

[← Knowledge Hub](#)

Tech stories

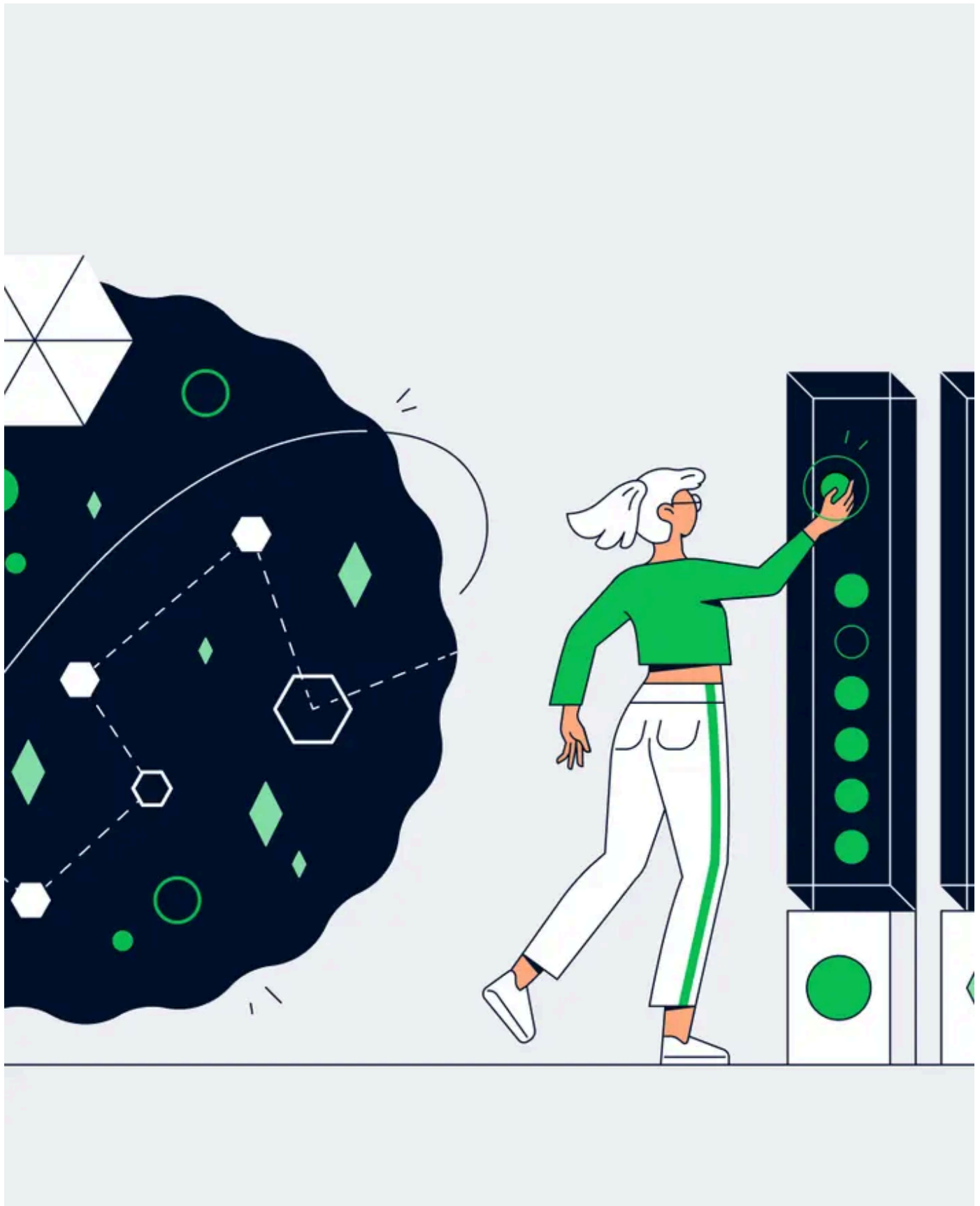
LLM inference at scale with TGI

By Martin Iglesias Goyanes, Machine Learning Engineer

September 3rd, 2024 · 20 minutes



[Pricing](#)



[LLM Inference Overview](#) [TGI: In Depth](#) [TGI: Relevant Metrics](#) [Takeaways](#) [Reference](#)

[Tech stories](#) [Dive into tech](#)



Optimizing Large Language Models (LLMs) for efficient inference is a complex task, and understanding the process can be equally challenging. This article is for those who want to look beyond the surface-level understanding of [Text Generation Inference \(TGI\)](#) by HuggingFace, an efficient and optimized solution for deploying LLMs in production. At Adyen, TGI has been adopted as our go-to approach for LLM inference in our internal GenAI Platform.

As was already discussed [in a previous article](#), some of the key advantages derived from its open-source nature are: cost savings, enhanced data privacy, control of the technology and flexibility for customization. This open-source ethos aligns with a commitment to transparency and collaborative advancement in the AI community.

We will start with a quick refresher on LLM inference, covering the key steps of **prefill** and **decode**. Then, we'll introduce TGI and dive deep into its two main components: the **server** and the **inference** engine. We will also provide insights into **relevant metrics** and performance considerations. Finally, we will offer key takeaways to summarize the discussion. The aim is to provide a detailed yet concise guide, offering valuable insights and practical takeaways for anyone looking to maximize the potential of LLMs in production with



[Pricing](#)

LLM Inference Overview

Prefill

During the Prefill stage, the input **prompt is tokenized** on the CPU and then transferred to the GPU. Tokenization is the process of converting the words into smaller units, known as tokens, which the model can process more efficiently. For example, given the prompt, "What is the capital of the US?" The model tokenizes the sentence and processes **it in one forward pass through the loaded model** on the GPU, generating an initial token. This initial pass is relatively quick as it only requires a single pass through the model to produce the first token, such as "Washington" in response to the prompt.

Decode

The Decode stage is where the **autoregressive** nature of LLMs comes into play. In this stage, the model generates text one token at a time, building upon the initial token from the Prefill stage. Each newly generated token is appended to the input sequence, creating a new context for the model to process. For example, as shown in Figure 1, after generating "Washington" as the initial token, the new sequence becomes, "What is the capital of the US? Washington". This updated sequence is then used to generate the next token. The model continues this process iteratively, with each new token influencing the generation of the next. This autoregressive approach allows the model to maintain context and generate coherent responses. The Decode stage continues until an end-of-sequence (EOS) token is generated, or the maximum sequence length, specified by *max_new_tokens*, is reached. At this point, the generated sequence is de-tokenized on the CPU, converting the tokens back into readable text.

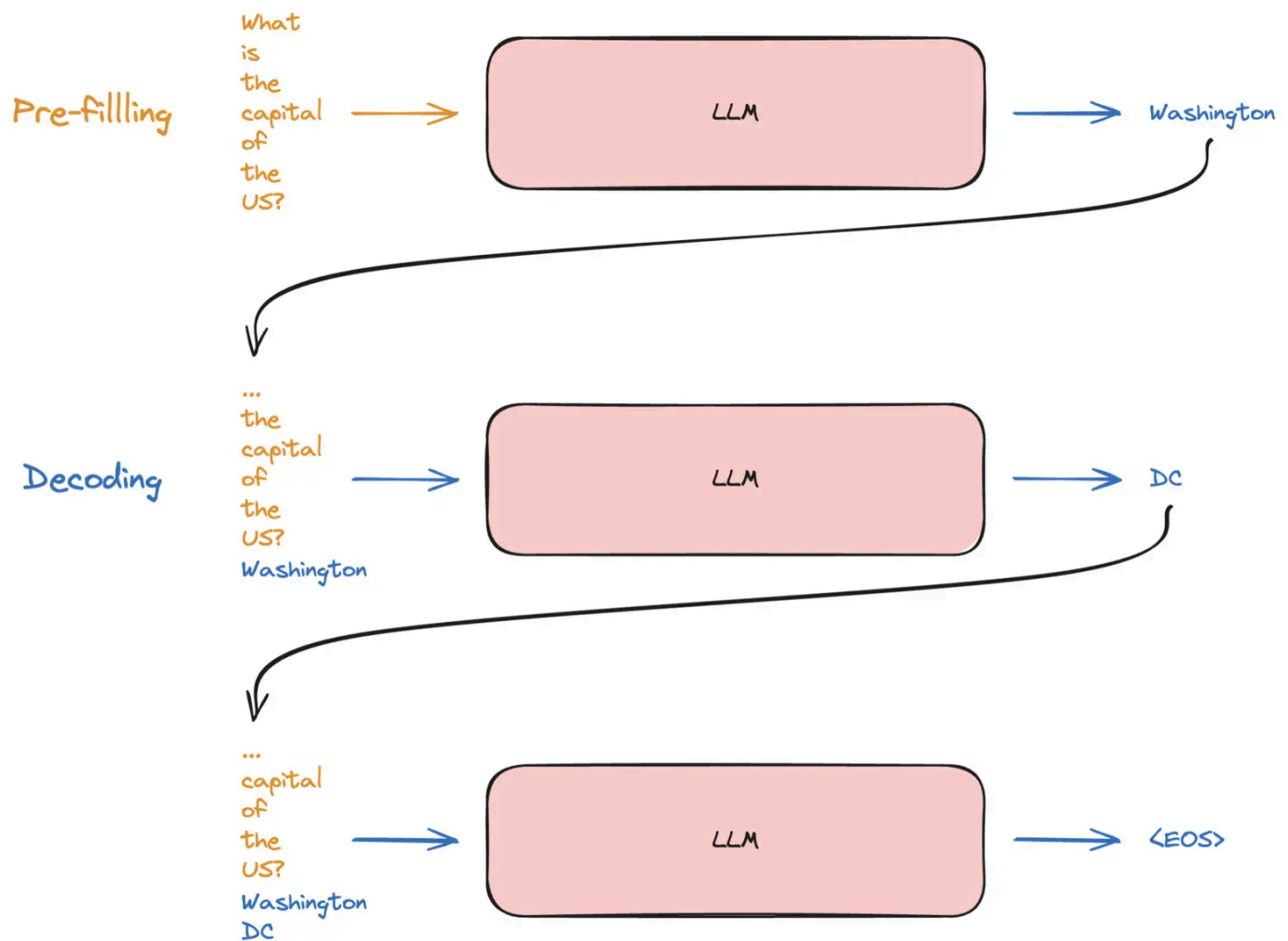


Figure 1: Prefill and Decode flow [1]

Why Separate Prefill and Decode?

The separation of the Prefill and Decode stages is essential due to the distinct computational characteristics of each stage. While the Prefill stage requires only a single forward pass, the Decode stage involves multiple passes, each dependent on the previously generated tokens. This autoregressive nature of the Decode stage contributes to longer processing times, and the computational expense scales quadratically with the total sequence length.

To optimize this process and mitigate quadratic scaling, a technique called **KV caching** [6] is employed. KV caching saves intermediate states, known as KV caches, generated at each token position during both the Prefill and Decode stages. By storing these KV caches in GPU memory, the model avoids the need to recompute them, reducing computational

efficiency and helping to manage the longer processing times associated with autoregressive token generation.

TGI: In Depth

TGI integrates numerous state-of-the-art techniques to provide smooth, **low-latency**, and **high-throughput** inference, making it an ideal choice for production environments where performance and scalability are critical. It offers a simple yet versatile launcher to serve various LLMs, along with distributed tracing via Open Telemetry and Prometheus metrics for comprehensive monitoring. TGI supports advanced attention mechanisms like **Flash Attention** and **Paged Attention**, ensuring optimized and efficient inference. The framework also provides fine-grained control through various arguments and per-request configurations, such as guided decoding for structured output generation.

When serving LLM-based applications, model serving can be divided into two main components: the engine and the server (as illustrated in Figure 2). The engine handles everything related to the models and batching requests, while the server focuses on forwarding user requests. In TGI, these components are named accordingly: the **server** is referred to as the router, and the **engine** is called the text_generation_server.

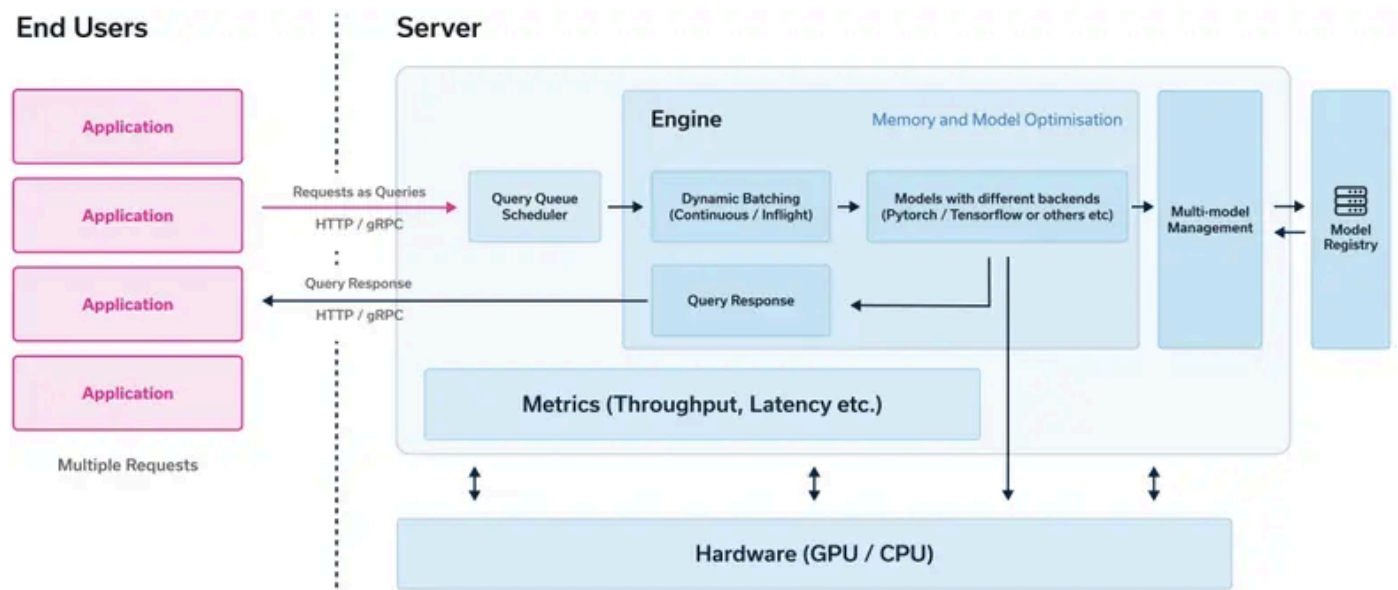


Figure 2: Architecture of an LLM backend [2]

The primary purpose of TGI router is to manage incoming requests and prevent the engine from encountering memory-related issues and ensuring smooth and efficient LLM inference. It employs a **smart continuous batching algorithm, dynamically adding requests to the running batch to optimize performance**. This dynamic batching approach strikes a balance between latency and throughput.

Upon initialization, the router triggers a warm-up phase on the inference engine. We'll cover that on the next section, but basically during this phase, the router determines the maximum capacity of the underlying hardware (GPU) for the deployed LLM:

- **MAX_BATCH_PREFILL_TOKENS**: The maximum number of tokens the GPU can handle in a single forward pass during the prefill stage.
- **MAX_BATCH_TOTAL_TOKENS**: The maximum tokens that can be processed concurrently during both prefill and decode steps.

The router's continuous batching algorithm is designed to prevent Out Of Memory (OOM) errors. Unlike static batching, where requests wait for the previous batch to complete, continuous batching allows for the dynamic addition of new requests to the running batch. That means that *"With continuous batching you can find a sweet spot. In general latency is the most critical parameter users care about. But a 2x latency slowdown for 10x more users on the same hardware is an acceptable trade off" [3]*

The logic behind the router's dynamic batching is illustrated in the provided pseudocode:

Language: python

```
1      # Initialize the batch and token budgets
2      batch = []
3      token_budget = max_batch_total_tokens
4
5      # Function to add requests to the prefill batch until the max_tokens budget is reached
6      def add_requests_to_prefill_batch(requests, batch, max_tokens):
7          while requests and sum(request.tokens for request in batch) < max_tokens:
8              batch.append(requests.pop(0))
9          return batch
10
11     # Add initial requests to the prefill batch
12     batch = add_requests_to_prefill_batch(request_queue, batch, max_batch_prefill_tokens)
```

Copy

```
15  prefill(batch)
16
17  # Main loop to manage requests
18  while batch:
19      # Update the token budget based on current batch
20      batch_max_tokens = sum(request.input_tokens + request.max_new_tokens for request in batch)
21      token_budget = max_batch_total_tokens - batch_max_tokens
22
23      # Add new requests to the batch based on token budgets
24      new_batch = add_requests_to_batch(request_queue, [], min(max_batch_prefill_tokens, token_budget))
25
26      # If new requests were added, handle prefill and decoding
27      if new_batch:
28          # Stop decoding and prefill the new batch
29          prefill(new_batch)
30
31          # Extend the original batch with the new requests
32          batch.extend(new_batch)
33
34      # Decode the current batch
35      decode(batch)
36
37      # Filter out completed requests that have reached EOS or max_new_tokens
38      batch = [request for request in batch if not request.reached_EOS and request.tokens_generated <
39 request.max_new_tokens]
40
41      # Update token budget by subtracting tokens from completed requests
42      completed_requests = [request for request in batch if request.reached_EOS or
43 request.tokens_generated >= request.max_new_tokens]
44      for request in completed_requests:
45          token_budget = token_budget - request.input_tokens + request.tokens_generated
```

To better illustrate how TGI's continuous batching algorithm works, let's walk through a specific example with the following initial setup seen in Table 1. Initially, no requests are being processed so the total token budget is equal to *MBT*.

the 16th request, with a smaller token count, fits within the *TTB* and successfully prefills the cache, joining the running decoding batch. At this point, the token budget is fully utilized, and we must wait for currently running requests to complete.

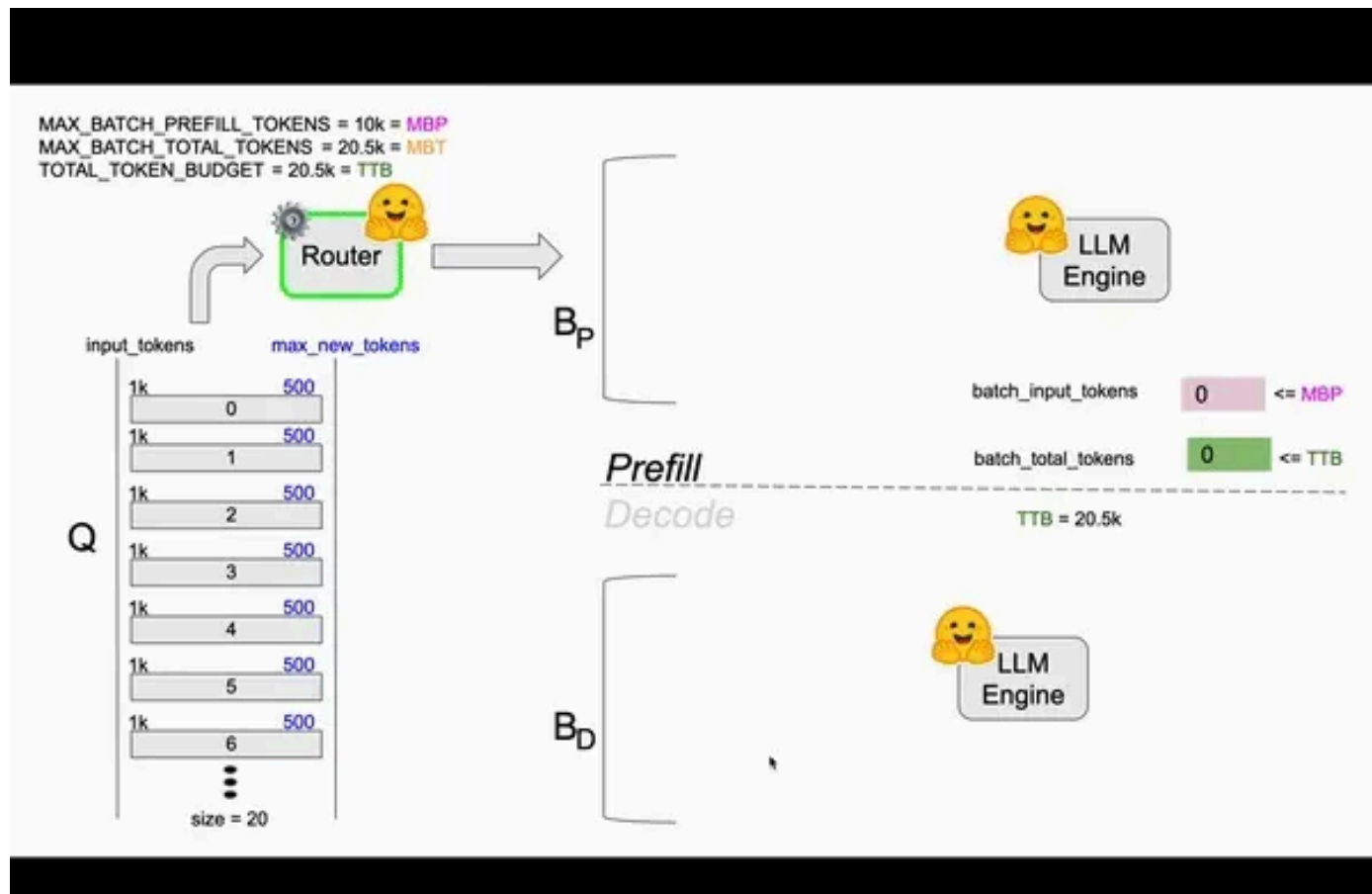


Figure 4: TGI Continuous Batching animation based on TGI router code.

Eventually, in figure 5, requests 0th, 9th, and 16th finish processing, freeing up token budget space. This allows requests 14th and 15th to proceed with prefill and decoding, leaving a *TTB* of 1,000 tokens. As the process continues, more requests complete, freeing up the budget for the remaining requests in the queue (17th, 18th, and 19th) to be processed.

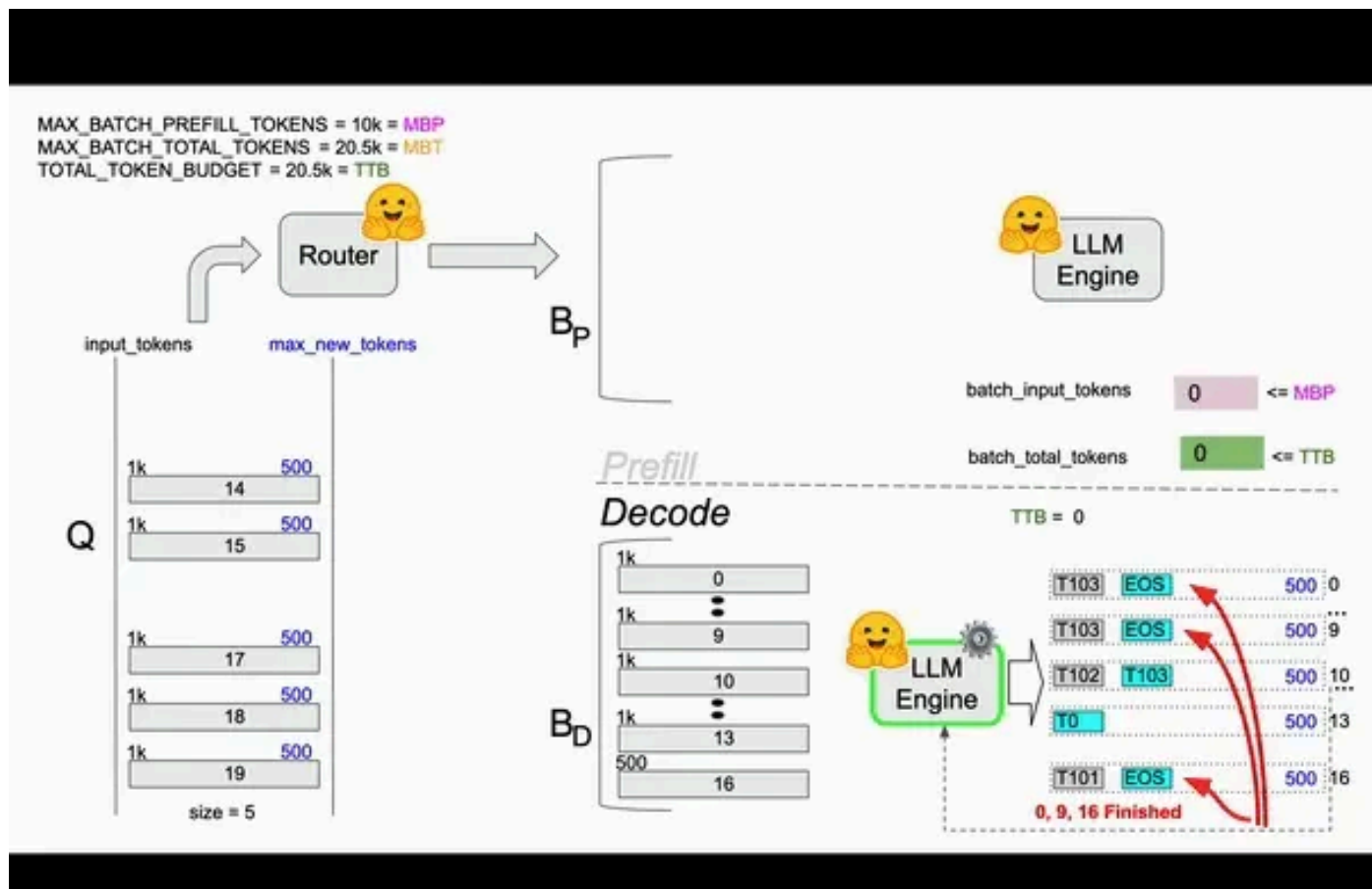


Figure 5: TGI Continuous Batching animation based on TGI router code.

One important observation is worth noting from Figure 3. The first 10 requests (0th to 9th) underwent the prefill step together, yet they did not saturate the available *TTB* of 20.5k tokens. This raises the question: why weren't more requests added to the batch? The answer lies in the token budget for a single forward pass, or *MBP*. Those 10 requests saturated the *MBP*, which is specific to the prefill stage. In later steps, the router adds requests to fill the memory for the decoding step, but these requests couldn't be included earlier as they would have exceeded the *MBP* budget. This scenario highlights the difference between *MBP* and *MBT*: while *MBP* focuses on the prefill stage, *MBT* represents the total token budget, with decoding benefiting from memory optimizations.

The distinction between *MBP* and *MBT* can be further explained by considering the nature of the prefill and decode stages. In the prefill step, the LLM engine processes *sum(req.input_tokens for req in requests)*. For instance, with 4 requests, each with 500 *input_tokens* and 500 *max_new_tokens*, the batch of 4 results in 2000 tokens processed in the prefill stage and another 2000 tokens to decode. This seems confusing as both stages

During prefill, the engine performs a full forward pass across all 2000 tokens to obtain the attention queries, keys, and values for each input token, leading to the output of the first decoded token for each sequence. In contrast, during decoding, the Nth token benefits from the KV Cache, where all previous tokens' attention keys, queries, and values are already cached. Thus, decoding is like running a forward pass on just one token, the Nth token. As decoding is autoregressive, it proceeds token by token, making the generation of 2000 tokens for 4 sequences akin to processing only 4 tokens concurrently. In comparison, prefill requires forwarding all 2000 tokens through the model for the first new token generation.

TGI offers configurable parameters to fine-tune the behavior of the prefill and decode stages for specific use cases. These parameters, set as environment variables (*WAITING_SERVED_RATIO*, *MAX_WAITING_TOKENS*, and *MAX_BATCH_SIZE*), allow for customization of the trade-offs between the two stages.

The implementation of continuous batching at the server level, using Rust, is a strategic choice by TGI developers. Rust's speed is your best ally in this case since Python would be adding some milliseconds per decision. More precisely, strict typing and real concurrency are what give Rust a huge boost over Python. When thinking of scale, this decision can happen 100x times for a single batch of requests which would add 100s of ms to the end to end latency.

The Inference Engine: Warmup and inference optimizations

The inference engine is the one in charge of processing the requests coming from the router. Essentially, it loads the model into the GPU's memory and then, runs the prefill and decode stages. We will cover what we consider are the most important features of TGI's inference engine: *warmup*, *kv caching*, *flash* and *paged attention*.

Warmup

This phase is run before starting to process any requests. First, it estimates the appropriate token budget based on the available hardware and the deployed model so that no OOM errors occur during inference. Also, if enabled, it records CUDA GRAPHS for LLM forward passes on a set of batch sizes: on a high level this is an efficient way of recording GPU operations for fixed size inputs, i.e batch sizes, reducing the overhead of CPU-GPU communication when replayed [4]. In order to estimate the prefill token budget, the engine adds requests of input tokens = max input tokens and max new tokens = max total tokens



Pricing

decrease *MAX_BATCH_PREFILL_TOKENS*. When this is done successfully, TGI goes on to estimating the total token budget.

For the total token budget estimation, the engine maps available memory to a total count of processable tokens. First the engine calculates 95% of the available VRAM, leaving 5% room for error, where *Available VRAM = GPU VRAM - Model VRAM - Prefill KV Cache VRAM*. The available memory is then divided by the memory required to process a block of tokens [5] yielding the total number of tokens that can be processed simultaneously. This value is set as the *MAX_BATCH_TOTAL_TOKENS*, essentially the tokens that in a block times the number of blocks that fit into memory.

Inference Optimizations

Additionally, in the case of TGI, this engine already comes with the common state-of-the-art algorithms for optimized LLM inference such as: Paged Attention [5], and Flash Attention [7].

PagedAttention addresses the memory-bound nature of LLMs by optimizing how memory is managed during inference. In a GPU, every memory movement impacts latency and throughput, and recreating KV-cache tensors for each request would be inefficient. PagedAttention splits the KV-cache into *N* pages, allowing each request to use *n* pages that are released upon completion. This paging system eliminates the need to re-allocate tensors, instead reusing pages for new requests, which reduces unnecessary memory movements. Although this may hurt cache locality in the kernels, the reduction in memory re-allocation makes the trade-off worthwhile [5].

FlashAttention is a valuable, though not critical, optimization at LLM inference time. Its primary impact lies in enabling the use of padless tensors. Previously, attention computation required tensors of shape *[batch_size, seq_len, ...]*, which required padding the shorter sequences to match the longest one, leading to increased memory movement and VRAM usage due to these added pad tokens. FlashAttention eliminates this need, significantly reducing VRAM consumption. While the SRAM benefits highlighted in the FlashAttention paper are most advantageous during training, which is compute-bound, the reduced VRAM usage and enhanced efficiency still provide considerable performance boosts during inference, especially with long sequences [7].



Latency and throughput drivers

Remember! LLM inference involves two key stages: **Prefill** and **Decode**. The prefill speed impacts the **Time To First Token** (TTFT), as token generation cannot begin until the input context has been processed. Then, the decoding speed influences the **Time Per Output Token** (TPOT), which measures the rate at which tokens are generated after the prefill. Both TTFT and TPOT are critical for user experience and play a vital role in defining LLM inference performance. Additionally, inference performance is also affected by throughput which is driven by memory, also known as GPU's VRAM. Available VRAM is largely determined by size of the model and the KV-cache. **VRAM** usage directly impacts the maximum batch size and sequence length.

In summary, LLM inference is characterized by **VRAM usage**, **TTFT**, and **TPOT**. To estimate these metrics, one must consider the data volume to be processed and the FLOPs (Floating Point Operations) required for computation.

GPUs: High level overview

In order to understand the following section, you need to know at least on a high level what a GPU does. Keeping it simple, it loads data (from GPU memory known as **HBM**/VRAM into the compute unit's SRAM) and computes **FLOPs** (mathematical operations like matrix multiplications). These operations are limited by how much memory per second the HBM can "move" and by how many FLOPs per second the SM can do [11]. A very important concept to remember is compute bound versus memory bound. A job is said to be memory bound if memory can not supply work at a rate to keep the processor busy. On the contrary, a job is said to compute bound if its bottleneck by the speed of the processor.

Metrics computation

Now is where we will see the big difference between prefill and decode, and how their separation impacts performance. Prefill loads the model once from memory to process all input tokens in parallel, which leads to a **compute bound** process with a high number of operations per byte read. In contrast, decode is a **memory bound** process since it loads the model *max_new_tokens* times, once for every single token generated (low number of ops per byte read) [9].

Let's assume we are serving **LlaMa-7b** using **16-bit** precision on an **A100 GPU**. We are going to compute the VRAM requirements and the different timings: prefill, decode, TTFT, TPOT

Variable	Value
S : input sequence tokens	512
O : output tokens	1024
B : batch size	4
H : hidden dimensions	4096
L : number of transformer layers	32
N : model parameters	7e9
GPU FLOPs Rate	312 TFLOPs/s
GPU High Bandwidth Memory Rate	1.5 TB/s

Table 2: Token load, model and hardware characteristics.

To derive the TTFT, TPOT and total times we first need to compute the prefill and decode times. Each of the prefill and decode stages have both a compute and a memory time. In terms of compute, a token's embedding needs to be multiplied with the model's weight matrix or parameters; this accounts for N computations. So for prefill step where we process the whole input of all sequences in a batch, we have $B*S$ tokens, therefore we perform $N*B*S$ calculations [10]. On the other hand, for decode step we only process one token at a time for each of the sequences in the batch, which is $B*1$ tokens, therefore we perform $N*B*1$ computations. We can't forget, though, that we are using 16-bit precisions which means for each computation we are using 2 bytes. In contrast, for memory time, we need to load the N model parameters into memory, each of those stored in 2 bytes (16-bit precision). A summary of the operations is shown in Table 3.

Stage	Compute Time	Memory Time	Type
Prefill	$\frac{2 * N * B * S}{FLOPs\ rate} = 91.9\ ms$	$\frac{2 * N}{HBM\ rate} = 9.3\ ms$	Compute bound
Decode	$2 * N * B * 1$	$2 * N$	Memory bound

Table 3: Math behind Compute and Memory types of Prefill and Decode stages

Now that we have these, we can compute TTFT, TPOT and total time. In Table 4, we take the maximum between compute and memory times, since they overlap among each other and the longest one is the dominant time that makes the process compute or memory bound.

Metric	Value
TTFT: Time To First Token	$\max(\frac{2*N*B*S}{FLOPs\ rate}, \frac{2*N}{HBM\ rate}) = 91.9\ ms$
TPOT: Time Per Output Token	$\max(\frac{2*N*B*1}{FLOPs\ rate}, \frac{2*N}{HBM\ rate}) = 9.3\ ms$
Total Time	$TTFT + TPOT * O = 9.5s$

Table 4: Math behind TTFT, TPOT and Total Time

We have so far made the calculations affecting latency, let's look into the ones that impact throughput. For that we will compute how much VRAM is available for inference, the more available, the more tokens we can process in parallel. Remember that we are using 2 byte precision and A100 has 80GB VRAM. As you see in Table 5 before processing any request, the KV cache is empty so the VRAM is only holding the *model_size* = 2*NGBs. Once TGI prefills for a batch of requests the VRAM usage increases *kv_cache_size* over *model_size*. The KV Cache size shown in Figure 6 is explained as follows: for each token there are two vectors, one for key and one for the value, each of these vectors exist in each of the attention heads *L* with dimension *H*. Initially, after the prefill, there are *B*S* tokens.

$$KV\ Cache\ Size = \underbrace{2}_{\substack{\text{16-bit precision so} \\ \text{2-bytes per token}}} * \underbrace{2}_{\substack{\text{2 vectors per token:} \\ \text{keys and values}}} * \underbrace{L}_{\substack{\text{each vector in L} \\ \text{attention heads}}} * \underbrace{H}_{\substack{\text{each attention head} \\ \text{has H dimensions}}} * \underbrace{B}_{\substack{\text{batch size}}} * \underbrace{S}_{\substack{\text{each request in} \\ \text{batch has S input} \\ \text{tokens}}}$$

Figure 6: Math behind KV Cache Size for prefill tokens linked to transformer components. Inspired from [10]

Eventually, in Figure 7, when TGI finishes decoding *kv_cache_size* would have grown proportional to *S+O*.

$$\text{KV Cache Size} = \underbrace{2}_{\substack{\text{16-bit precision so} \\ \text{2-bytes per token}}} * \underbrace{2}_{\substack{\text{2 vectors per token:} \\ \text{keys and values}}} * \underbrace{L}_{\substack{\text{each vector in L} \\ \text{attention heads}}} * \underbrace{H}_{\substack{\text{each attention head} \\ \text{has H dimensions}}} * \underbrace{B}_{\substack{\text{batch size}}} * \underbrace{(S+O)}_{\substack{\text{each request in} \\ \text{batch has S input} \\ \text{tokens} \quad \text{each request} \\ \text{grows up to O} \\ \text{tokens}}}$$

Figure 7: Math behind KV Cache Size for total tokens, linked to transformer components. Inspired from [10]

Inference stage	VRAM Used	VRAM Free
Before Prefill	$model_{size} = 14\text{ GB}$	$80 - 14 = 66\text{GB}$
After Prefill	$model_{size} + kvCache_{size} = 14.12\text{ GB}$	$80 - 14.12 = 65.88\text{GB}$
After Decode	$model_{size} + kvCache_{size} = 14.6\text{ GB}$	$80 - 14.6 = 65.4\text{GB}$

Table 5: Math behind VRAM usage.

As we see in Table 5, in our example, since the A100 GPU has 80GB of VRAM, we can comfortably handle such a token load. However, if we increase the token load to $S=3000$, $O=2000$ and $B=32$, this results in $VRAM\text{ Used} = 14\text{GB} + 67\text{GB} = 83.8\text{GB} > 80\text{GB}$. Therefore, we can not handle this token load on a single A100 GPU. We must either use a smaller model, a GPU with more VRAM, we leverage tensor parallelism across more hardware or we could quantize our model weights.

Relevant metrics per use case

Depending on the use case of your downstream application you will care about different performance metrics. For example, if you are serving a RAG application then you will probably care much about latency and less about throughput, in particular you will care about TTFT and TPOT to be faster than the end user's read speed. Alternatively, if you have an application that summarizes every incoming ticket sent to the customer support area, then you care about the total time it takes the complete summary to be ready. In such a case, your use case is less dependent on TTFT and more on TPOT multiplied by the amount of tokens the summary needs. On the other hand, if you are processing financial documents overnight for classification then you care mostly about how many documents you can fit at once, i.e you will completely disregard latency and only care about throughput.

do in Figure 8, keep it simple, how many tokens go in the model? How many come out? It's not the same to have a simple chat than a RAG app.

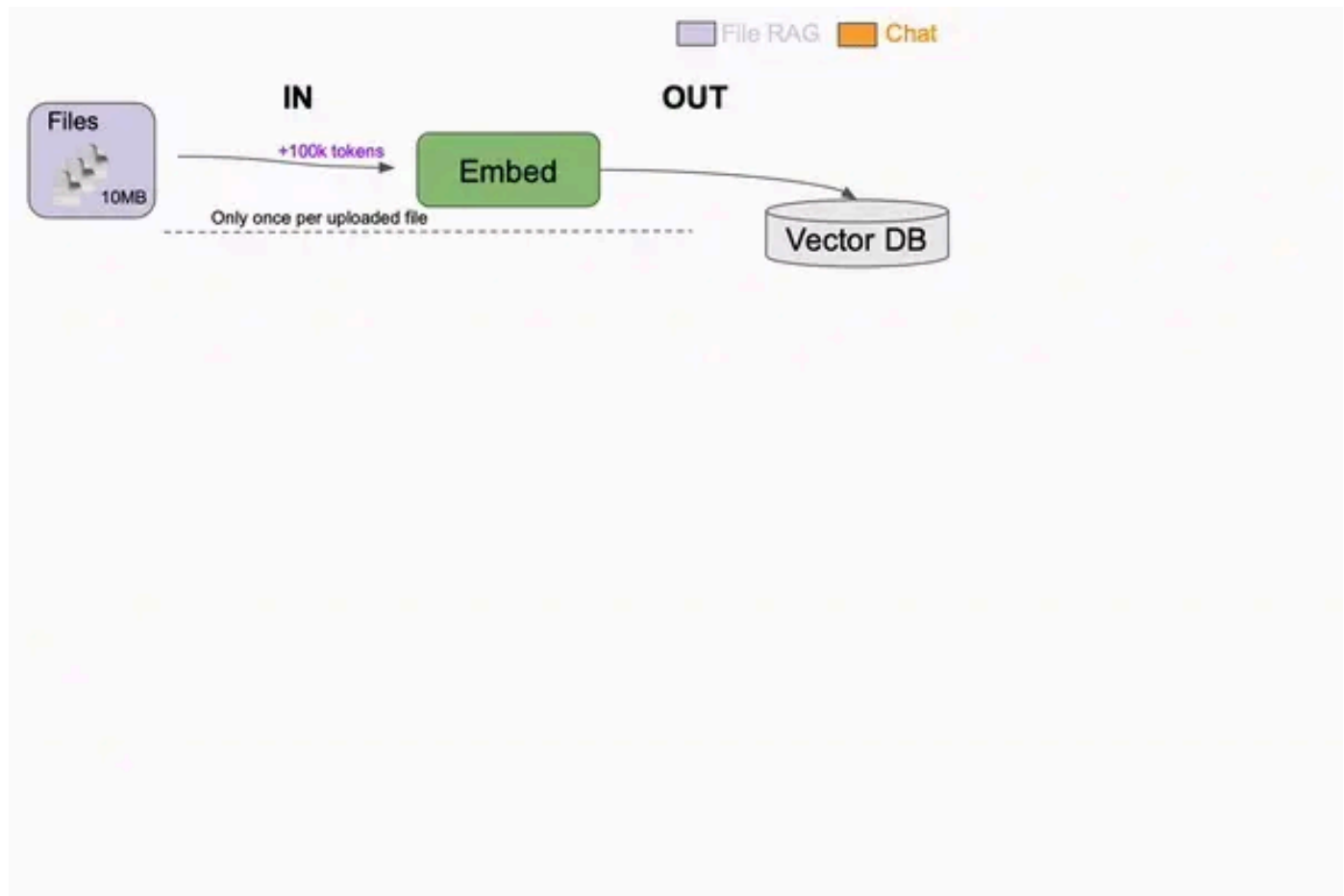


Figure 8: Comparison of token budgets of chat vs file RAG applications.

For example in Figure 8, we compare the amount of tokens to be processed by a file RAG application versus just a chat application. A file RAG app also needs a chat interface to allow the user to write queries about the uploaded file, so we distinguish in purple what is explicitly needed for the RAG app and in orange what is needed for a chat app. We can see how total input tokens are 109k if we consider the initial file upload, if we don't consider, then it is just 9k tokens. However, if we only count the orange tokens, we see that a chat app only needs 5k input tokens and 1k output tokens, which is almost half of what the file RAG app needs.

- The autoregressive nature of the decode step is the key bottleneck for latency and throughput. In order to alleviate these, TGI has adopted many techniques to cut down latency and bring up throughput while decoding: Paged Attention [5], KV Caching [6] and Flash Attention [9] among others.
- TGI's *router* takes advantage that generations can finish unexpectedly because of an EOS token and decode token budget is larger than prefill token budget. Therefore, instead of static batching, it continuously batches requests to the *inference engine* intertwining prefill-decode steps and filters away finished requests.
- The LLM and GPU chosen are the most important drivers of performance: throughput and latency. More precisely, performance is a function of the LLM parameters size, the GPU's High Bandwidth Memory and the GPU's FLOPs.
- It is critical to think in tokens and not requests when working with TGI. This means to understand the flow of tokens in your use case and find the relevant per-token metrics you need to optimize for.
- TGI's benchmarking tool is great for getting familiar with main bottlenecks affecting your use case. However, it is skipping the *router* (not leveraging continuous batching), in order to test TGI as a whole, *router* and *inference engine* together, it's preferable to use a load testing tool such as k6.

References

[1] Thomas, D. (2024, May 29). *Benchmarking Text Generation Inference*. Hugging Face. Retrieved June 29, 2024, from <https://huggingface.co/blog/tgi-benchmarking>

[2] *What it means to serve an LLM and which serving technology to choose from*. (2024, January 9). Run:ai. Retrieved June 29, 2024, from <https://www.run.ai/blog/serving-large-language-models>

[3] Patry, N. (2023, May 1). *TGI Router Documentation*. Github. <https://github.com/huggingface/text-generation-inference/blob/main/router/README.md>

[4] Reed, J. K., Dzhulgakov, D., & Morales, S. (2023, August 29). *Speed, Python: Pick Two. How CUDA Graphs Enable Fast Python Code for Deep Learning*. Fireworks.ai. Retrieved



Pricing

- [5] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., & Stoica, I. (2023, September 12). *Efficient memory management for large language model serving with paged attention*. arXiv.org. <https://arxiv.org/abs/2309.06180>
- [6] Lienhart, P. (2023, December 22). *LLM Inference Series: 3. KV caching explained* | by Pierre Lienhart. Medium. Retrieved June 29, 2024, from <https://medium.com/@plienhar/llm-inference-series-3-kv-caching-unveiled-048152e461c8>
- [7] Dao, T., Fu, D. Y., Ermon, S., Rudra, A., & Ré, C. (2022, June 23). *FlashAttention: Fast and memory-efficient exact attention with IO-awareness*. arXiv.org. <https://arxiv.org/abs/2205.14135>
- [8] Hugging Face. (n.d.). *Flash Attention*. Hugging Face. Retrieved June 30, 2024, from https://huggingface.co/docs/text-generation-inference/en/conceptual/flash_attention
- [9] Chen, J. (2023, December 19). *Estimate LLM inference speed and VRAM usage quickly: With a llama-7b case study*. <https://www.jinghong-chen.net/estimate-vram-usage-in-llm-inference/>
- [10] Chen, Carol. (2022) "Transformer Inference Arithmetic", <https://kipply.com/blog/transformer-inference-arithmetic/>

Bibliography

- KV Cache explained: [LLM Inference Series: 3. KV caching explained | by Pierre Lienhart | Medium](#)
- Advanced Transformer Inference walkthrough: [Transformer Inference Arithmetic | kipply's blog](#)
- Latency and Throughput estimations: [Estimate LLM inference speed and VRAM usage quickly: with a Llama-7B case study](#)
- Advanced Transformer Training walkthrough: [Transformer Math 101 | EleutherAI Blog](#)
- PagedAttention broken down: [vLLM Paged Attention](#)
- Benchmarking Study of LLM inference frameworks: [A Benchmarking Study: Which](#)