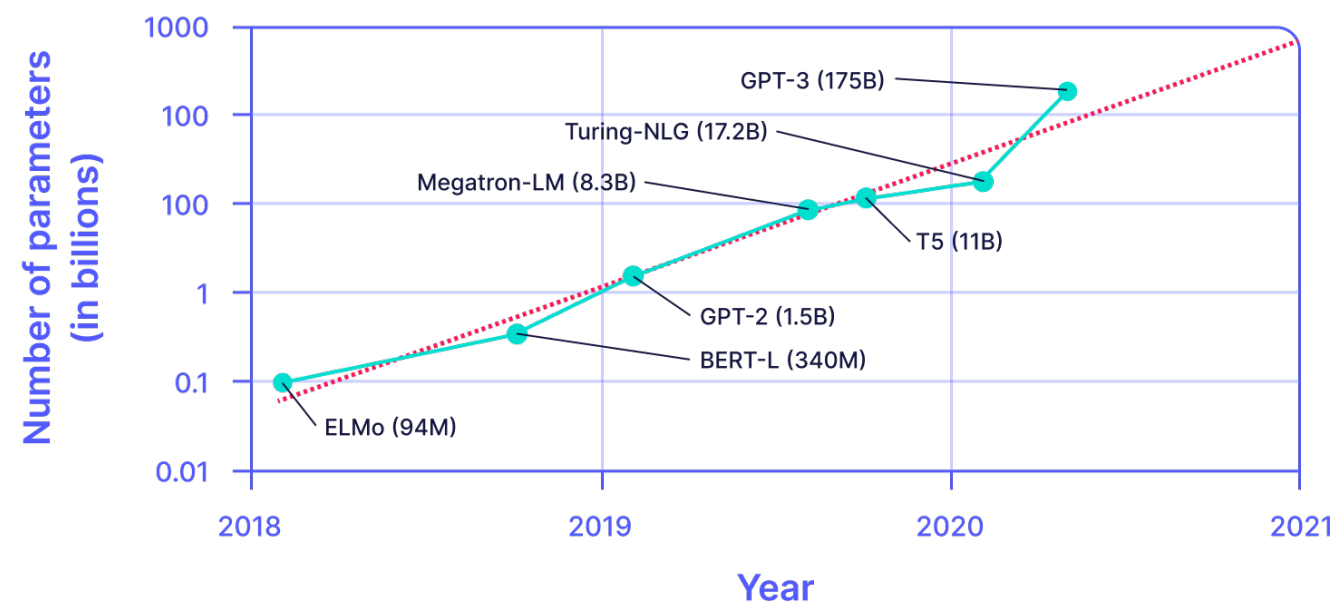


For instance, a document summarisation batch workload requires a different strategy than an interactive chat application. Careful optimization based on the use case is required to avoid overprovisioning expensive GPU hardware or hurting end users' latencies.

Optimizing LLM deployment requires balancing multiple factors in a use-case dependent manner. There is no one-size-fits-all solution.

### LLM Memory Optimizations

The current trend is for ever-larger language models, as increased parameters boost their learning potential. ChatGPT-4 is six times larger than ChatGPT-3, and is likely to be in the [trillion parameter range](#). It's not clear if this trend is going to continue, given that their memory requirements for inference become impractical on current GPUs. This requires multiple terabytes of memory which poses a major challenge. When an LLM has a trillion parameters to load at inference time, it requires at least 2 TBs of GPU HBM (assuming fp16 precision).



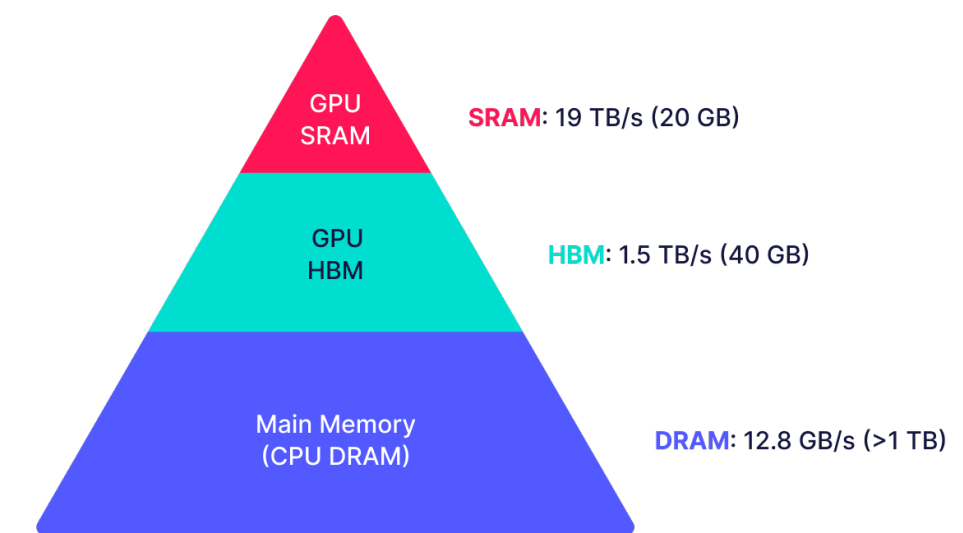
Trend of state-of-the-art NLP model sizes with time

Memory requirements for language models go beyond just model parameters. Attention layer calculation, cache management, and large prompts for in-context learning all add substantial requirements. The demand for in-context learning in LLM applications such as retrieval augmented generation (RAG) adds more pressure on the memory requirement.

For example, Claude 2 allows up to 100k tokens to be used as input prompts, which could need 1MB per token cached. Paired with several concurrent requests and big prompts, KV cache can easily get into the terabyte range.

However, capacity isn't the only concern. Memory bandwidth matters as well. Lots of data movement slows performance, particularly with small batch sizes where workloads tend to be IO bound.

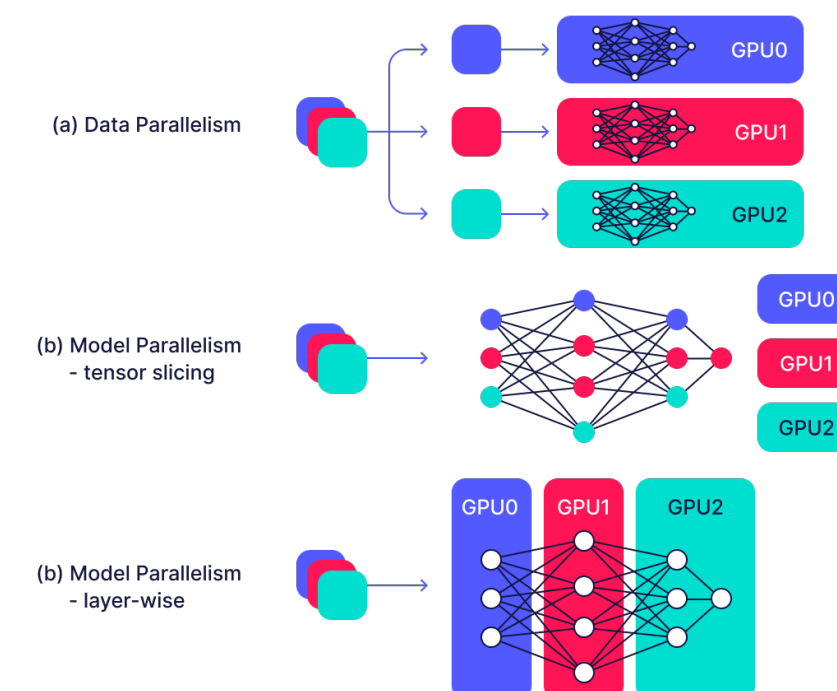
### Memory Hierarchy with Bandwidth & Memory Size



Next, we'll explore the many options that users can use to get around such a memory bottleneck.

One solution for deploying LLMs that don't fit on a single GPU is to use parallelism (e.g. sharding) by splitting the model across multiple GPUs.

### Data parallelism and model parallelism





## ① Data Parallelism (DP)

Data parallelism (DP) replicates the deployment of the model several times and splits incoming requests across the various replicas of the model. This enables the deployment to absorb a higher number of requests as they are served in parallel by the model replicas. This doesn't help oversized models fit on a single GPU. In that case, users will need to leverage model parallelism.

## ② Tensor Parallelism (TP)

Tensor Parallelism (intra-operation parallelism) is a type of strategy for model parallelism. TP splits the model horizontally, with each GPU holding a slice. Each GPU is computing a partial result for a slice of the input tensor that corresponds to the slice of the weights that is loaded. This process requires synchronization to combine partial results. This technique works best when GPUs are co-located within the same node and interconnected with high-speed links (e.g. using NVLink).

## ③ Pipeline Parallelism (PP)

PP (inter-operation parallelism) is another orthogonal approach that doesn't require a lot of synchronization among GPUs. This method divides the model vertically, with each GPU processing certain layers. This pipeline approach creates an assembly line: when one GPU is done computing the corresponding layers, the intermediate result is sent to the next GPU in the pipeline and so on.

This strategy requires less synchronization which means it can be used inter-node. However, it can suffer from low utilization and leave all but one GPU idle (bubble). There are ways to get around this bubble. One technique is staggering multiple requests at different pipeline stages across the GPUs, provided the traffic patterns permit this. This improves overall throughput.

## ④ Hybrid Parallelism

Hybrid parallelism combines techniques like DP and TP for optimization based on the model size, available hardware, traffic patterns, and use case. For example, DP and TP can be used together if the model cannot fit into one GPU (e.g. requiring at least 2 shards) and also replicated to serve a certain inference load.

Libraries like Deepspeed and Parallelfomers enable model parallelism.

## Compression (e.g. Quantization)

One strategy to lower the amount of memory an LLM requires is to use a form of compression. There are a few different techniques for compressing the models that include Distillation, Pruning, and Quantization. We will focus on quantization because this strategy can be applied at deployment time and doesn't require the model to be re-trained.

Quantization saves memory by using lower precision numbers (e.g. by using int8 (1 byte) instead of fp32 (4 bytes) to represent the model weights and activations.) With this example the model requires  $\frac{1}{4}$  of memory compared to the unquantized version. This optimization allows for deployment on fewer GPUs, making it more affordable.

Quantization is expected to have minimal impact on model performance. However, it is important to test the quantized model depending on the use cases to ensure there is no substantial regression in the quality of the results.

The conversion process in mixed-precision quantization can add overheads that creates sub-optimal inference latencies, especially for medium-sized models.

Possible libraries that can be leveraged for quantization are [Bitsandbytes](#), [GPTQ](#) and [Deepspeed](#).

## Attention Layer Optimization

Attention is at the heart of the transformer architecture, but it requires substantial memory and compute. Standard attention scales quadratically as the number of tokens grows, which requires extra GPU memory to cache intermediate (KV) results. Memory bandwidth also needs to be considered as these tensors are moved from GPU memory to the registers of the tensor cores.

Optimization techniques like [FlashAttention](#) and [PagedAttention](#) reduce the amount of data transfer in attention. This improves latency and throughput of the attention layer.

These optimizations target specific use cases like those with low batch size or executing on a single GPU. There is no universal solution for optimizing attention memory.





## Scheduling Optimizations

As we noted earlier, large language models typically have high and variable latency at inference time. Efficient scheduling is critical for LLMs otherwise the user experience will be poor and users will complain. There are tradeoffs at different levels of scheduling which we'll explore next.

### Request Level Scheduling

A standard level of scheduling is at the granularity of the particular request. In this scenario, when a request arrives at the model server, the request can be served if there is compute space for it. If there isn't, it is added to a pending requests queue to be served later. An idle server will execute requests immediately as it has available compute capacity for it. If there's subsequent requests that arrive while the first request is being served, it will cause queueing delays until the first request is completely finished.

Scheduling at the request level has some characteristics. If compute capacity is available, the request will be served as fast as possible. If this isn't addressed, considering that an LLM can take several seconds to serve a complete request, other incoming requests will incur a high latency until compute resources become available. So, head-of-line blocking results in high average latency without abundant servers. GPU utilization is also low due to small batch sizes.

### Batch Level Scheduling

Batch level scheduling improves on request scheduling by grouping requests into batches within a timeframe and treating them as a single batch for scheduling. This is also referred to as adaptive batching. The benefits of this technique are:

- Multiple requests can be served at the same time, which reduces the overall average latency compare to request level scheduling
- Larger batched inputs increase GPU utilization

However, from a scheduling perspective, batches are static. New requests arriving after a batch starts must wait for the entire batch to finish. So, the longest generation request in that batch will determine how long it takes to process a batch. This can unnecessarily delay other requests from being served.



### Iteration Level Scheduling and Continuous Batching

The techniques mentioned above have drawbacks—incoming requests wait idly for ongoing generations to complete, causing delays and increased latency.

A solution to this issue is dynamic batching at the token iteration level, also called continuous batching. With this approach, new requests join the active batch for concurrent processing while finished requests leave the batch as soon as they are done. This ensures all requests progress efficiently at the same time.

This can be achieved by leveraging iteration level scheduling, which schedules tokens one by one. After generating each token across the batch, the scheduler modifies the batch accordingly. This fine-grained dynamic batch adjustment is pioneered by [Orca](#) and also implemented in other LLM serving platforms like [vLLM](#) and [TGI](#).



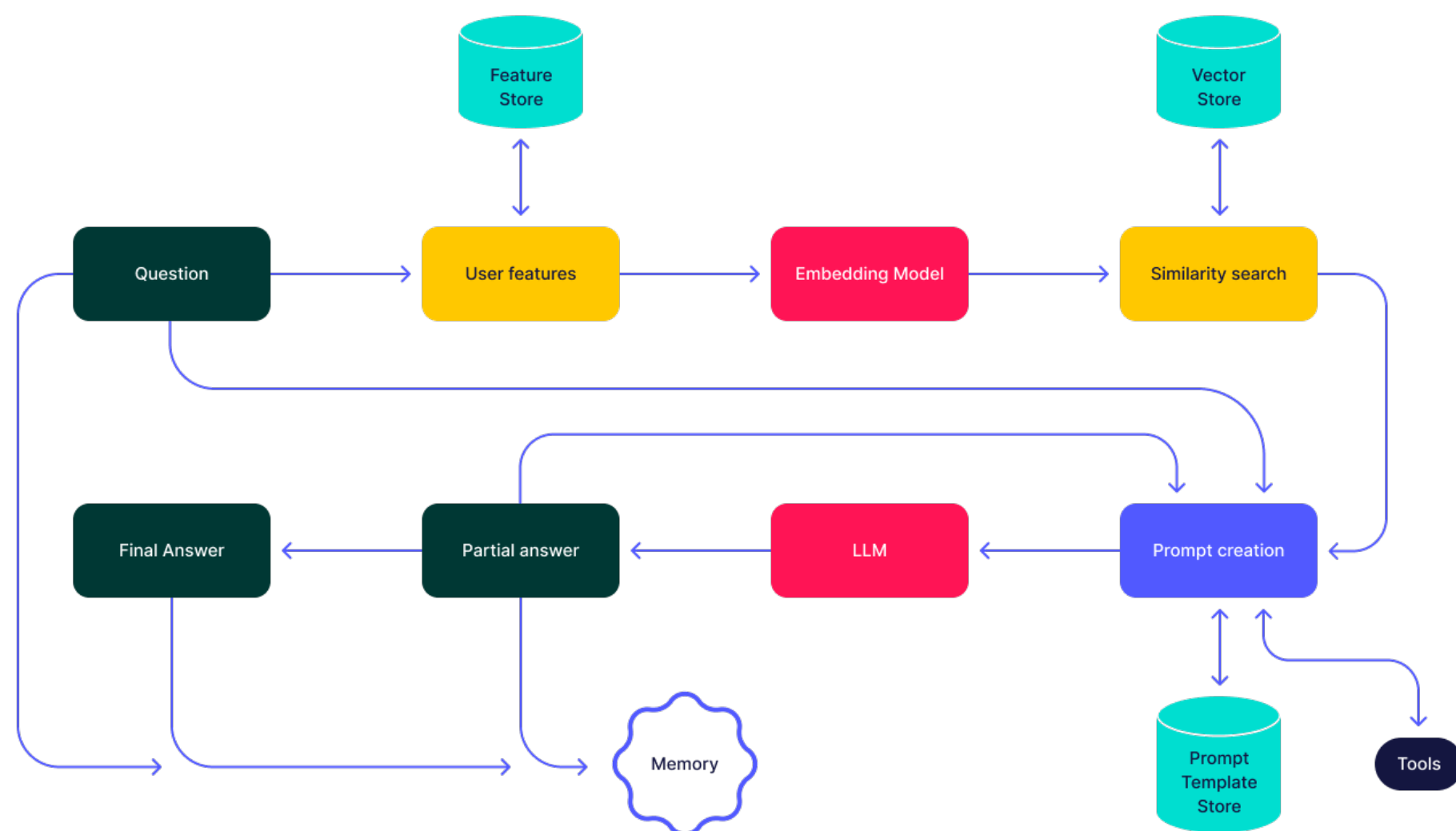


## Best Practices for Orchestrating LLMs in Production

So far, you've explored an overview of LLM architecture along with the challenges that come with deploying individual LLMs to production. To achieve success in deployment, a balance needs to be struck between cost, efficiency, latency and throughput.

Now, let's move on to discuss the key challenges when building a full production application that utilizes LLMs. In this guide, we won't focus on fine-tuning. We will assume you have modified a foundation model to fit your particular use case and can deploy it as a part of a wider application.

Let's look at an example of a document question answering system. To help you get a better understanding, the diagram below shows how the data flows through the system:



## Data Flows Explained

The flow diagram above can be summarized as follows:

1. A question comes in from a user
2. Meta data is added to the request by calling a feature store to get details about the user
3. Next, call an embedding model on the question to get one or more vectors
4. Now, a similarity search is conducted against a vector store that holds vector representations of internal documents to find a set of candidate documents
5. Once several useful documents are found, we mix them with the original question and construct a prompt. With the query in hand, we call our LLM and get an answer back
6. Afterwards, we decide whether this answer is the final answer or we need to call the LLM again, possibly with an altered prompt, while saving the state as we proceed in the loop
7. Finally, when we are confident in the result, we'll eventually return a final answer to the user

## LangChain

As you can see, building full LLM applications requires integrating a variety of tools. LangChain is the most popular all-in-one solution to manage:

- LLMs (local or via APIs)
- Prompt templating techniques
- Vector stores
- Feature stores
- Memory and state management
- Agents and tools

Although powerful, [LangChain has some challenges](#) when using it to move your application to production. It provides a large matrix of options which all need to work together. There can be unexpected problems when tools do not play nicely together or the defined logic has some unexpected assumptions. Simpler alternatives like [simpleaichat](#) (following a [blog post by the author](#)) and [minichain](#) might be easier for some use cases.

## Guided Prompting

A key subprocess is guided prompting. This is the core process of directing the LLM to solve tasks while adhering to existing knowledge and constraints. Two interesting projects in this space are [Guidance](#) and [LMQL](#).



## Guidance

Guidance offers a templating language using handlebars templating to incrementally construct prompts with associated restrictions on LLM-generated text. An example from their docs is shown below where a proverb is generated while adhering to certain restrictions:

```
import guidance

# set the default language model used to execute guidance programs
guidance.llm = guidance.llms.OpenAI("text-davinci-003")

# define a guidance program that adapts a proverb
program = guidance("""Tweak this proverb to apply to model instructions instead.

{{proverb}}
- {{book}} {{chapter}}:{{verse}}

UPDATED
Where there is no guidance{{gen 'rewrite' stop="\n-"}}
- GPT {{#select 'chapter'}}9{{or}}10{{or}}11{{/select}}:{{gen 'verse'}}""")

# execute the program on a specific proverb
executed_program = program(
    proverb="Where there is no guidance, a people falls,\nbut in an abundance of counselors there is
    book="Proverbs",
    chapter=11,
    verse=14
)
```

Tweak this proverb to apply to model instructions instead.

Where there is no guidance, a people falls,  
but in an abundance of counselors there is safety.  
- Proverbs 11:14

UPDATED  
Where there is no guidance, a model fails,  
but in an abundance of instructions there is safety.  
- GPT 11:14

The Guidance program supports parameter input and defined slots for generation with constraints. Let's highlight two interesting capabilities that Guidance has. Firstly, when Guidance is used with a local model, it can optimize inference by caching key-value (KV) attention computations. This prevents repeat computations when using previously processed prompts. An example illustrating this is filling in a character definition over repeated model calls as shown below with the generation steps shown in green.

The following is a character profile for an RPG game in JSON format.

```
```json
{
  "id": "e1f491f7-7ab8-4dac-8c20-c92b5e7d883d",
  "description": "A quick and nimble fighter.",
  "name": "Fighter",
  "age": 18,
  "armor": "plate",
  "weapon": "sword",
  "class": "fighter",
  "mantra": "I will protect the weak.",
  "strength": 10,
  "items": ["Hero's Hammer", "Fast-Healing Potion", "Magic Boots", "Shield of the Ancients", "Mystic Bow"]
}```
```

Another intriguing capability is token healing. Language models map character sequences to individual tokens, and sometimes tokens overlap with subsets of characters from other tokens. In their example, they demonstrate how a prompt generating a URL prompt that ends with a colon can lead to an invalid sequence. Guidance resolves this by modifying the prompt. It removes the colon from the prompt and ensures that any token starts with a colon as its initial character. This adjustment ensures that the model produces outputs that match the intended format, as shown below:

```
# we use StableLM as an open example, but these issues impact all models to varying degrees
guidance.llm = guidance.llms.Transformers("stabilityai/stablelm-base-alpha-3b", device=0)

# we turn token healing off so that guidance acts like a normal prompting library
program = guidance('The link is <a href="http:{{gen max_tokens=10 token_healing=False}}')
program()
```

The link is <a href="http://www.google.com/search?q

```
guidance('The link is <a href="http:{{gen max_tokens=10}}')()
```

The link is <a href="http://www.youtube.com/v/s\_

For the example above to work like it does with KV caching it requires a close integration with the inference server.



## LMQL

Moving on to LMQL, it also provides a templating language for guided generation, an example of which is shown below:

LMQL Open In Playground

```

argmax
.....
"""A list of good dad jokes. A indicates
- the punchline
Q: How does a penguin build its house?
A: Igloos it together.
Q: Which knight invented King Arthur's
- Round Table?
A: Sir Cumference.
Q: [JOKE]
A: [PUNCHLINE]"""
from
"openai/text-davinci-003"
where
len(JOKE) < 120 and
STOPS_AT(JOKE, "?") and
STOPS_AT(PUNCHLINE, "\n") and
len(PUNCHLINE) > 1
                    
```

MODEL OUTPUT

A list of good dad jokes. A indicates the punchline

Q: How does a penguin build its house?  
A: Igloos it together.

Q: Which knight invented King Arthur's Round Table?  
A: Sir Cumference.

Q: **JOKE** What did the fish say when it hit the wall?  
A: **PUNCHLINE** Dam!

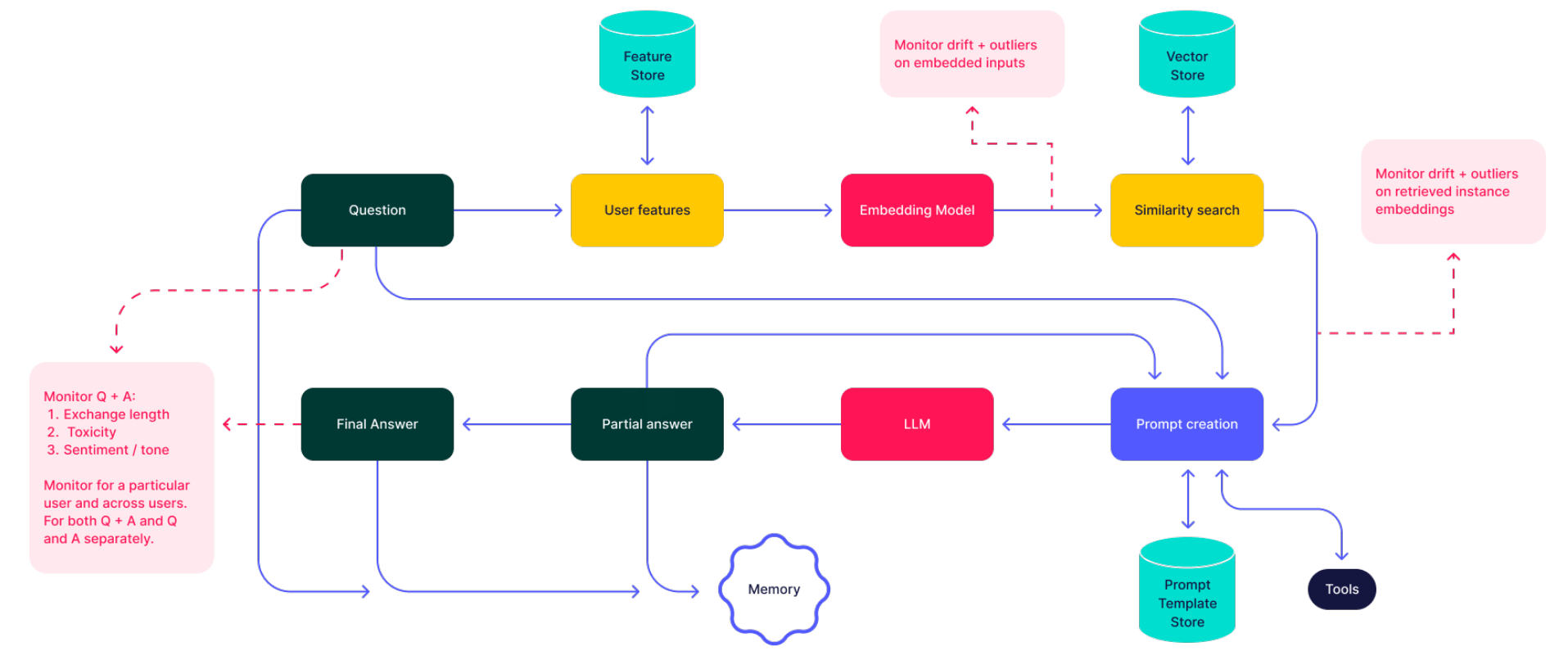
Highlighted text is model output.

LMQL allows adding constraints to control generated text. This can be seen above providing constraints that restrict length and format of the generated joke and punchline.

One interesting feature of LMQL is the ability to do scripted beam search, which searches over multiple generation points in the prompt template. Again, this requires close integration between the guidance engine and the inference server.

### Monitoring in Data Flows

Now, let's move back to the wider application data flow. When you place your application into production, the data flow diagram we showed above can be annotated with several points where monitoring is crucial for making sure there is correct and safe operation of the application. You can see this in the diagram on the next page:



The key areas highlighted are:

#### Monitor Drift

Monitoring for drift and outliers from the question embeddings and document similarity search results.

#### Monitor Q + A

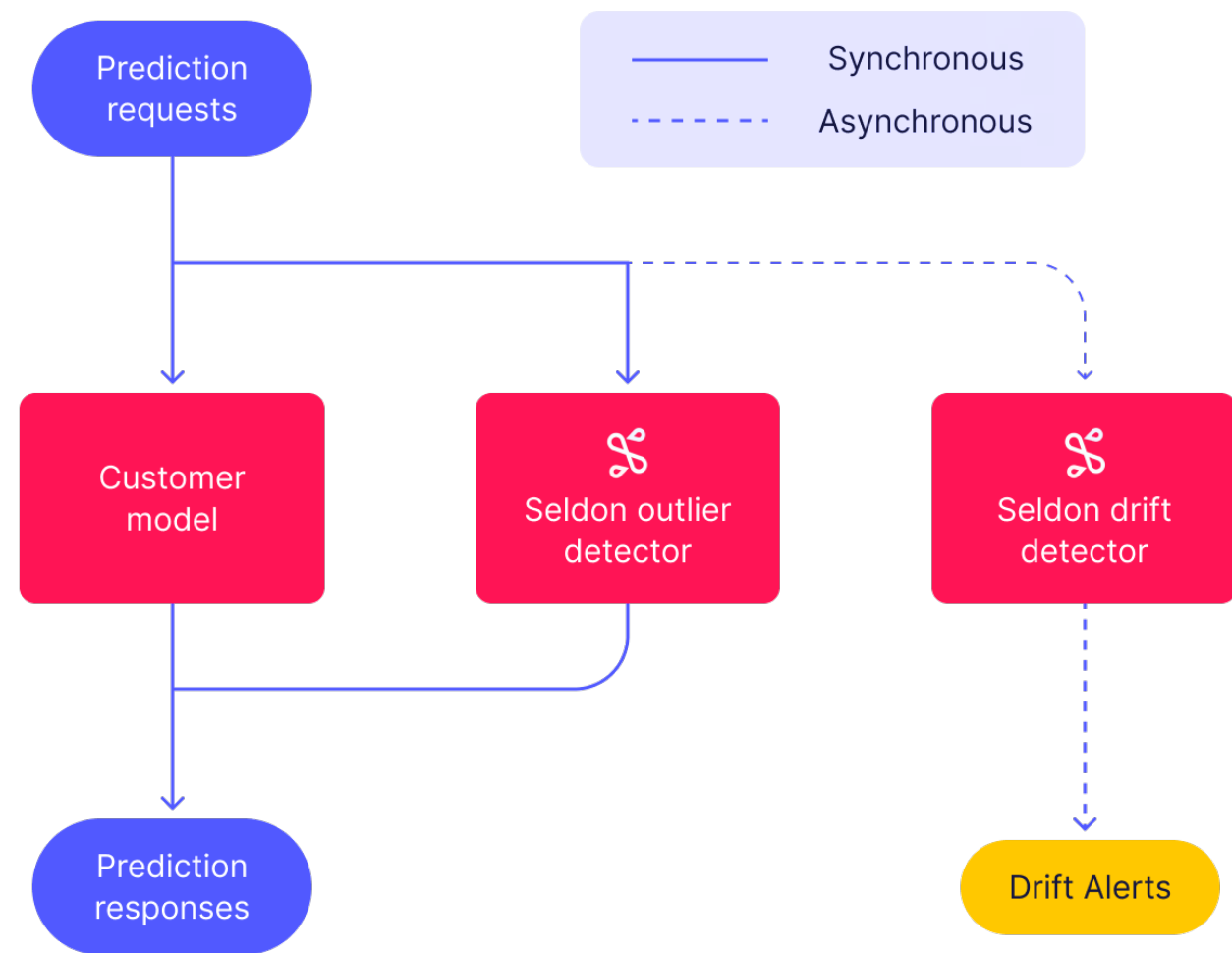
Monitoring the questions and answers in isolation for issues like: toxicity, sentiment changes, hallucinations, and conversation length.

These checks could also be incorporated into real-time inference flows to change the output of the individual request if we believe there are outliers, toxicity, or unexpected changes in sentiment.





The data flow paradigm of clearly exposing the data at any point in the transformation steps from question to answer is a key feature to Seldon's data-centric approach to machine learning pipelines. This is exemplified in Seldon Core V2, which provides the ability to create well-defined data flows containing ML models along with monitoring components.



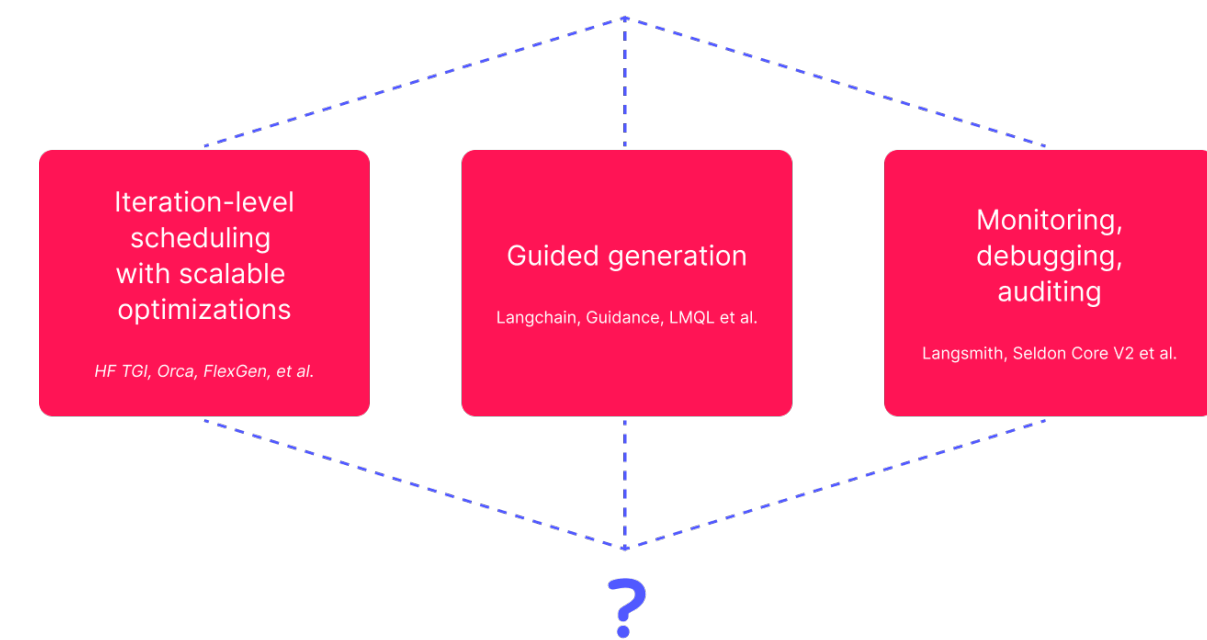
[LangSmith](#), a recently created offering by LangChain, provides a more API-based approach for post-hoc analysis. Organizations should invest in these tools to make sure their LLM-based applications are performing properly and can be clearly audited.

To sum up, production-ready LLM applications need scalable guided inference with monitoring and debugging.

The industry still has some way to progress before a solution is fully ready.

## Production LLMOps Solution?

Scalable token by token guided inference with monitoring and debugging



## Empowering your Business with LLMs

Large language Models represent a seismic shift for organizations across every industry. These powerful AI systems have demonstrated immense potential to transform workflows, enhance the customer service experience, and provide actionable insights from data. Organizations that fail to embrace LLMs as part of their business strategy risk falling behind their competition.

Deploying Large Language Models into production requires careful planning and execution to realize benefits while mitigating risks. As covered in the first section, LLMs have diverse applications ranging from content generation to search optimization when combined with the right prompts, vector database, agents, and tools.

However, challenges arise in real-world deployment around bias, safety, robustness, and cost. Without proper monitoring and guardrails, LLMs can generate toxic, incorrect, or meaningless outputs. We've provided best practices to address these challenges, including using tools such as LangChain, Guidance, or LMQL.

When thoughtfully orchestrated, LLMs become powerful assets for businesses. The key is implementing them responsibly, with governance and infrastructure to ensure quality control. Are you ready to unlock their game-changing capabilities?



## About Seldon



Seldon is the world's leading MLOps platform trusted by the most innovative MLOps teams. We are on a mission to accelerate the adoption of ML to improve business performance and manage risk. Our platform decreases time to value and drives cost savings through efficiency gains and by augmenting data science skills and workflows.

Interested in seeing why Seldon is trusted by the world's leading MLOps teams? Request a demo and an expert from our team will be happy to schedule one for you.

[Request a demo](#)

Want to chat to other ML professionals about LLMs?



[Join our Slack community](#) and post in the [#alibi-detect channel](#).

We'd love to hear from you!  
Here's how to connect with us:

[Sign up for our MLOps monthly newsletter](#)