# Deployment

In today's fast-paced technological landscape, the use of Large Language Models (LLMs) is rapidly expanding. As a result, it's crucial for developers to understand how to effectively deploy these models in production environments. LLM interfaces typically fall into two categories:

- **Case 1: Utilizing External LLM Providers (OpenAI, Anthropic, etc.)** In this scenario, most of the computational burden is handled by the LLM providers, while LangChain simplifies the implementation of business logic around these services. This approach includes features such as prompt templating, chat message generation, caching, vector embedding database creation, preprocessing, etc.

- **Case 2: Self-hosted Open-Source Models** Alternatively, developers can opt to use smaller, yet comparably capable, self-hosted open-source LLM models. This approach can significantly decrease costs, latency, and privacy concerns associated with transferring data to external LLM providers.

Regardless of the framework that forms the backbone of your product, deploying LLM applications comes with its own set of challenges. It's vital to understand the trade-offs and key considerations when evaluating serving frameworks.

## Outline

This guide aims to provide a comprehensive overview of the requirements for deploying LLMs in a production setting, focusing on:

- **Designing a Robust LLM Application Service**
- **Maintaining Cost-Efficiency**
- **Ensuring Rapid Iteration**

Understanding these components is crucial when assessing serving systems. LangChain integrates with several open-source projects designed to tackle these issues, providing a robust framework for productionizing your LLM applications. Some notable frameworks include:

- Ray Serve

- BentoML
- OpenLLM
- Modal
- Jina

These links will provide further information on each ecosystem, assisting you in finding the best fit for your LLM deployment needs.

## Designing a Robust LLM Application Service

When deploying an LLM service in production, it's imperative to provide a seamless user experience free from outages. Achieving 24/7 service availability involves creating and maintaining several sub-systems surrounding your application.

### Monitoring

Monitoring forms an integral part of any system running in a production environment. In the context of LLMs, it is essential to monitor both performance and quality metrics.

**Performance Metrics:** These metrics provide insights into the efficiency and capacity of your model. Here are some key examples:

- Query per second (QPS): This measures the number of queries your model processes in a second, offering insights into its utilization.
- Latency: This metric quantifies the delay from when your client sends a request to when they receive a response.
- Tokens Per Second (TPS): This represents the number of tokens your model can generate in a second.

**Quality Metrics:** These metrics are typically customized according to the business use-case. For instance, how does the output of your system compare to a baseline, such as a previous version? Although these metrics can be calculated offline, you need to log the necessary data to use them later.

### Fault tolerance

Your application may encounter errors such as exceptions in your model inference or business logic code, causing failures and disrupting traffic. Other potential issues could arise from the machine running your application, such as unexpected hardware breakdowns or loss of spot-instances during high-demand periods. One way to mitigate these risks is by increasing redundancy through replica scaling and implementing recovery mechanisms for failed replicas. However, model replicas aren't the only potential points of failure. It's essential to build resilience against various failures that could occur at any point in your stack.

### Zero down time upgrade

System upgrades are often necessary but can result in service disruptions if not handled correctly. One way to prevent downtime during upgrades is by implementing a smooth transition process from the old version to the new one. Ideally, the new version of your LLM service is deployed, and traffic gradually shifts from the old to the new version, maintaining a constant QPS throughout the process.

### Load balancing

Load balancing, in simple terms, is a technique to distribute work evenly across multiple computers, servers, or other resources to optimize the utilization of the system, maximize throughput, minimize response time, and avoid overload of any single resource. Think of it as a traffic officer directing cars (requests) to different roads (servers) so that no single road becomes too congested.

There are several strategies for load balancing. For example, one common method is the *Round Robin* strategy, where each request is sent to the next server in line, cycling back to the first when all servers have received a request. This works well when all servers are equally capable. However, if some servers are more powerful than others, you might use a *Weighted Round Robin* or *Least Connections* strategy, where more requests are sent to the more powerful servers, or to those currently handling the fewest active requests. Let's imagine you're running a LLM chain. If your application becomes popular, you could have hundreds or even thousands of users asking questions at the same time. If one server gets too busy (high load), the load balancer would direct new requests to another server that is less busy. This way, all your users get a timely response and the system remains stable.

## Maintaining Cost-Efficiency and Scalability

Deploying LLM services can be costly, especially when you're handling a large volume of user interactions. Charges by LLM providers are usually based on tokens used, making a chat system inference on these models potentially expensive. However, several strategies can help manage these costs without compromising the quality of the service.

### Self-hosting models

Several smaller and open-source LLMs are emerging to tackle the issue of reliance on LLM providers. Self-hosting allows you to maintain similar quality to LLM provider models while managing costs. The challenge lies in building a reliable, high-performing LLM serving system on your own machines.

### Resource Management and Auto-Scaling

Computational logic within your application requires precise resource allocation. For instance, if part of your traffic is served by an OpenAI endpoint and another part by a self-hosted model, it's crucial to allocate suitable resources for each. Auto-scaling—adjusting resource allocation based on traffic—can significantly impact the cost of running your application. This strategy requires a balance between cost and responsiveness, ensuring neither resource over-provisioning nor compromised application responsiveness.

### Utilizing Spot Instances

On platforms like AWS, spot instances offer substantial cost savings, typically priced at about a third of on-demand instances. The trade-off is a higher crash rate, necessitating a robust fault-tolerance mechanism for effective use.

### Independent Scaling

When self-hosting your models, you should consider independent scaling. For example, if you have two translation models, one fine-tuned for French and another for Spanish, incoming requests might necessitate different scaling requirements for each.

### Batching requests

In the context of Large Language Models, batching requests can enhance efficiency by better utilizing your GPU resources. GPUs are inherently parallel processors, designed to handle multiple tasks simultaneously. If you send individual requests to the model, the GPU might not be fully utilized as it's only working on a single task at a time. On the other hand, by batching requests together, you're allowing the GPU to work on multiple tasks at once, maximizing its utilization and improving inference speed. This not only leads to cost savings but can also improve the overall latency of your LLM service.

In summary, managing costs while scaling your LLM services requires a strategic approach. Utilizing self-hosting models, managing resources effectively, employing auto-scaling, using spot instances, independently scaling models, and batching requests are key strategies to consider. Open-source libraries such as Ray Serve and BentoML are designed to deal with these complexities.

## Ensuring Rapid Iteration

The LLM landscape is evolving at an unprecedented pace, with new libraries and model architectures being introduced constantly. Consequently, it's crucial to avoid tying yourself to a solution specific to one particular framework. This is especially relevant in serving, where changes to your infrastructure can be time-consuming, expensive, and risky. Strive for infrastructure that is not locked into any specific machine learning library or framework, but instead offers a general-purpose, scalable serving layer. Here are some aspects where flexibility plays a key role:

### Model composition

Deploying systems like LangChain demands the ability to piece together different models and connect them via logic. Take the example of building a natural language input SQL query engine. Querying an LLM and obtaining the SQL command is only part of the system. You need to extract metadata from the connected database, construct a prompt for the LLM, run the SQL query on an engine, collect and feed back the response to the LLM as the query runs, and present the results to the user. This demonstrates the need to seamlessly integrate various complex components built in Python into a dynamic chain of logical blocks that can be served together.

## Cloud providers

Many hosted solutions are restricted to a single cloud provider, which can limit your options in today's multi-cloud world. Depending on where your other infrastructure components are built, you might prefer to stick with your chosen cloud provider.

## Infrastructure as Code (IaC)

Rapid iteration also involves the ability to recreate your infrastructure quickly and reliably. This is where Infrastructure as Code (IaC) tools like Terraform, CloudFormation, or Kubernetes YAML files come into play. They allow you to define your infrastructure in code files, which can be version controlled and quickly deployed, enabling faster and more reliable iterations.

## CI/CD

In a fast-paced environment, implementing CI/CD pipelines can significantly speed up the iteration process. They help automate the testing and deployment of your LLM applications, reducing the risk of errors and enabling faster feedback and iteration.