Efficiently Serve Large Language Models (LLMs) with OpenVINO™ Model Server

1. Executive Summary

Large language models (LLMs) have become essential in various AI applications, but their significant memory and resource demands limit their deployment on consumer devices. To address this, LLMs are often hosted on servers, allowing users to access their capabilities via web interfaces without needing high-end hardware. OpenVINO™ Model Server (OVMS) provides a simple and efficient solution for serving LLMs, with the following benefits:

- **Continuous Batching and Paged Attention**: State-of-the-art techniques that significantly improve latency and throughput for LLMs.

- **Model Optimization and Weight Compression**: Tools for converting and compressing models to efficient formats (FP16, INT8, INT4), enabling easy optimization and deployment.

- **Broad Model Support**: Compatibility with most frameworks and LLMs available on HuggingFace Hub.

- **Pre-Optimized Models**: Availability of pre-converted and compressed models for immediate use.

By leveraging OVMS and OpenVINO™, companies can efficiently deploy and manage high-performance LLMs, achieving significant performance gains without additional hardware investments. This white paper defines the key performance metrics for hosted LLM applications, explains techniques for batched inference with LLMs, gives step-by-step instructions for building an LLM container with OVMS, and shares LLM performance benchmarks from the latest OpenVINO™ 2024.3 release.

2. Introduction to Serving LLMs

Large language models (LLMs) are becoming popular in a variety of AI applications, but their large memory footprint and resource requirements [1] prohibit them from being deployed on consumer devices. To overcome this roadblock, LLMs can be hosted on cloud servers where users can submit queries and receive responses. ChatGPT is a popular example of a served LLM: people use a web browser interface to send questions from any device, and a high-power server hosts the model to run inference and return a response. This allows people to utilize LLMs without needing expensive hardware.

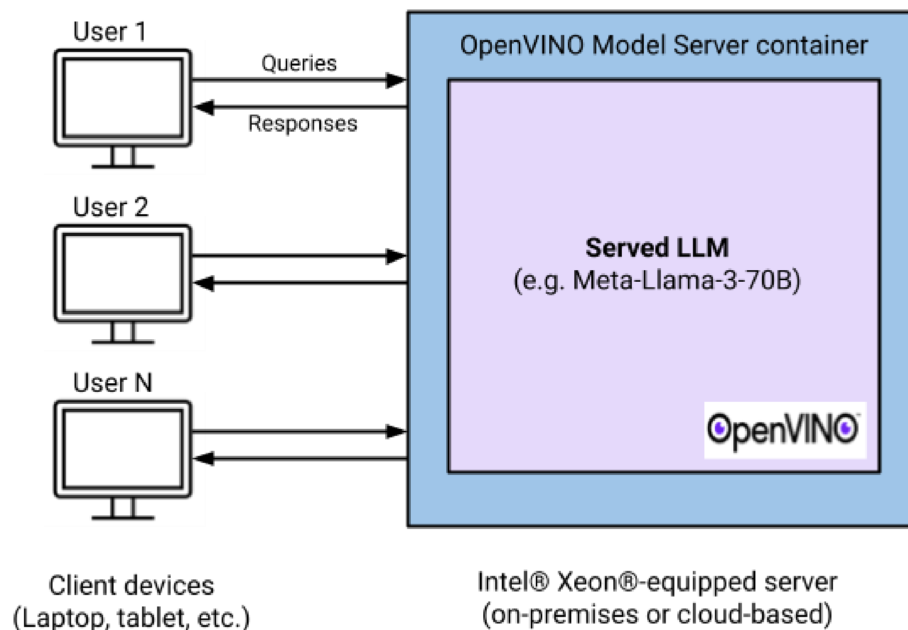

Figure 1. A server-hosted LLM can process queries from multiple users.

As AI-enabled applications grow in popularity, companies across all industries are working to create hosted services where customers and users can interact with trained LLMs. Use cases include providing chatbot Q&A support, code assistants, writing assistants, and more. Over two-thirds of companies are currently using or exploring the use of generative AI in their workflows and service offerings.

Example Use Case: Acme Insurance Chatbot Agent

Let's consider an example use case with Acme Insurance, a health insurance company made up for the sake of this white paper. They want to create an AI chatbot agent that answers support questions and helps potential customers decide which insurance policy best matches their needs. They've fine-tuned a LLM on their support documentation or set it up for retrieval-augmented generation [2] so it has specific knowledge about the product and documentation. Now, they desire to host this LLM on the cloud so users can interact with it through an app or web browser.

Models can be externally hosted on cloud services like AWS or self-hosted on local servers. Frontend tools like Gradio [3] or the OpenAI API can be used to provide a chat interface to a hosted model. Users connect to the chat interface and send a message to the model. The model processes the question and begins sending a response word-by-word. Depending on the application, thousands of users may be simultaneously submitting messages and receiving responses. Acme Insurance wants the application to support at least 50 active simultaneous users.

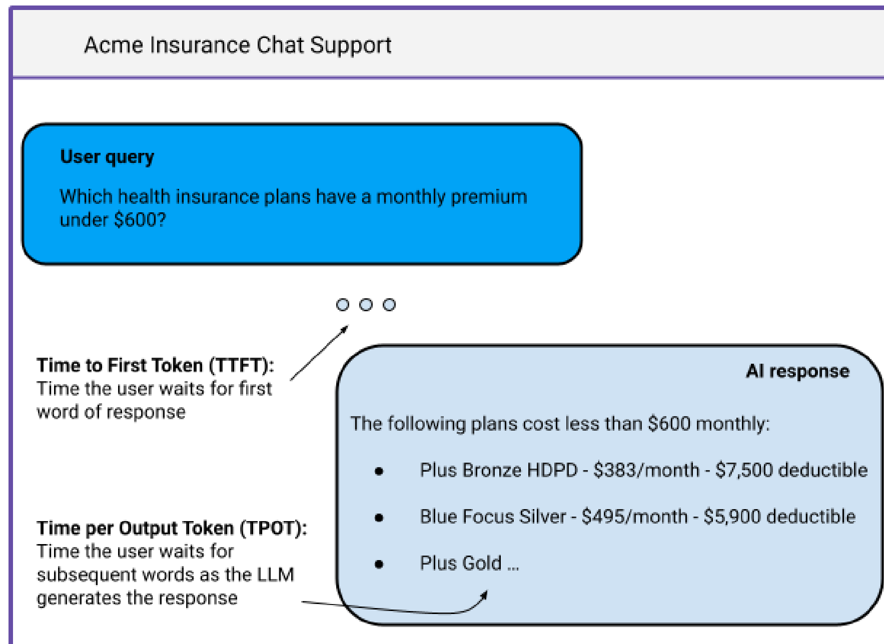

Figure 2. Typical chatbot application where latency (TTFT and TPOT) is an important metric that dictates the total amount of time a user has to wait for a complete response.

There are several important performance metrics to consider when serving an LLM.

- **Request rate (requests/s):** the cumulative amount of requests being submitted the served LLM across all active users. This depends on the average number of active users and the average time between requests.

  Request rate = Average # of active users * Average requests per second from one user

- **Latency:** the time between when a user sends a message and begins receiving the initial and subsequent words of a response. This affects the perceived responsiveness of the application. If latency is too high, the user may feel the chat service is laggy or that it takes too long to receive a complete response. There are two metrics to describe latency.

  Time To First Token or TTFT (ms): The amount of time between the LLM receiving a

request and when it generates its first token in response

Time Per Output Token or TPOT (ms): The average amount of time between subsequent tokens of the response

Consider a user's interaction with ChatGPT. When a message is sent, there's a brief wait time before receiving the first word of response (TTFT). Then, words are generated at an average rate (TPOT) until the response is completed. The average output rate dictates how long it will take to finish its response.

- **Throughput (tokens/s):** the cumulative number of output tokens generated by the model per second. This is the total number of output tokens the model can generate across all user sessions, and can be thought of as the "bandwidth" of the model. The higher the throughput, the more users can be served.

- **Memory requirement (GB):** the memory required to store the LLM's weights and intermediate tensor values as it performs inference on a message. LLMs require a significant amount of memory, which often necessitates the use of high-RAM server instances and/or high-RAM GPUs. The lower the memory requirement, the less needs to be spent on server hardware.

- **Server cost and number of nodes ($):** Hosting and running inference with top-performing LLMs requires high compute resources. Generally, latency and throughput can be improved by paying for servers with better hardware or by deploying more server nodes (either on-premises or in the cloud). However, there are techniques that give significant performance improvements without upgrading hardware - read on in this paper to learn more!

The above KPMs all must be considered when designing and deploying a served LLM application. Companies should define maximum acceptable latency times and then design the system to provide enough throughput without exceeding that latency. (Typically, latencies of 50 - 100 ms between tokens are good enough for a user.)

For example, Acme Insurance may choose to set 100ms as their max latency specification for 250 active users. This will dictate the number of server nodes they will need to set up for that application. Section 5 of this white paper provides OpenVINO™ benchmark data for these metrics and Section 5.3 discusses how to use the data to choose design parameters.

3. Techniques for Serving LLMs

This section explains a range of techniques for serving LLMs. It starts with basic techniques, steps through more advanced options, and ends with state-of-the-art techniques such as the paged attention algorithm.

3.1. Single Inference Requests

When a single user is interacting with a LLM, the process is fairly straightforward. The user sends an input prompt, the words in the prompt are converted to tokens, and the model ingests the prompt by doing a single forward pass to pre-compute the key-value (KV) cache. The KV cache holds all the key and value calculations from attention layers inside the neural network. Many of these values do not change on each inference loop, so they are stored in memory to avoid redundant calculations. After the prefill phase is completed, the model enters a text-generation loop where it continuously does forward passes to predict subsequent words (tokens). It continues this loop until a maximum sequence length is reached or an "end of sequence" token is generated. For a more in-depth explanation of the process, visit the Generation with LLMs article from HuggingFace [4].

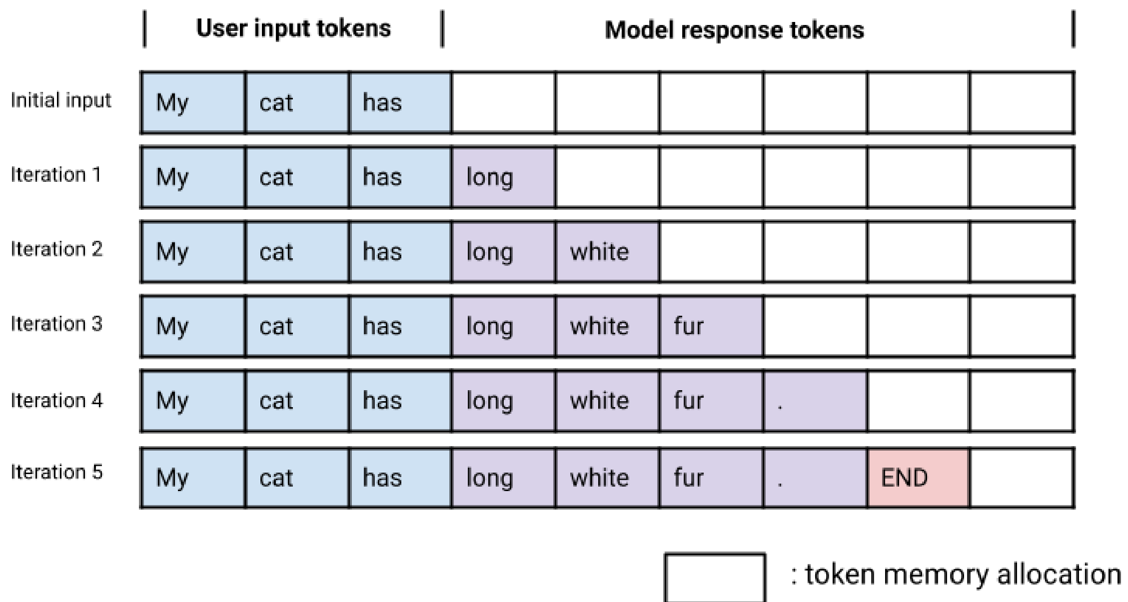

Figure 3. A LLM iteratively generates the next tokens of an input sequence. Generation stops when the model generates an end-of-sequence token. Each token requires a unit of memory allocation.

The memory required to perform inference on a single inference input primarily consists of the model's parameters and its KV cache. A model with 7 billion parameters (e.g., Llama 2 7B) in BF16 format requires roughly 14GB of memory to store the parameter values. The KV cache size varies depending on the model architecture and length of the input sequence. For a Llama 2 7B model, it takes about 0.5MB per token [5], so a 2,048-token input requires about 1GB of

memory. Thus, this model requires roughly 14GB +1GB = 15GB of system memory to handle a single inference request.

Processing single inference requests one at a time results in low latency, but it does not provide very high throughput. It does not take full advantage of the hardware's parallel processing capabilities. If multiple users are making requests to the model, they will experience slow response times while they wait for other requests to be completed. As a result, single-batch inference is almost never used when serving LLMs. Instead, throughput can be improved using batched inputs so multiple requests are processed simultaneously.

3.2. Batched Inference Requests

A hosted LLM will receive constant requests from the users interacting with it. The total request rate for a served model can be estimated by [avg_number_of_users] * [avg_user_request_rate]. For example, if an app has 50,000 users and each user makes 10 requests per day, that results in 500,000 requests per day or 5.79 requests per second. To avoid excessive latency on these requests, the served model must have high throughput. One method for increasing model throughput is by using batched inference.

Batched inference takes advantage of parallel compute capabilities by processing multiple requests at once. The model parameters are loaded into memory once, and multiple input sequences are ingested into the model. This relieves the memory bandwidth bottleneck, because rather than loading and unloading the KV caches 16 times for 16 inference requests, it only loads the KV caches once for 16 inference requests.

While batched inference improves throughput, it also requires more system memory. Instead of holding the KV cache for one input sequence in memory, the system now has to hold multiple KV caches in memory. For the Llama 2 7B example, a 2,048-token input requires 1GB of memory. If there are 16 requests, it requires 16GB of memory, bringing the total requirement to 14GB + 16GB = 30GB.

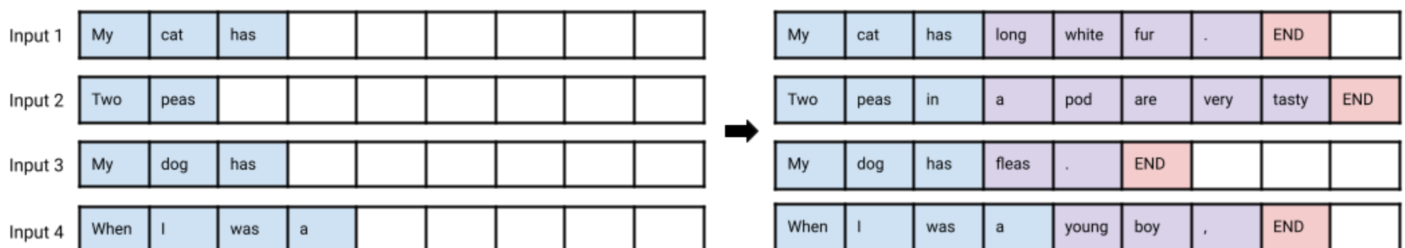

Figure 4. Generating responses to a batch of four user inputs, where some responses are different in length. The system reserves memory and processing resources for all four inputs, so some resources are left idle while waiting for the longer responses to finish.

This traditional method of batched inference is called “static batching”. The model waits for a fixed-size batch of requests to come in, and then starts processing them. There are two drawbacks to this method. First, it increases overall latency because the model needs to wait for a full batch of requests to arrive before executing inference.

Second, it still doesn’t fully utilize the compute resources of the hardware. This is due to the variable nature of the LLM’s iterative text generation process. The responses generated for some inputs may only be a few tokens long, while other responses may be hundreds of tokens long. If all inputs are processed as a batch, the shorter responses will finish before the longer responses. While the longer responses are still being processed, the hardware cores dedicated to computing the shorter responses will sit idle.

This is tough! Both the single inference and batched inference methods have their own drawbacks and don’t fully utilize the hardware. High-RAM GPU servers are expensive, so developers were motivated to find ways to get the most out of the underlying server hardware. This led to the creation of another method: continuous batching.

3.3 Continuous Batching

Continuous batching improves hardware utilization for a hosted LLM by ensuring that all compute cores are continuously saturated with data to process. Rather than waiting for a full batch of requests to be completed before, it immediately starts inference on a new request once a previous request has finished processing. The processing required to schedule and mix new requests into the batch is minimal compared to the processing needed to inference the requests. Continuous batching ensures all cores of the processor are always utilized.

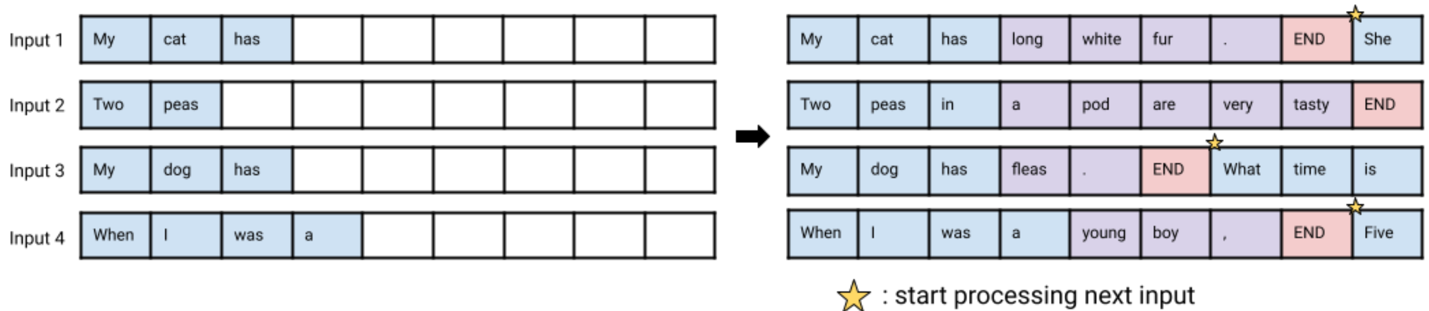

Figure 5. Using continuous batching to begin processing new inputs immediately after previous responses have completed. This improves processor utilization, because the system no longer has to wait for all responses to complete and resources can remain fully saturated.

While continuous batching helps with processor utilization, it doesn’t provide much benefit with memory requirements. When a new request is pre-filled, the system must allocate enough memory to hold the full KV cache for that request. The amount of memory that must be reserved is dependent on the sequence length. Sequence lengths are highly variable, so efficiently managing the memory is challenging. Existing systems waste 60% – 80% of memory due to fragmentation and over-reservation [6]. The latest state-of-the-art technique that addresses this

issue is dynamic KV cache memory management, or paged attention, first introduced in the vLLM paper [7].

3.4. Dynamic KV Cache Memory Management

Memory allocation issues can be solved by breaking the KV cache for a request into blocks, where each block contains the keys and values for a set number of tokens. When attention is being computed during inference, the blocks can be dynamically fetched into memory without needing to reserve a full chunk of memory for every token of the request.

The memory blocks do not need to be contiguous, so they can be managed more flexibly. Their management is similar to that of an OS's virtual memory, where bytes are stored in pages that can be dynamically loaded as-needed by processes. With paged attention, tokens are stored in virtual blocks that can be dynamically loaded during inference. Virtual blocks are mapped to physical blocks in memory, and these are allocated on demand as new tokens are generated.

The paged attention algorithm reduces memory usage by up to 55% [6]. This can result in up to 2.2x increased throughput for served LLMs. OpenVINO™ Model Server allows developers to implement continuous batching and paged attention for served LLMs. Section 4 shows how to set up continuous batching with OVMS and explains the benefits provided by OpenVINO™.

4. How to Serve LLMs with Continuous Batching in OpenVINO™ Model Server

OpenVINO™ Model Server (OVMS) is a high-performance system for serving deep learning models that supports LLMs. It allows them to be hosted on local or remote servers and accessed by software applications over standard APIs such as the OpenAI API. It provides its own implementation of continuous batching and paged attention so LLMs can be efficiently served at high throughput. Serving models with OVMS also unlocks many of the other benefits provided by OpenVINO™, such as weight compression and support for a wide range of models.

4.1. Benefits Provided by OpenVINO™ and OVMS

There are several benefits from using OVMS to serve LLMs.

Continuous Batching and Paged Attention
The 2024.3 release of OpenVINO™ includes the continuous batching and paged attention techniques discussed in Section 3.3 and Section 3.4 for LLMs being served with OVMS. It supports advanced features like chunked prefill and prefix caching while leveraging OpenVINO™'s optimized kernels to provide the best performance on Intel hardware. These updates allow OVMS to achieve better latency.

Model Optimization and Weight Compression
OpenVINO™ tools like Optimum-Intel and NNCF make it easy to download language models from HuggingFace, convert them to OpenVINO™ IR format, and compress them to FP16, INT8, or INT4 precision. Optimized models can easily be deployed in a Docker container using OVMS.

Model Support
OpenVINO™ supports many of the LLMs available on HuggingFace Hub, and it is compatible with models from a wide range of frameworks. Developers can quickly take models from their favorite framework and deploy them with OpenVINO™.

Pre-Optimized Models
The OpenVINO™ Hugging Face repository has several popular LLMs that have been pre-converted to OpenVINO™ IR format and compressed to FP16, INT8, or INT4. Visit the repository for a list of pre-converted LLMs. These are some of the popular options:

- zephyr-7b-beta-int8-ov
- phi-2-int8-ov
- mixtral-8x7b-Instruct-v0.1-int4-ov

4.2. Step-by-Step Instructions to Serve LLMs with OVMS

The following demo shows how to deploy LLMs in OVMS with continuous batching and paged attention algorithms. It provides step-by-step instructions showing how to build an OVMS Docker image, convert and compress an LLM, and launch it in a Docker container. It gives a brief Python example showing how to query and interact with the model using the OpenAI API. This guide is also available on the OpenVINO™ documentation.

This guide is written for Ubuntu, but OpenVINO™ Model Server supports other distributions of Linux, including Red Hat. The demo was tested on an Intel® Xeon® Platinum 8480+ running Ubuntu 22.04. The demo uses the Meta-Llama-3-8B-Instruct model but can be modified to use other LLMs like the pre-optimized models in the OpenVINO™ repository on Hugging Face Hub.

The best performance for this demo will be achieved on 4th-generation or newer Xeon Scalable processors with built-in AI acceleration from AMX.

1. Download the Docker Image

Install Docker Engine following the instructions on its installation page. Then, run the following command to pull the pre-built image.

```
docker pull openvino/model_server:latest
```

This will pull the latest pre-built container image called openvino/model_server:latest.

2. Prepare Model and Configure Server

In this step, the Meta-Llama-3-8B-Instruct model from Hugging Face will be downloaded, converted to OpenVINO™ IR format, and have its weights compressed. Weight compression provides faster initialization time, better performance, and lower memory consumption.

Create a virtual environment, activate it, then install the Python dependencies for downloading and converting the model using the following commands.

```
python3 -m venv ovms_env

source ovms_env/bin/activate

pip3 install -r
https://raw.githubusercontent.com/openvinotoolkit/model_server/releases/2024/3/demos/c
ontinuous_batching/requirements.txt
```

Before downloading the model, access must be requested. Follow the instructions on the HuggingFace model page to request access. When access is granted, create an authentication token in the HuggingFace account -> Settings -> Access Tokens page. Issue the following command and enter the authentication token.

```
huggingface-cli login
```

Download and convert the model using the following Optimum CLI command. This will automatically run OpenVINO™ NNCF to compress the model's weights to INT8 format. To learn more about converting LLMs to OpenVINO™ IR format using Optimum, visit the [Optimum Inference with OpenVINO™](#) page.

```
git clone https://github.com/openvinotoolkit/model_server.git
```

```
cd model_server/demos/continuous_batching
```

```
optimum-cli export openvino --disable-convert-tokenizer --model meta-llama/Meta-Llama-3-8B-Instruct --weight-format fp16 Meta-Llama-3-8B-Instruct
```

```
convert_tokenizer -o Meta-Llama-3-8B-Instruct --with-detokenizer --skip-special-tokens --streaming-detokenizer --not-add-special-tokens meta-llama/Meta-Llama-3-8B-Instruct
```

When the process is finished, the Llama-3 model and its tokenizer will be located in the `Meta-Llama-3-8B-Instruct` folder. A pre-defined MediaPipe graph file is provided in the demos/continuous_batching folder. Copy the graph.pbtxt file into the model folder by issuing the following command. Use the `cat` command to display the file for more information about the node configuration.

```
cp graph.pbtxt Meta-Llama-3-8B-Instruct/
```

```
cat Meta-Llama-3-8B-Instruct/graph.pbtxt
```

In the `graph.pbtxt` file, the default configuration of the `LLMExecutor` should work in most cases, but the parameters can be tuned inside the node_options section. The `models_path` parameter in the graph file can be an absolute path or be relative to the `base_path` from `config.json`.

The `config.json` file provides information about the model name and location. It will be attached to the Docker container when it is launched. To view the file's contents, use `cat config.json`.

3. Launch Docker Container

Now that the model and Docker container are configured, launch the container using the following command.

```
sudo docker run -d --rm -p 8000:8000 -v $(pwd)/:/workspace:ro openvino/model_server:latest --port 9000 --rest_port 8000 --config_path /workspace/config.json
```

The container and model may take a minute to initialize, so wait for it to finish loading. Check its status with the curl command below. When the model is ready, it will return the output shown below.

```
curl http://localhost:8000/v1/config

# Response:
{
"meta-llama/Meta-Llama-3-8B-Instruct" :
{
 "model_version_status": [
  {
   "version": "1",
   "state": "AVAILABLE",
   "status": {
    "error_code": "OK",
    "error_message": "OK"
   }
  }
 ]
}
```

4. Stream Output from Model

A chat stream with the model can be created using the OpenAI API. First, install the OpenAI library:

```
pip3 install openai
```

The Python example below shows how to create a stream, send a message, and receive a response. Run this code from the client device (in this demo, the client is the PC and the server is the Docker container hosted on localhost:8000).

```python
from openai import OpenAI

client = OpenAI(
  base_url="http://localhost:8000/v3",
  api_key="unused"
)

stream = client.chat.completions.create(
    model="meta-llama/Meta-Llama-3-8B-Instruct",
    messages=[{"role": "user", "content": "Say this is a test"}],
    stream=True,
)
for chunk in stream:
    if chunk.choices[0].delta.content is not None:
        print(chunk.choices[0].delta.content, end="", flush=True)
```

The model's output will be printed in the terminal. Try changing the `"content"` parameter to test other inputs. The length of the model's output can be adjusted using the `max_tokens` parameter.

5. Benchmark Model

OpenVINO™ Model Server employs efficient parallelization for text generation. It works well for models shared by multiple clients so text can be generated with high concurrency. The benchmarking app from the vLLM repository can be used to test a model hosted on OVMS. Run the commands below to set up and run benchmarking. (These commands can be run from the same `ovms_env` environment activated in Step 1.)

```
cd ~

git clone https://github.com/vllm-project/vllm

cd vllm

pip3 install -r requirements-cpu.txt

cd benchmarks

wget
https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered/resolve/main
/ShareGPT_V3_unfiltered_cleaned_split.json

python benchmark_serving.py --host localhost --port 8000 --endpoint
/v3/chat/completions --backend openai-chat --model meta-llama/Meta-Llama-3-8B-Instruct
--dataset ShareGPT_V3_unfiltered_cleaned_split.json --num-prompts 1000 --request-rate
inf
```

The program will run 1000 prompts (from the ShareGPT_Vicuna_unfiltered dataset) on the hosted model. When completed, it will output the following data.

```
Namespace(backend='openai-chat', version='N/A', base_url=None, host='localhost',
port=8000, endpoint='/v3/chat/completions',
dataset='ShareGPT_V3_unfiltered_cleaned_split.json', model='meta-llama/Meta-Llama-3-
8B-Instruct', tokenizer=None, best_of=1, use_beam_search=False, num_prompts=1000,
request_rate=inf.0, seed=0, trust_remote_code=False, disable_tqdm=False,
save_result=False)
Traffic request rate: inf
100%|██████████████████████████████████████████████████| 1000/1000 [17:17<00:00,
1.04s/it]
============ Serving Benchmark Result ============
Successful requests:                     1000
Benchmark duration (s):                  447.62
Total input tokens:                      215201
Total generated tokens:                  198588
```

```
Request throughput (req/s):                 2.23
Input token throughput (tok/s):             480.76
Output token throughput (tok/s):            443.65
---------------Time to First Token----------------
Mean TTFT (ms):                             171999.94
Median TTFT (ms):                           170699.21
P99 TTFT (ms):                              360941.40
-----Time per Output Token (excl. 1st token)------
Mean TPOT (ms):                             211.31
Median TPOT (ms):                           223.79
P99 TPOT (ms):                              246.48
==================================================
```

The request rate and number of prompts can be adjusted using the `--request-rate` and `--num-prompts` parameters. Experiment with various request rates to find the best Time To First Token (TTFT) and Time Per Output Token (TPOT) for the application. The metrics are discussed further in the next section.

5. Performance Benchmarks for LLMs on OVMS

This section provides benchmark results from testing several popular LLMs with OpenVINO™ Model Server using continuous batching and paged attention. The benchmarks focus on measuring latency and throughput. The following metrics are measured at various request rates:

- Latency: Time to First Token (s)
- Latency: Time per Output Token (s)
- Throughput: Cumulative tokens per second

Read Section 2. Introduction to Serving LLMs for a full definition of these terms. Ultimately, they help to determine how many simultaneous users can be supported by a served LLM and how much time the users will have to wait for a response.

5.1. Methodology

The setup for the OVMS Benchmark Client is intended to simulate the conditions of a hosted LLM application serving multiple concurrent users. The hosted model receives many prompts from the users, processes them in batches, and returns responses to each user. The request rate is assumed to be a fixed average, but the token length of the prompts and responses will vary.

Request Rate
The benchmark client controls the frequency of user requests made to the served LLM. This variable can be adjusted to see how the system performs at different average request rates (0.2 requests/s, 0.6 requests/s, etc.). The results shown below are gathered from testing at different request rates.

Input Prompts
To simulate the randomness of user input prompts, the benchmark routine randomly selects samples from a database of prompts (the ShareGPT_Vicuna_unfiltered dataset).

Output Responses
The length of the LLM's output response will vary significantly depending on the input prompt. (For example, the response to "What day is Christmas celebrated?" is much shorter than "Please summarize the reign of the Roman Emperor Constantine.") To simulate random response lengths, the benchmark app samples a value from an exponential distribution with mean=128, and then uses that value as the max token length for a given response. Essentially, this means the model tends to generate more short responses, but occasionally generates long responses.

Models and Precisions Tested
The following models and precisions were benchmarked:

- meta-llama/Llama-2-7b-chat-hf INT8-CW
- meta-llama/Meta-Llama-3-8B-Instruct INT8-CW
- mistralai/Mistral-7B-v0.1 INT8-CW

Frameworks Used
Benchmarking is run using OpenVINO™ 2024.3 release OVMS Benchmark Client. Models are hosted in a Docker-deployed OVMS instance.

Hardware

The following hardware was used for benchmarking. See Appendix 2 for configuration details.

- Intel® Xeon® Platinum 8480+ CPU
- Intel® Xeon® Platinum 8580 CPU

5.2. Results

The graphs below show the benchmarking results for each model and hardware at various requests rates.

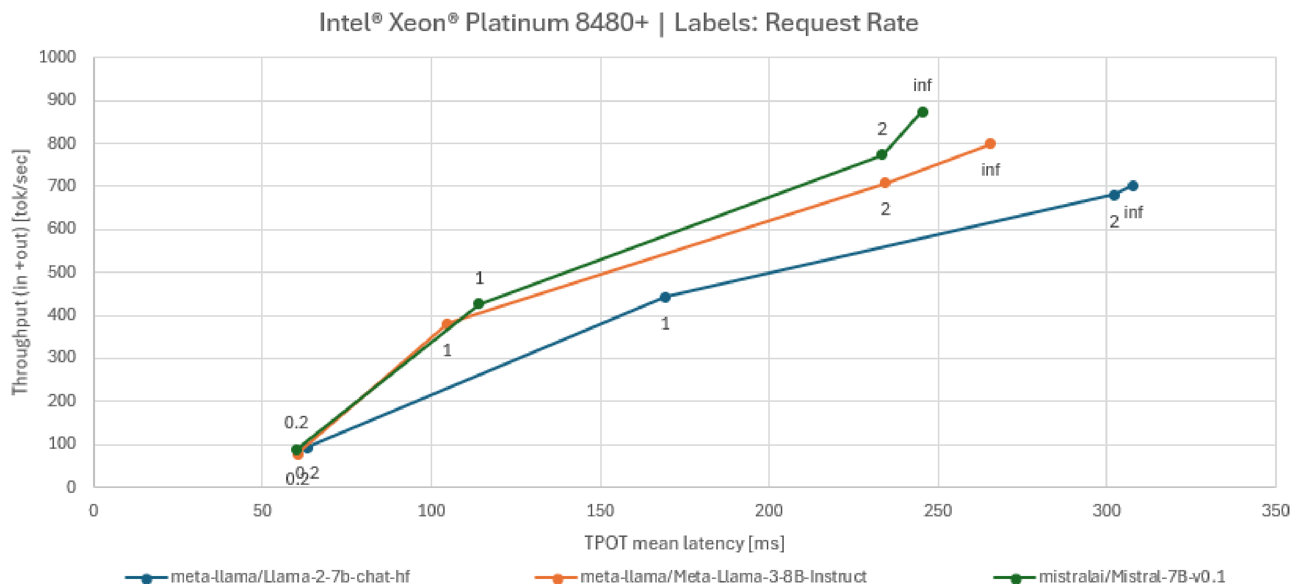

Figure 6. LLM benchmarking results with OVMS benchmarking client on Intel® Xeon® Platinum 8480+ CPU. *See Appendix for workloads and configurations. Results may vary.*

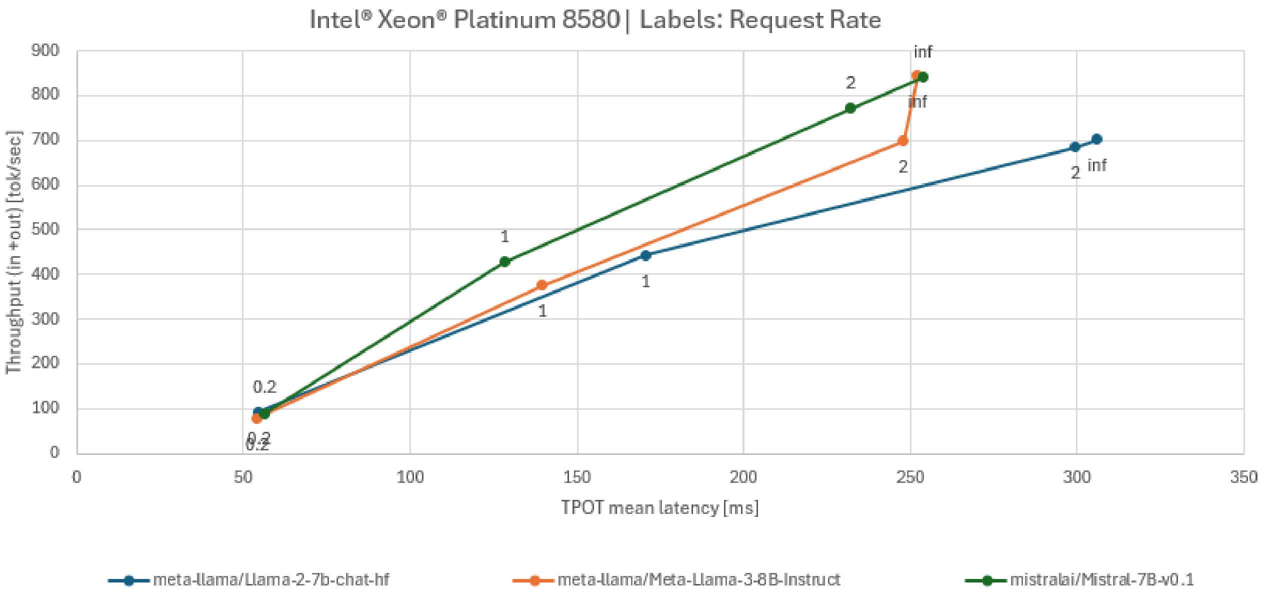

Figure 7. LLM benchmarking results with OVMS benchmarking client on Intel® Xeon® Platinum 8580 CPU. *See Appendix for workloads and configurations. Results may vary.*

Each colored line in the graph shows the throughput and mean latency for a model when tested at various request rates. For example, in Figure 7 the orange dot closest to the lower left-hand corner shows that the Meta-Llama-3-8B-Instruct model has a mean latency of 54 ms and throughput of 78.61 tokens/s when running at a request rate of 0.2 requests/s. Similarly, the same model has a mean latency of 139 ms and throughput of 376 tokens/s when running at a request rate of 1.0 requests/s.

The graph shows the general tradeoff between latency, throughput, and request rate. Using a higher request rate will increase the overall throughput but will also increase the latency. The LLM is able to process a higher number of requests more efficiently (thus increasing throughput), but it takes a longer time to deliver a response to an individual request (thus increasing latency). The graph in Figure 6 and 7 supports this: as the request rate for each model increases, throughput and latency also increase. A full table of numerical results can be found in Appendix 1: Full Benchmark Results.

5.3. Using Results

These results can be used to guide the design of a hosted LLM application and determine how many nodes are needed to support the required number of users. To determine the number of nodes required for hosting the application (whether they be on-premises servers or cloud servers), the developer should consider the total expected number of users and the maximum desired latency.

Latency is a key performance parameter: the higher the latency, the longer a user has to wait to receive a response. Applications with high latency can be perceived as laggy or low quality. Typically, a latency of 50 - 100 ms is considered acceptable to a user. Developers should specify the maximum average latency they want their users to experience, and then design the system to remain below that number. Let's continue the example for Acme Insurance presented in the introduction of this paper. They specify their maximum latency to be no greater than 100 ms.

Next, developers at Acme Insurance should estimate the peak request rate that their application will experience. For example, if 50 users are actively logged on at peak hours and submitting one request every 25 seconds, the average request rate will be [50 users] * [1 request / 25 s] = 2 requests per second.

Knowing the required request rate, the developer can estimate the expected latency for the model they are using. For example, if the application hosts a Meta-Llama-3-8B-Instruct model using OVMS, it will achieve a mean latency of about 234.44 ms at a request rate of 2 requests/s. (This value is from the graph in Figure 6 and Appendix 1 on Intel® Xeon® Platinum 8480+). Developers can also run their own benchmarking to determine the average latency at the required request rate.

Since the required request rate (2 requests/s) results in a mean latency (234.44 ms) that is higher than the max specification (100 ms), multiple hosted instances of the LLM will need to be used in the application. If two models are hosted, the effective request rate for each model is cut in half. Now the request rate is 1 requests/s, which results in a latency of 104.71 ms and almost satisfies the latency requirement. (This value is from Appendix 1).

In short, the model used, the required request rate, and the maximum allowable latency dictate the number of nodes that need to be hosted for the application. Developers can use benchmark data from Intel® or run their own benchmarking to determine the best design for their application.

6. References and Where to Learn More

These are the sources referenced throughout this paper. They provide more information on techniques for serving LLMs and the considerations to make when developing a hosted LLM application.

- [1] - Optimizing Large Language Models with the OpenVINO™ Toolkit
- [2] - LLM Retrieval-Augmented Generation (RAG) with OpenVINO™ and LangChain
- [3] - Gradio home page
- [4] - HuggingFace: Generation with LLMs
- [5] - Mastering LLM Techniques: Inference Optimization
- [6] - vLLM: Easy, Fast, and Cheap LLM Serving with PagedAttention
- [7] - Efficient Memory Management for Large Language Model Serving with PagedAttention

OpenVINO™ documentation also has more information on serving LLMs:

- Efficient LLM Serving
- Efficient LLM Serving - quickstart
- How to serve LLM models with Continuous Batching via OpenAI API

7. Appendices

Appendix 1: Full Benchmark Results

OpenVINO™ 2024.3 Test date: August 2024

Product	Model	Framework	Precision	Node	Request Rate	Throughput [tok/s]	TPOT Mean Latency [ms]
OVMS Benchmark Client	meta-llama/Llama-2-7b-chat-hf	PT	INT8-CW	Intel® Xeon® Platinum 8480+	0.2	92.63	63.23
OVMS Benchmark Client	meta-llama/Llama-2-7b-chat-hf	PT	INT8-CW	Intel® Xeon® Platinum 8480+	1	442.69	169.24
OVMS Benchmark Client	meta-llama/Llama-2-7b-chat-hf	PT	INT8-CW	Intel® Xeon® Platinum 8480+	2	680.45	302.09
OVMS Benchmark Client	meta-llama/Llama-2-7b-chat-hf	PT	INT8-CW	Intel® Xeon® Platinum 8480+	inf	702.42	307.82

OVMS Benchmark Client	meta-llama/Meta-Llama-3-8B-Instruct	PT	INT8-CW	Intel® Xeon® Platinum 8480+	0.2	78.3	60.37
OVMS Benchmark Client	meta-llama/Meta-Llama-3-8B-Instruct	PT	INT8-CW	Intel® Xeon® Platinum 8480+	1	380.61	104.71
OVMS Benchmark Client	meta-llama/Meta-Llama-3-8B-Instruct	PT	INT8-CW	Intel® Xeon® Platinum 8480+	2	707.46	234.44
OVMS Benchmark Client	meta-llama/Meta-Llama-3-8B-Instruct	PT	INT8-CW	Intel® Xeon® Platinum 8480+	inf	799.46	265.5
OVMS Benchmark Client	mistralai/Mistral-7B-v0.1	PT	INT8-CW	Intel® Xeon® Platinum 8480+	0.2	88.9	60.04
OVMS Benchmark Client	mistralai/Mistral-7B-v0.1	PT	INT8-CW	Intel® Xeon® Platinum 8480+	1	427.37	114.16
OVMS Benchmark Client	mistralai/Mistral-7B-v0.1	PT	INT8-CW	Intel® Xeon® Platinum 8480+	2	774.3	233.49
OVMS Benchmark Client	mistralai/Mistral-7B-v0.1	PT	INT8-CW	Intel® Xeon® Platinum 8480+	inf	873.93	245.31

OVMS Benchmark Client	meta-llama/Llama-2-7b-chat-hf	PT	INT8-CW	Intel® Xeon® Platinum 8580	0.2	92.89	54.69
OVMS Benchmark Client	meta-llama/Llama-2-7b-chat-hf	PT	INT8-CW	Intel® Xeon® Platinum 8580	1	442.46	170.65
OVMS Benchmark Client	meta-llama/Llama-2-7b-chat-hf	PT	INT8-CW	Intel® Xeon® Platinum 8580	2	684.4	299.41
OVMS Benchmark Client	meta-llama/Llama-2-7b-chat-hf	PT	INT8-CW	Intel® Xeon® Platinum 8580	inf	701.91	305.9
OVMS Benchmark Client	meta-llama/Meta-Llama-3-8B-Instruct	PT	INT8-CW	Intel® Xeon® Platinum 8580	0.2	78.61	54.12
OVMS Benchmark Client	meta-llama/Meta-Llama-3-8B-Instruct	PT	INT8-CW	Intel® Xeon® Platinum 8580	1	376.36	139.62

OVMS Benchmark Client	meta-llama/Meta-Llama-3-8B-Instruct	PT	INT8-CW	Intel® Xeon® Platinum 8580	2	698.38	247.77
OVMS Benchmark Client	meta-llama/Meta-Llama-3-8B-Instruct	PT	INT8-CW	Intel® Xeon® Platinum 8580	inf	843.51	252.12
OVMS Benchmark Client	mistralai/Mistral-7B-v0.1	PT	INT8-CW	Intel® Xeon® Platinum 8580	0.2	88.92	56.33
OVMS Benchmark Client	mistralai/Mistral-7B-v0.1	PT	INT8-CW	Intel® Xeon® Platinum 8580	1	427.85	128.33
OVMS Benchmark Client	mistralai/Mistral-7B-v0.1	PT	INT8-CW	Intel® Xeon® Platinum 8580	2	771.17	232.08
OVMS Benchmark Client	mistralai/Mistral-7B-v0.1	PT	INT8-CW	Intel® Xeon® Platinum 8580	inf	839.74	253.74

Appendix 2: Configuration Details

CPU Inference Engines:	Intel® Xeon® Platinum 8480+	Intel® Xeon® Platinum 8580
Motherboard	Intel® Corporation / Archer City	Intel® Corporation Archer City CRB
CPU	Intel® Xeon® Gold 8480+ CPU @ 1.9 GHz.	Intel® Xeon® Platinum 8580 CPU @ 2.0GHz
Hyper Threading	on	on
Turbo Setting	on	on
Memory	16 x 16 GB DDR5 4800MHz	12 x 16 GB DDR5 5600MHz
Operating System	Ubuntu* 22.04.4 LTS	Ubuntu* 22.04.4 LTS
Kernel version	6.5.0-41-generic	6.5.0-35-generic
BIOS Vendor	Intel Corporation	Intel® Corporation

BIOS Version	EGSDREL1.SYS.9409.P31.2302280828	EGSDREL1.SYS.1752.P05.2401050250
BIOS Release	2/28/2023	1/5/2024
NUMA nodes	2	2
Test Date	8/3/2024	8/3/2024

https://docs.openvino.ai/2024/_static/benchmarks_files/OV-2024.3-platform_list.pdf

8. Notices & Disclaimers

Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more on the Performance Index site.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See Appendicies for configuration details. No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.